PROJECT TECHNICAL REPORT
for
DBT Project BT/PRO363/BID/18/003/96

Title	**:**	**Design and Implementation of a Bio-diversity Database Management System**
Sponsor	**:**	**Dept. of Bio-Technology, Government of India**
Principal Investigator	**:**	**Jayant Haritsa Associate Professor Supercomputer Education & Research Centre Indian Institute of Science**
Co-Investigators	**:**	**Prof. Madhav Gadgil, CES, IISc**
	:	**Prof. V. Nanjundiah, CES, IISc**

June 2001

Centre for Sponsored Schemes and Projects

Indian Institute of Science

Bangalore 560012, India

Contents

1 Overview of Project **1**
- 1.1 Design Goals . . . 1
- 1.2 Design Description . . . 2
  - 1.2.1 Object Services . . . 2
  - 1.2.2 Spatial Services . . . 5
  - 1.2.3 Sequence Services . . . 7
- 1.3 Implementation Details . . . 8
  - 1.3.1 Spatial and Sequence primitives . . . 8
  - 1.3.2 Indexes over Objects . . . 8
  - 1.3.3 Spatial Services . . . 10
  - 1.3.4 Sequence Services . . . 10
- 1.4 Query Processing . . . 10
  - 1.4.1 Schema Definition . . . 10
  - 1.4.2 Query Flow . . . 10
- 1.5 Client Interface Framework . . . 12
- 1.6 Current Status . . . 13
- 1.7 Related Work . . . 13
- 1.8 Summary . . . 13

2 The SHORE System **15**
- 2.1 Drawbacks with EXODUS . . . 15
- 2.2 How SHORE differs from EXODUS . . . 15
- 2.3 The SHORE Architecture . . . 16
- 2.4 SHORE Software Components . . . 16
  - 2.4.1 The Language Independent Library . . . 16
  - 2.4.2 The SHORE server . . . 17
- 2.5 Value Added Server . . . 17
- 2.6 The VAS-SM Programming Interface . . . 17
  - 2.6.1 Storage Facilities . . . 17
  - 2.6.2 Transaction Facilities . . . 18
  - 2.6.3 Crash Recovery Facilities . . . 19
  - 2.6.4 Thread Management . . . 19
  - 2.6.5 Communication and RPC Facilities . . . 19
- 2.7 Conclusion . . . 19

3 Handling Taxonomy Data **20**
- 3.1 Taxonomy Data Model . . . 20
  - 3.1.1 Analysis . . . 20
  - 3.1.2 The Object Model . . . 21
  - 3.1.3 Class Hierarchy Indexing Methods . . . 23
  - 3.1.4 Execution of Queries . . . 25
  - 3.1.5 Advantages of MT-index . . . 25
- 3.2 Aggregation Hierarchy Indexing Methods . . . 25
  - 3.2.1 Basic Problems . . . 26
  - 3.2.2 Related Work . . . 26
  - 3.2.3 Our Choice: Path Dictionary Index . . . 27
  - 3.2.4 Implementation of the Data Structures . . . 29
  - 3.2.5 Execution of Queries . . . 30
  - 3.2.6 Advantages of Path Dictionary Index . . . 31
- 3.3 Implementation Issues . . . 32
  - 3.3.1 Data Model . . . 32

3.3.2 Path Dictionary Index . . . 32
3.3.3 MT-index . . . 32
3.3.4 Summary . . . 33

4 Handling Spatial Data **34**
4.1 Modeling of Spatial Data . . . 34
4.1.1 The Object Model . . . 34
4.1.2 Spatial Data Types and Relationships . . . 34
4.2 Spatial Access Methods . . . 37
4.2.1 R-Tree . . . 38
4.2.2 Hilbert R-tree . . . 39
4.3 Implementation Details . . . 42
4.3.1 Object Model implementation . . . 43
4.3.2 Data Manipulation . . . 43
4.3.3 Access Methods . . . 44
4.4 Summary . . . 46

5 Query Processing and Optimization **47**
5.1 Design Issues . . . 47
5.1.1 OODBMS Query Optimization Issues . . . 47
5.1.2 Spatial Query Optimization Issues . . . 47
5.2 The Bodhi Language Interface . . . 48
5.2.1 Data Definition Language . . . 48
5.2.2 Query Language . . . 48
5.3 Query Processing and Optimizations in Lambda-DB . . . 50
5.3.1 Introduction . . . 51
5.3.2 The Monoid Comprehension Calculus . . . 51
5.3.3 The Monoid Algebra . . . 51
5.3.4 OPTL . . . 53
5.3.5 Phases of Lambda-DB Optimizer . . . 54
5.3.6 Parsing And TypeChecking . . . 54
5.3.7 Normalization . . . 54
5.3.8 Translation into an algebraic form and Unnesting . . . 55
5.3.9 Materialization of Path Expressions . . . 55
5.3.10 Operator Ordering . . . 56
5.3.11 Physical Plan Generation . . . 56
5.4 Query Processing . . . 57
5.4.1 Data Flow in Bodhi . . . 57
5.4.2 Schema Manager . . . 58
5.4.3 Interface With Visualization . . . 58
5.5 Query Optimizations in Bodhi . . . 61
5.5.1 Path Expression . . . 61
5.5.2 Derived Attributes . . . 66
5.6 Extensions . . . 67
5.7 Summary . . . 68

6 XML Visualization Interface **69**
6.1 Introduction . . . 69
6.2 Visualization Architecture . . . 69
6.3 System Requirements . . . 70
6.3.1 Modeling requirements . . . 70
6.3.2 Typical queries . . . 70
6.4 System Design and Modeling . . . 71
6.4.1 Library Package . . . 71
6.4.2 Network Interconnection . . . 72
6.4.3 The Native Interface . . . 72
6.4.4 Displaying Query Results . . . 72
6.5 Implementation Details . . . 73
6.5.1 Data Transaction between the GUI and the Database server . . . 73
6.5.2 Query Results Exchange . . . 75
6.6 Related Work . . . 75
6.7 Summary . . . 76

7 System Integration, Testing and Evaluation **79**
7.1 Motivation . . . 80
7.2 Related Works . . . 80
7.3 Issues in Integration of Spatial Module . . . 81
7.3.1 Extension of Schema Definition Language . . . 81
7.3.2 Extension of Query Language . . . 82
7.3.3 Problems in underlying *Shore* . . . 82
7.4 Benchmark Description and System Evaluation . . . 83
7.4.1 Benchmark Data . . . 83
7.4.2 Query Descriptions . . . 83
7.4.3 System Evaluation Result . . . 84
7.5 Summary . . . 85

Bibliography **85**

A The Rumbaugh Notation **91**

B Taxonomy Data Model **92**

C Taxonomy Data Model in ODL **96**

D ODL Declarations of the Spatial Data Types **101**

E Spatial Queries **103**
E.1 Inside Queries . . . 103
E.1.1 IsPointOnLine . . . 103
E.1.2 IsPointInsidePolygon . . . 103
E.1.3 IsLineInsidePolygon . . . 103
E.1.4 IsPolygonInsidePolygon . . . 103
E.2 Overlapping Queries . . . 103
E.2.1 IsLineInstersectingLine . . . 103
E.2.2 IsLineIntersectingPolygon . . . 103
E.2.3 IsPolygonIntersectingPolygon . . . 104
E.3 Adjacency Queries . . . 104
E.3.1 IsLineMeetingLine . . . 104
E.3.2 IsLineMeetingPolygon . . . 104
E.3.3 IsPolygonAdjacentToPolygon . . . 104

F DDL of Oshadhi in BNF **105**

G Query Language of Oshadhi in BNF **107**

H Meta-Data Export format in BNF **109**

I Remote Method Invocation **110**
I.1 Architectural Overview . . . 110
I.2 How RMI works . . . 111
I.2.1 Develop Remote Object Code . . . 111
I.2.2 Develop the Server Code . . . 112
I.2.3 Setting up a security manager . . . 113
I.2.4 Develop the Client Code . . . 113
I.2.5 Compile the code . . . 114
I.2.6 Start the Registry . . . 114
I.2.7 Start the Server . . . 114

J Java Native Interface **115**
J.1 Where JNI might be used . . . 115
J.2 Steps to write a JNI application . . . 116

Chapter 1

Overview of Project

Over the last decade, there has been a phenomenal growth in the area of *plant bio-diversity* studies, largely motivated by the economic and humanitarian potential that underlies the understanding of plant dynamics – in fact, a new area called "bio-prospecting" has arisen, whose focus is solely on sifting through bio-diversity data to locate potentially profitable biological sources.

A major hurdle faced by bio-diversity scientists is the effective management and access of the large amounts and varied types of data that arise in their studies, ranging from micro-level biological information (such as genetic makeup of organisms and plants) to macro-level information including ecological and habitat characteristics of species. Six years ago, we made a first attempt to address the above problem in the *Oshadhi* [VH95] project, which developed an object-based database system for efficiently handling plant taxonomies. However, it had only a rudimentary spatial-data component and a grossly simplified query processor/optimizer. Further, during the interim period, there have been a variety of new technologies that have arisen both on the biological and on the database fronts. These include the Internet/Web, XML, sophisticated spatial indexing schemes, genome sequencing algorithms, and semistructured data-handling techniques, all of which can be usefully incorporated in biodiversity information systems.

In light of the above, over the last few years, with financial support from the Department of Bio-technology, Government of India, we have been developing the next generation biodiversity information system, called **BODHI** (Biodiversity Object Database arcHItecture).[1] BODHI is an object-oriented database system built around the SHORE kernel [CDF$^+$94] which incorporates the latest algorithms and data handling structures, provides integration with the Internet for data dissemination, and handles the needs of bio-diversity researchers with diverse focus areas. In this report, we describe our experiences in the design and implementation of the BODHI prototype.

1.1 Design Goals

In this section, we highlight the main features that would be desirable in a bio-diversity information system.

Handling of Complex Data Types: Plant bio-diversity data can be broadly classified into the following three categories: (1) **Taxonomy Data.** This is data about the relationships between species based on their characteristics. In turn, this consists of *phenetic relationships* (based on comparison of physical characteristics or *phenotype*) and *phylogenetic relationships* (based on evolutionary theory)[Pan91]. The various characteristics on which these relationships depend on may vary in time due to discovery of a new class of characteristics, corrections to previously recorded characteristics, etc.; (2) **Geo-spatial Data.** The study of ecology of species involves recording the geographical and geological features of their habitats, water-bodies, artificial structures like highways which might affect the ecology, etc. These are represented on a map of the region and have to be handled as spatial data by the database; (3) **Bio-molecular Data.** The genetic makeup of species is becoming increasingly important with a large number of genome sequencing projects working on organisms and plants. Bio-prospectors look for indigenous sources of medicines, pesticides and other useful extracts. Such data can be discovered from the biomolecular and genetic composition of species.

These datatypes have complex and deeply-nested relationships within and between themselves. Further, they may involve sophisticated structures such as sequences and sets.

Molecular Pattern Discovery: The molecules that are of interest in bio-diversity are DNA and Proteins. DNA is represented as a long sequence based on a four nucleotide alphabet. There are regions in DNA sequence, called *exons*, which contain genetic code for the synthesis of Proteins. The proteins are long chains of 20 amino acids. Each protein is characterized by the amino acid patterns it has, and is responsible for various functionalities in a cell which determine the characteristics of the organism or plant.

[1] Gautama Buddha gained enlightenment under the Bodhi tree.

The similarity between two genetic sequences is a measure of their functional similarity. Analysis of DNA and Protein sequences from different sources gives important clues about the structure and function of Proteins, evolutionary relationships between organisms, and helps in discovering drug targets.

There are a number of algorithms for performing the similarity search over genetic sequences. Researchers and bio-prospectors frequently search the database using these algorithms to locate gene sequences of interest. However, the implementation of these algorithms is typically external to the database, making them relatively slow. It therefore appears attractive to consider the possibility of integrating these algorithms in the database engine. [2]

Internet Access: As with all other scientific communities, the bio-diversity community relies on timely knowledge dissemination. Therefore, supporting access through the Internet is vital for maximizing the utility of the information stored in the database.

XML Compliant: Typically, bio-diversity data is collected and managed by individual research institutions and commercial enterprises autonomously. In order to provide larger scope of data availability, it is necessary that such localized and autonomous data repositories be able to exchange data. The current state of information exchange amongst various bio-diversity data repositories is not very satisfactory[Saa99]. However, with the advent of XML, many research groups are proposing DTDs in individual fields of ecology and genetics[ANZ, Bio] . The bio-diversity information system should support these DTDs for handling data over heterogenous set of repositories.

Visual Interface: It is imperative to have a good visualization interface for the results produced by the system since (a) the end-users are biologists, and (b) the results could range from simple text to multidimensional spatial objects. The database system must support easy integration of a visual interface.

Low Cost: Most of the research in bio-diversity, at least in India, is done by small teams of researchers who work within low budgets and are unable to afford high-cost data repository systems. Therefore, solutions that are completely or largely based on public-domain freeware are essential for these groups.

1.2 Design Description

In this section, we present the highlights of the design of the BODHI system intended to efficiently achieve the goals described in the previous section. The overall architecture is pictorially shown in Figure 1.1. The back-end micro-kernel is the SHORE [CDF+94] object-based storage manager, which provides the basic database features such as device and storage management, transaction processing, logging and recovery management. Above this are three modules, *Object Services*, *Spatial Services* and *Sequence Services*, which provide the core functionalities of the BODHI system – these modules are described in detail in the remainder of this section. The three services are integrated through the *Query Processor* which interacts with the service modules and the SHORE server to optimize and process the queries on the database. Clients submit queries, over the Internet, in the form of OQL [Cat94] command strings – the *Client Interface Framework* receives these queries and returns results in the formats requested by the clients.

1.2.1 Object Services

In BODHI, the object oriented paradigm is used to achieve the following features necessary for bio-diversity data:

- Support multiple data primitives through the use of type libraries at the database layer. (The spatial and genome sequence data primitives are provided using this facility.)
- Representation of complex relationships such as sets, sequences and bags.
- Build new types through inheritance and aggregation of previously defined non-primitive types.

The data modeling language of BODHI extends the standard ODL[Cat94] by introducing new primitives for modeling spatial and sequence data. These primitives can be used in conjunction with standard primitives provided by ODL and various relationships between objects can be easily modeled. The schema definition supports two types of relationship-paths over objects, namely, Inheritance hierarchies and Object Relationship paths. The queries over the database consist of both value based queries and also on the context of the object in the relationship graph and the position of its type in the associated class hierarchy.

[2]This observation is gaining currency in the commercial database arena as well, as exemplified by IBM's provision of homology searching as UDFs in DB2.

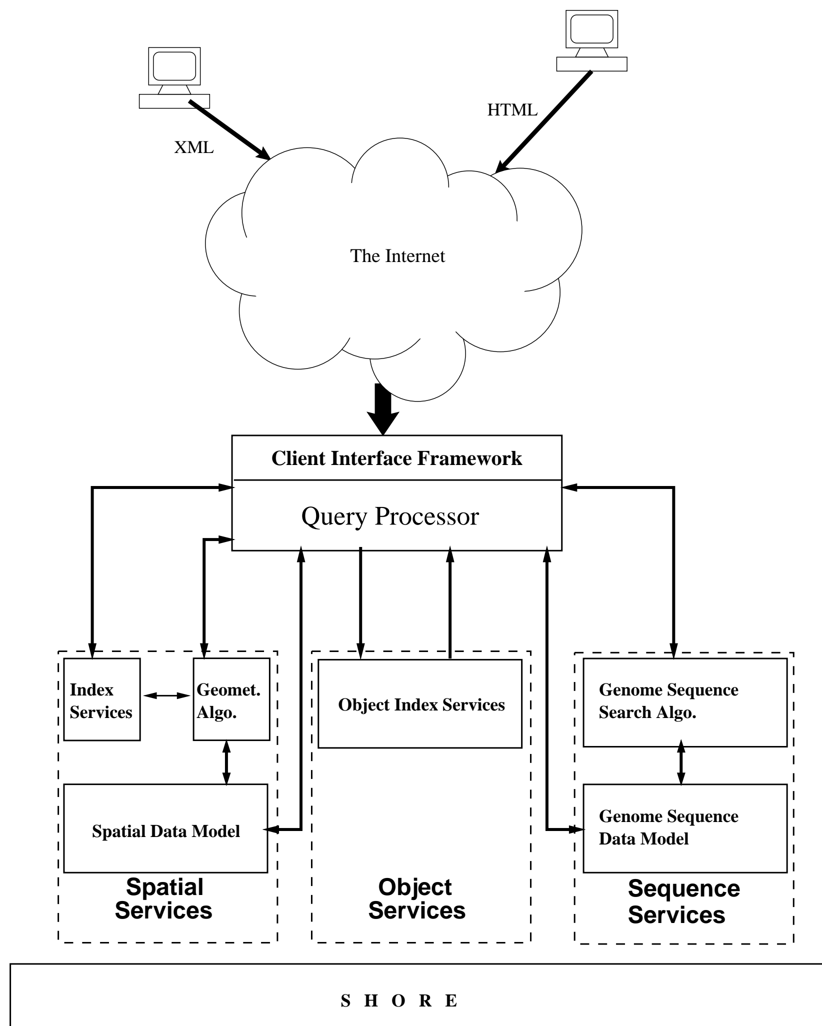


Figure 1.1: Schematic of Architecture of BODHI

Indexing the Inheritance Hierarchies

Based on the context, the scope of queries over inheritance hierarchies in the bio-diversity domain can be either limited to the immediate objects of the predicate class (i.e. single-class query) or can be extended to include all objects of the sub-hierarchy rooted at the predicate class (i.e. class-hierarchy query). For example, with reference to the schema of Figure 1.2, which is a (partial) class diagram of the plant taxonomy model,

Give names of all **PlantSpecies** *associated with a* **GeoRegion**

We have two possible semantics for this query, (i) search in the complete inheritance hierarchy rooted at *PlantSpecies* and return objects of type *PlantSpecies* and *MedicinalPlants* associated with the given *GeoRegion*, (ii) search objects of only *PlantSpecies* type associated with given *GeoRegion*, without searching for *MedicinalPlants*.

To efficiently support both types of queries, the *Multikey Type Index*(MT-index)[MP97a] approach is used in BODHI. The basic idea behind MT-index is a mapping algorithm, called *Linearization Algorithm*, which maps type hierarchies to linearly ordered attribute domains in such a way that each sub-hierarchy is represented by an interval of this domain. Using this algorithm, MT-index incorporates the type hierarchy structure into a standard multi-attribute search structure, with the hierarchy mapped onto one of the attribute domains (type domain). The result is an index on $K+1$ keys, K indexed object attributes and one additional key representing the type. This scheme not only supports single-class and class-hierarchy queries, but also makes answering multi-attribute queries straight forward. Further, the implementation of MT-index can use any of the multi-dimensional indexing schemes (like R^*-trees, supported by SHORE), which simplifies the implementation overhead.

1. Using a combination of two hierarchy indexing approaches, MT-index can answer both single and class hierarchy queries with equal efficiency.

2. By extending the number of dimensions, queries over multiple attributes can be easily answered without any

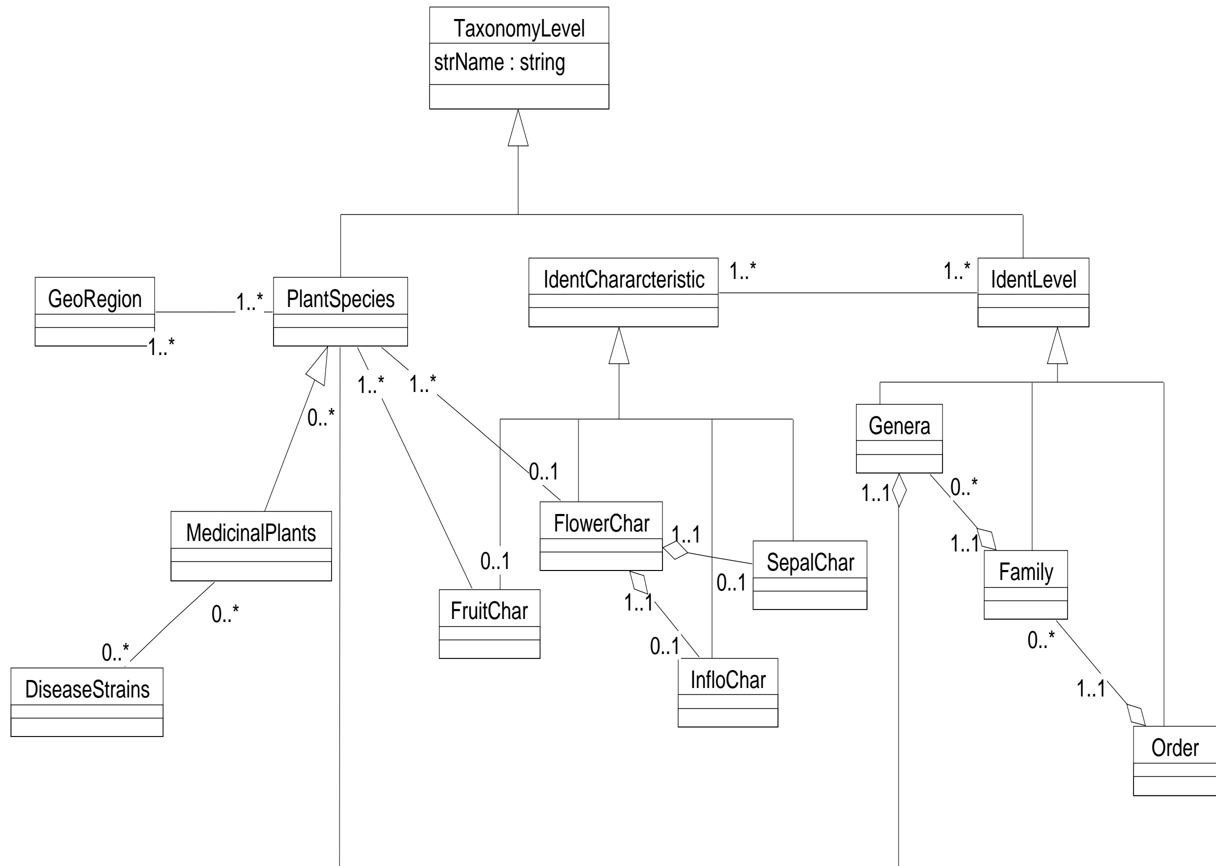


Figure 1.2: Partial Schema of Plant Taxonomy Database.

time overhead.

Indexing the Relationship Paths

Referring again to the schema of Fig 1.2, we can see that the relationship hierarchy between objects in bio-diversity database schema can be arbitrarily deep. Further, it is possible to have recursive relationships, for example, the *Predator-Prey* relationship among Species. Queries over such relationship graphs can have either the ancestor class or the nested class, as the predicate class. To illustrate, consider the following pair of queries:

- *Identify the* **PlantSpecies** *based on one or more of its* **IdentCharacteristics**.
  Here, we need to scan for object relationship path(s) culminating at the specified characteristic, and then locate the species that form the root(s) of that path(s). Such queries are called TP (Target-Predicate) queries, following the terminology of [LL98a].

- *Retrieve all* **IdentCharacteristics** *of a given* **PlantSpecies**.
  Here, the predicate classes are ancestor classes (PlantSpecies) and the target classes are nested classes (IdentCharacteristics). These types of queries are called PT (Predicate-Target) queries in [LL98a].

To efficiently handle both PT and TP queries, BODHI implements the *Path Dictionary Index*(PD-Index)[LL98a] approach. The PD-Index consists of three parts: the *path dictionary* which supports the efficient traversal of the path, and the *identity index* and *attribute index* which support associative search. The identity index and attribute index are built on top of the path dictionary.

Conceptually, the path dictionary extracts the compound objects, without the primitive attributes, to represent the connections between these objects. Since primitive attribute values are not stored in the path dictionary, it is much faster to traverse the nodes. In order to support associative search based on attribute values, PD-Index provides attribute indexes which can be built for each attribute frequently queried on. When OID of the object is given, the path information is obtained using the identity index built over the path dictionary.

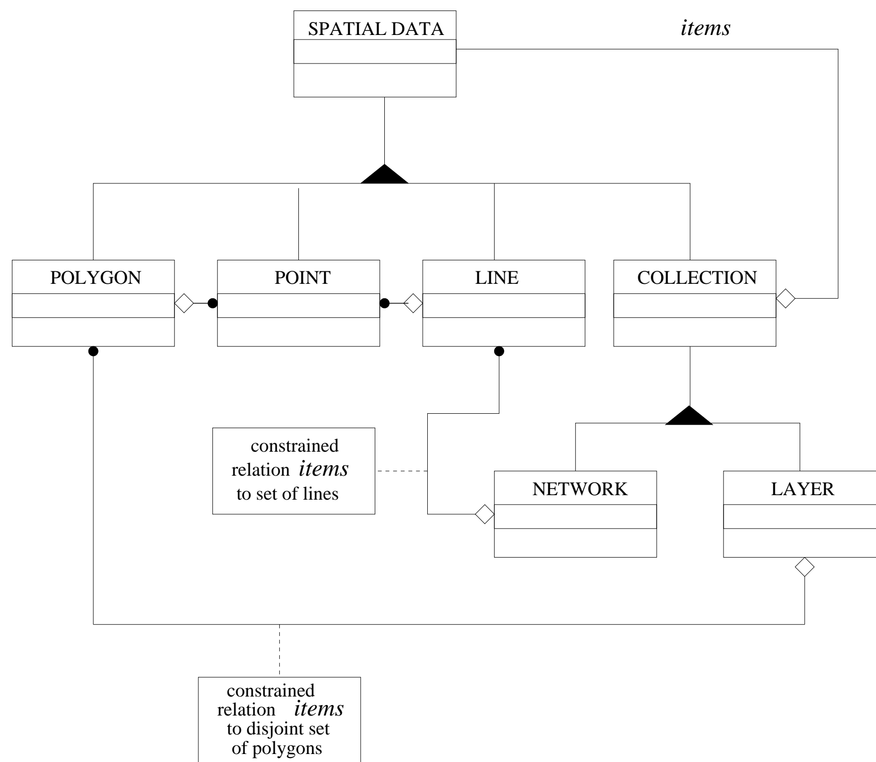


Figure 1.3: Class diagram of Spatial Data model in BODHI

The PD-index provides the following advantages:

- Since both forward and backward traversals are supported by the path dictionary, it is possible to evaluate both PT and TP queries with equal efficiency.
- Since intermediate objects need not be traversed in deep aggregations, lot of savings in retrieval time can be achieved.
- Identity index reduces the update overhead by directly pointing the path dictionary locations where update is to be done.

A limitation of the PD-index, as per its implementation description in [LL98a], is that it only handles 1:N relationships or 1:1 relationships. However, a typical schema of bio-diversity database consists of aggregations of N:M cardinality, and structures like sets, bags and sequences in the aggregation path. In order to handle these complex relationships, we have extended the implementation of the PD-Index – the details of our enhancements are in Section 1.3.

1.2.2 Spatial Services

Spatial data or geographic data forms a key component of a bio-diversity data repository. The spatial data is frequently queried upon, and a spatial query could involve operations (geometric and topological) which could be extremely costly to evaluate. And more importantly, a large size of macro-level biodiversity information consists of topographic and ecological maps.

In BODHI, we provide *Spatial Data Types* (SDTs) in our data model and query languages (ODL/OQL extended for the purpose), and support efficient spatial indexing and spatial join algorithms.

Spatial Data Types

BODHI provides a set of spatial data primitives to represent single spatial objects like country, state, forest, river etc. as well as to represent spatially interrelated collection of objects such as *"Political map of India"*, which can be modeled as a topologically related collection of polygons representing states.

The standard ODL modeling language has been extended with new keywords to enable users to include these primitives in their schema descriptions.

The spatial model of BODHI is based on the *Realm/ROSE Algebra*[Güt94b], and provides two categories of primitives: *Simple Primitives* and *Compound Primitives*. The simple primitives enable modeling of single objects in

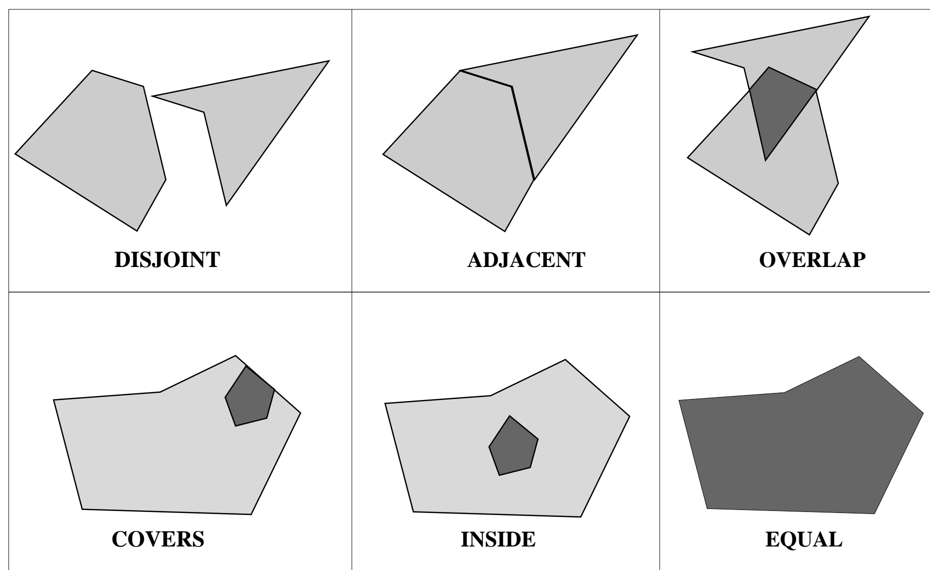


Figure 1.4: Spatial Relationships in BODHI

space, and includes types for *Point*, *Polyline* and *Polygon*. The compound primitives, on the other hand, are used to model spatially-related collection of objects. There are two compound primitives: *Layer* and *Network*, for modeling collections of *Polygon* and *PolyLine*, respectively.

The fig. 1.3 gives the class diagram of the Spatial data model of BODHI. As illustrated, the Point forms the basic information entity of the spatial data. Every Polyline is modeled within the type library, using the information about the end points of each segment that forms the Polyline. The Polygons are the collection of single-segment lines.

Querying on Spatial Objects

Spatial queries consist of selecting objects which satisfy some spatial relationship(s). There are three classes of such spatial relationships,

- *Topological relationships*, such as *adjacent*, *inside* etc. which are invariant under geometrical transformations like translation, scaling and rotation.
- *Direction relationships*, like *above*, *north-of* etc.
- *Metric relationships*, based on the distance measure between spatial objects. E.g., *"distance < 100"* etc.

Of all the relationships in these three categories, [Güt94b] observes that only six relationships, *disjoint*, *in*, *touch*, *equal*, *cover* and *overlap*, which are illustrated in Figure 1.4, are most important relationships in geo-spatial applications (even though only polygons are used for illustrating the relationships, these are also defined for line, layer and network primitives). The *Geometric Algorithms* part of the Spatial Services of BODHI currently provides these six relationships. These algorithms form the core of behavioural abstraction of the spatial primitives described above.

Spatial Indexing

Special indexing structures are required for efficiently accessing spatial objects. While numerous indices have been proposed in the literature (see [GG98] for a survey), R^*-Trees are extremely popular spatial indexing structures, as they have been designed to achieve better packing of nodes and fewer disk accesses in comparison with many other variants of R-Trees.

A fundamental difficulty in designing of multidimensional index structures stems from the following reason: There is no total ordering that preserves spatial proximity. One way out of this is to find heuristic solutions, that is, to look for total orders preserving spatial proximity to a large extent. The requirement is that if two objects are located close together in multidimensional space, there should be a high probability that they are close together in the total order. One such heuristic is the concept of *space-filling curves*[Jag90a] and there have been proposals based on this heuristic (for a survey of these and other spatial access methods, refer to [GG98]). All the proposed index structures have the following in common: They first partition the multidimensional space with a grid, and then use a curve such as *Peano Curve* or *Hilbert Curve*[FR89a], which visits all the points in the grid only once without crossing itself, to obtain a total ordering of the spatial objects.

BODHI indexes spatial objects using one such proposal, the *Hilbert R-Tree* (HR-Tree) [KF94a], where total ordering is achieved through Hilbert Curve for total ordering. The linear ordering imposed by the space filling curve

is used to optimize the structure of the R-Tree. The performance gains have been shown to be more than 25% over R^*-Tree, considering a wide range of workloads[KF94a]. We expect that similar performance gains might be obtained in BODHI as the benchmarks used in [KF94a] closely match the real-life needs.

Spatial Join Strategies

Spatial joins, wherein sets of objects are compared using predicates on their spatial attribute values, are common in bio-diversity applications. For example,
"select all the states in India where the rivers Godavari, Krishna and Kaveri flow".
Obviously, any cartesian product based solution is impractical for spatial joins since they are extremely expensive to evaluate.

However, while spatial index structures can be used for computing the join efficiently, a refinement of the result is required since the index structures approximate each object's contour to its bounding box. So the key ideas for computing spatial joins are the filter and refinement strategy and the use of spatial index structures. If the filter step is based on the bounding boxes, the problem is to determine for two sets of rectangles R, S, all pairs (r,s), r∈R, s∈S, such that r intersects s. There are different policies for join processing depending on the availability of the index on the operands.

- If one operand is represented in a spatial index, then an *index join* or *repeated search join* can be used [LR94]. This is a classical technique usually used with a B^+-tree index, which can be applied equally well to spatial index structures. Hence, if the "inner" operand is represented in an spatial index supporting rectangle intersection queries one can scan the "outer" operand set; for each object, the bounding box of its SDT attribute is used as a search argument of the index. As a result one again obtains a set of candidate pairs with intersecting rectangles. Repeated search join is especially efficient if the outer set is not too big (e.g., it is the result of a selection from a large set). If both are large, *bb_join* may be more efficient [Güt94b] and query processor makes the appropriate choice.

- If both the operands have spatial indexes built, the basic idea is to perform a synchronized traversal of the two index structures so that pairs of cells whose respective partitions cover the same part of space are encountered together. The parallel traversal of R-trees method discussed in [BKS93] is appropriate in BODHI, because it uses any of the R-tree family of index structures and hence can achieve all the benefits of the Hilbert R-tree index structure for join processing also.

- If none of the operands have index, the index of one operand can also be built on the fly, as suggested by [LR94]. This is considered by the query processor before the next option.

- If none of the operands is represented in a spatial index, a good technique suggested by [Güt94b] is to use a rectangle intersection algorithm from computational geometry called *bb_join*. This computes the join using bounding box information of spatial objects rather than the actual extent, making it faster than cartesian product. Later, we perform the filtering over the join result.

The query processor of BODHI chooses the appropriate policy of spatial join computation, from among those listed above, depending on the availability of index over the attributes under query.

1.2.3 Sequence Services

As we already highlighted in Section 1.1, the bio-molecular information of various species is growing at a very fast rate. The study of genetic makeup of species in conjunction with macro-level information is becoming an important activity in bio-diversity studies. BODHI facilitates for such integrated study by providing data model and query language support for modeling of biological sequences, and implementing commonly used sequence similarity algorithms over the genome sequence data.

Biological Sequence Primitives

In biology, a bio-molecular sequence is either a *DNA* (Deoxyribonucleic acid) or a *Protein* sequence. These two sequences together form the genetic makeup of all species.

A protein is a chain of simple molecules called *amino acids* and is responsible for various functions and properties that characterize an organism. In nature, normally we find 20 different amino acids in a protein, but there are few cases when nonstandard amino acids are seen in a protein[SM97]. These proteins are assembled using the information encoded in associated DNA molecules. A DNA molecule is also composed of simpler molecules, called nucleotides, which are exactly 4 in number. Though the structure of DNA molecule is a twisted helix, the sequence of only one of the strands forming this helix is sufficient to describe the DNA (the other strand would be a *complement* strand, which can be determined from the original). Thus, both protein and DNA molecules are represented as strings of letters, with 20 alphabets for protein and 4 alphabets for DNA molecules.

The data model provides two classes, *DNAStr* and *ProteinStr*, to represent the sequence information of segment of a DNA molecule or a Protein molecule respectively.The storage representation of DNAStr and ProteinStr makes use of the small alphabet set, and encodes the DNA sequence using two bits and Protein sequence using 5 bits.

The behavioral part of this data model, provides functions for translation of DNA sequence into a Protein sequence[3] and vice-versa, complementary DNA strand generation and substring operations.

Similarity Search

Sequence comparison is the most important operation in computational biology, and it forms the basis for many other complex manipulations. The similarity searching over DNA and Protein sequences gives important clues about the genetic and phylogenetic relationships between species.

A fundamental concept in measuring similarity between two sequences is that of *alignment score* between two sequences or their subsequences. The alignment between two sequences is defined as the insertion of spaces in arbitrary locations along the sequences so that they end up with the same size, subject to the constraint that no space in one sequence be aligned with a space in the other. The scores for all possible combination over alphabets (including space) are provided as cost matrices, and the score of an alignment is computed as the sum of scores of individual alphabet alignments in it.

There are many sequence similarity search algorithms based on various techniques [Gus97, SM97], but the most popular are *BLAST* [AGM+90] and *FastA* [LP85]. Both are heuristic based algorithms, based on the notion of *Local Alignment* [SW81], using the alignment scores of segments of sequences, as a measure of similarity between two sequences involved in the search.

Both these algorithms involve a complete scan of the sequence database for every query sequence. The index structures designed for nearest neighborhood searching in metric spaces cannot be used, as the similarity measures do not form a metric space since the scoring matrices contain negative costs [SM97]. Efficiency can be clearly improved by supporting these algorithms within the database rather than at the application layer.

BODHI provides these two algorithms, over DNA as well as protein sequences, and they can be part of an OQL query. The similarity is computed using a set of default cost matrices, (i) PAM-120 [DSO78] matrices for proteins, and (ii) for DNA identities are scored with +5, mismatches as -4 following the scheme in [AGM+90].

1.3 Implementation Details

As previously described in Section 1.2, the service modules of the BODHI, provide indexing methods, over object hierarchies and spatial data, and sequence similarity algorithms. In this section, we focus on the implementation details of these service modules with emphasis on index structures.

Shore supports extensions to the standard set of features it supports in the form of Value Added Servers(VAS). The storage manager provides a programming interface, called Storage Manager Programming Interface [CDF+94], for writing these extensions. The service modules in BODHI, are implemented at this layer.

1.3.1 Spatial and Sequence primitives

The modeling primitives for spatial and sequence data are introduced as declarations in SDL(SHORE Definition Language), data definition language provided by SHORE. The implementations of the operations on them are provided in separate C++ files, which are compiled and maintained separately as libraries. The schema manager and query processor of BODHI, have been programmed to consider these types as primitive datatypes.

1.3.2 Indexes over Objects

The index structures over class inheritance hierarchies and object relationship paths are part of the Object Services VAS. The interface with query processor consists of methods to create and destroy indexes, to retrieve and to scan the indexes. These methods are called by the query processor using RPCs which are evaluated by the VAS and the results are returned to query processor.

Path Dictionary

The implementation of PD-index for object relationship graph, extends the the s-expression based implementation suggested in [LL98a] to handle the presence of N:M relationships between objects and compound aggregations such as bags, sets and sequences. The original implementation encodes all paths terminating at the same object into an *s-expression.* The s-expression for the path $C_1.C_2.C_3...C_n$ is defined as follows:

- $S_n = \theta_n$, where θ_n is the OID of an object in class C_n or null.

[3]Three nucleotides in DNA sequence determine the amino acid in the Protein sequence encoded by the DNA. And for every DNA sequence, there are three possible reading frames giving distinct Protein sequence, and all reading frames are simultaneously translated during this operation.

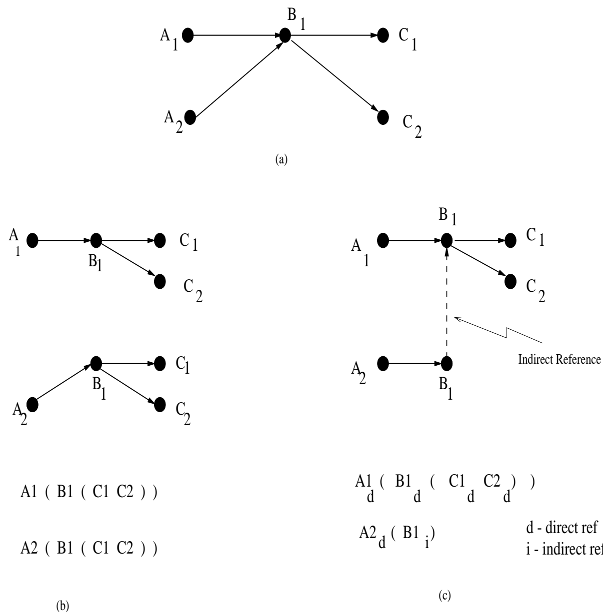


Figure 1.5: Representing N:M relationships (a) N:M relationship (b) Equivalent 1:N relationships with replicated paths (c) Equivalent 1:N relationships with indirect references.

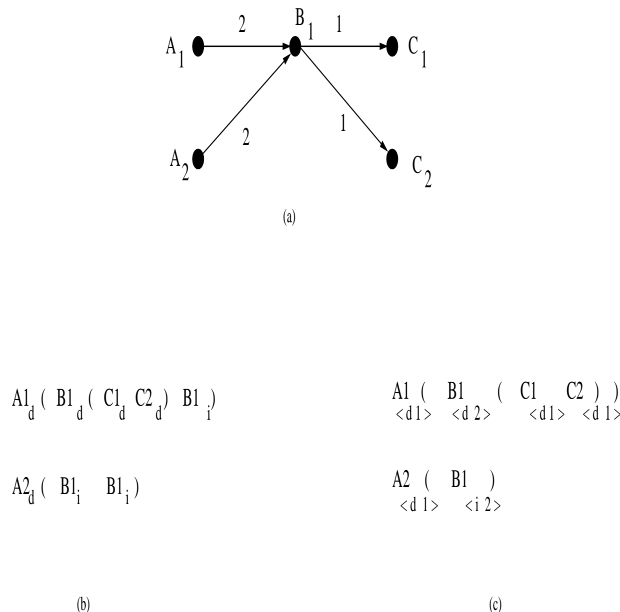


Figure 1.6: Representing N:M relationships in presence of (a) Bags (b) Equivalent 1:N relationships with indirect references (c) Equivalent 1:N relationships with indirect references as well as replication counts

- $S_i = \theta_i(S_{i+1}[, S_{i+1}])$ $1 \leq i < n$, where θ_i is the OID of an object in class C_i or null, and S_{i+1} is an s-expression for the path $C_{i+1} \ldots C_n$.

S_i is an s-expression at i^{th} level in which the list associated with θ_i contains recursively the OIDs of all descendant objects of θ_i. Except for the objects in C_n, every object on the path has a descendant list, which may be empty.

Due to the inherent tree structure, the s-expression scheme supports 1:1 and 1:N relationships naturally. But in models representing the bio-diversity data, the possibility of N:M relationships cannot be overlooked. To illustrate this, consider the relationship between the PlantSpecies and their associated GeoRegions, in the schema of Figure 1.2. Note that a PlantSpecies can be found in any number of GeoRegions and a single GeoRegion may harbor many PlantSpecies. This forms a natural N:M relationship between species and their geographical location.

Moreover, structures like *Bag* and *Sequence* are common in biological data models. Each of these relationships need different treatment at the implementation level to preserve the correctness and to reduce redundancy without compromising on retrieval time.

We have extended the implementation given in [LL98a] to support these additional requirements. The main idea behind our extensions is to break the of N:M relationship into multiple 1:N relationships. But a straightforward application of this idea introduces complications in maintenance of s-expressions.

Supporting N:M relationships: Consider the representative N:M relationship graph shown in Figure 1.5(a). If we break this into multiple 1:N relationships, the graphs and the corresponding s-expressions look as in Figure 1.5(b). Note the redundancy in these s-expressions: The children of B_1 are replicated in both of the s-expressions of A_1 and A_2. This problem can be solved by using a flag in the entries of s-expression. The flag denotes whether the entry is a direct reference or an indirect reference. All the descendant entries of an OID will be stored only in the entry which contains direct reference to that OID. This modification is shown in form of a graph in Figure 1.5(c) with their corresponding s-expressions. Note that the suffix for each entry denotes whether it is a direct reference or an indirect reference. Though this modification, duplicates (with different flag values) the B_1 entry, we avoid duplicating the children of B_1, thus saving space.

Extensions to support Bags and Sequences: The above modification works fine for storing ordinary references and sets. But in presence of bags, further redundancy is possible. The example for this is shown in Figure 1.6. The number on edge from a to b denotes the number of times b appeared as a reference in the bag of a. The corresponding s-expressions for this graph using above implementation are given in Figure 1.6(b). Note that the entry of B_1 is repeated n times in each expression, where n denotes the number of times B_1 is referenced in parent object. This replication can be eliminated by introducing one more field in the entry of s-expression which stores this replication count. This reduces the storage overhead for storing bags since OIDs are not duplicated. The s-expressions with this modification are shown in Figure 1.6(c).

The implementation also supports sequences by maintaining the order of the children of a given parent in the s-expressions.

MT-Index

The MT-Index over type hierarchy, turns out to be a multi-dimensional index with type as a dimension, after the linearization of the inheritance hierarchy as proposed in [MP97a]. SHORE storage manager already provides a multi-dimensional index, R^*-Tree. Hence, the MT-Index has been implemented using R^*-Trees with an additional dimension to represent the type of the object.

1.3.3 Spatial Services

Aiming to provide an efficient implementation of Spatial Services, we were faced with an operational problem. The Shore SM interface, the interface to extend the features provided by the storage manager, allows one to introduce new logical index structures over data but no page-level storage control is provided to index structures. This excludes the possibility of implmenting index structures, such as Hilbert R-Tree, that rely on physical clustering of data for performance reasons.

This forced us to implement the Hilbert R-tree by refactoring the existing code for R^*-Trees within Shore storage manager and exporting its interface to both VAS layer (through SM interface) and to Application layer (through SDL interface).

1.3.4 Sequence Services

As already discussed, the genome sequence support in BODHI, includes a pair of sequence similarity algorithms (BLAST and FastA) in addition to the data primitives to store and manipulate finished DNA and Protein sequences.

The Sequence services module provides an efficient storage representation of DNAStr and ProteinStr datatypes by encoding them using 2 bits and 5 bits per alphabet respectively (though further compression could be possible on these strings, it will significantly slowdown the similarity searching). Each sequence is stored in the form of a record of a file in the storage manager. This scheme of storage allows us to perform complete scans of the sequences in an efficient manner. In addition, we make use of the fine-granularity locking provided by the SHORE storage manager, to search even when a record is being updated and to perform multi-threaded searches.

The current implementation of BLAST and FastA algorithms, provide for simultaneous execution of two similarity searches over proteins and over DNA sequences.

1.4 Query Processing

The query processor of BODHI integrates the features provided by the service modules through extended ODL/OQL for modeling and querying the database. In addition, it optimizes the queries using the metadata and index information, and as part of the *Client Interface Framework*(discussed in Section 1.5) it facilitates transformation of query results into an interchangeable format as requested by the user.

The core of the Query Processor module, is *lambda-DB*, a freely available, rule-based query processor and optimizer for ODL/OQL. Lambda-DB, performs all optimizations on the query at the compile-time, producing an corresponding executable, resulting in extremely fast query executions.

The schematic representing the flow of schema definitions and queries over the database, is illustrated in Figure 1.7.

1.4.1 Schema Definition

The schema is defined, as already mentioned, using ODL – with extensions necessary for BODHI. The schema declarations are first converted into SDL, before getting compiled into an C++ header file. During this phase, the *Schema Manager* of the query processor obtains the metadata needed for typechecking and optimization of queries and maintains it in the database. The implementation part of the schema declaration, is abstracted out into a C++ code and is available for compilation into a linkable library.

1.4.2 Query Flow

The BODHI consists of a web-enabled front end for querying with the query processor forming the back-end. The interface between two is provided through a *JNI*(Java Native Interface) layer, which receives the queries from clients through *RMI*(Remote Method Invocation).

The query strings obtained are type checked, parsed and compiled into C++ code with execution plans generated after incorporating the rules specified in the query optimizer. The type library of spatial and sequence data primitives and the implementations of various operations defined over them, which are precompiled into linkable libraries and header files, are linked to generate the executable of the query.

This executable contains implementations for interacting with the SHORE storage manager, Value Added Servers of Object and Sequence Services, and the implementation needed for transforming query results into interchangeable format. The transformed results are returned back, again through JNI layer, to the client.

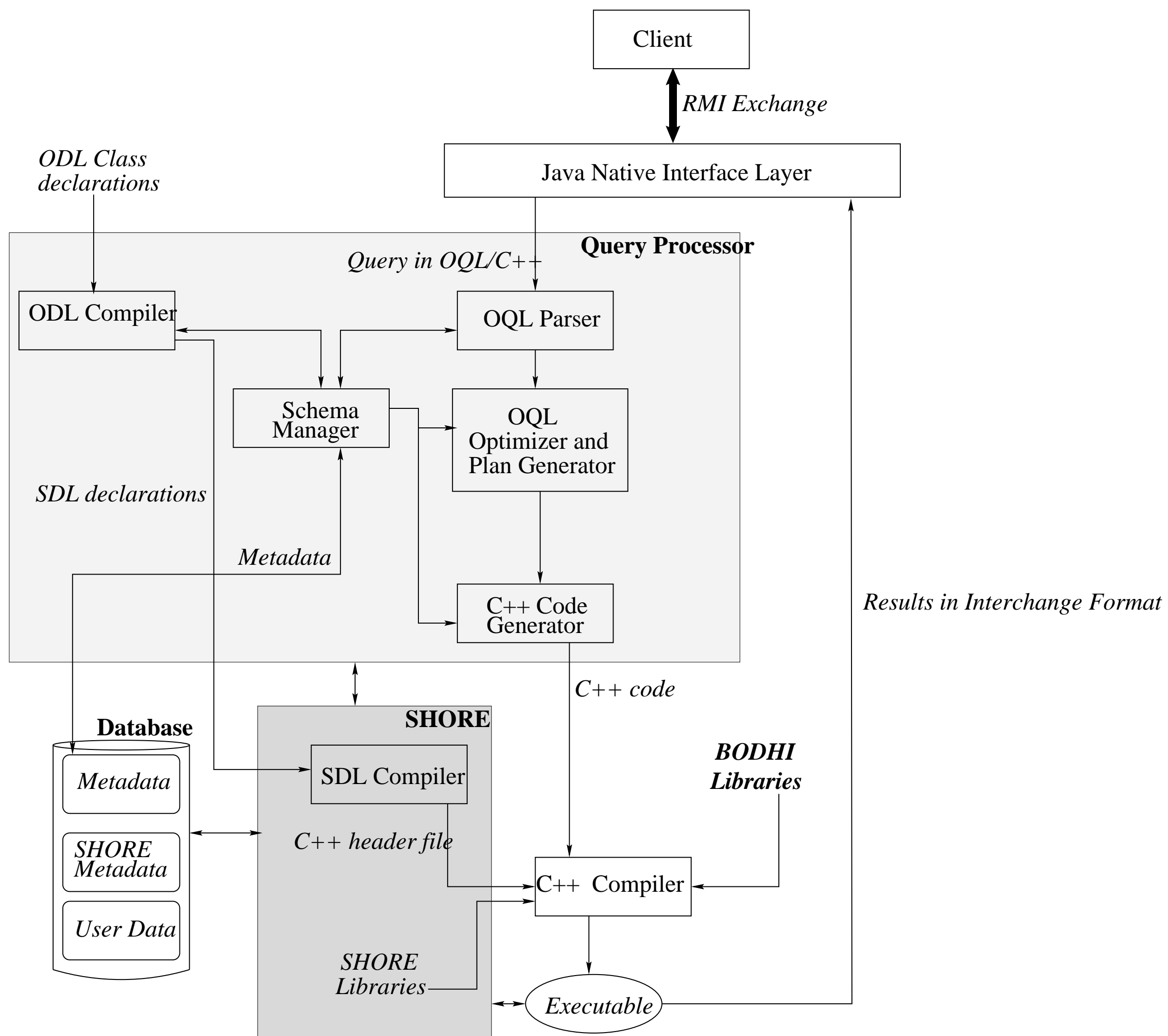


Figure 1.7: Schema Definition and Query Flow in BODHI

Strategies for Query Optimization

The lambda-DB, provides a language, OPTL, which can be used to specify various optimization rules. The default ruleset that ships with lambda-DB, supports the standard optimizations including making use of B^+-Tree and R^*-Tree indexes in Shore. This ruleset has been enhanced to make use of specialized indexing schemes in BODHI.

Path Expression Computation As we already described in Section 1.2, due to the presence of object relationship paths, queries can be issued on any object deep in the relationship paths. So a query might involve computation of the predicate $C_1.a_1.a_2.....a_m = C_2.b_1.b_2......b_n$. This can be easily observed to be a join of the extents of the classes that form the path expression. So, one of the optimization schemes followed in lambda-DB, is to convert the path expression into a join[Feg97a], and then select from the resultant join table. But in BODHI, a specialized index structure, Path Dictionary, is provided for this purpose, which avoids the costly joins altogether.

A Path Dictionary can be built over most often queried paths in the application schema. But the identities of these paths is totally application dependent. The user can build a Path Dictionary using the extended OQL command for the purpose. The syntax of the command would look like *"create path index id1 on Species.statesFound.capital.location;"*.

Queries on Class Hierarchies The queries on class hierarchies, such those illustrated in Section 1.2.1, are not supported in the standard ruleset of the lambda-DB. BODHI provides Multi-key Type Index (MT-Index), for the purpose. Extra optimization rules are provided, which exploit the presence of an MT-Index over the class hierarchy. The user can optionally build an index on a class hierarchy rooted at object X, using the OQL command which would be like *"create mt index id2 on X"*.

Derived Attributes Derived attributes, represented as functions with no parameters, can be treated as though they are value attributes of the object and indexes can be maintained on them. Such functions can depend on other attribute values of the object, and such dependencies cannot be inferred from their implementation. In order to support indexing over derived attributes,

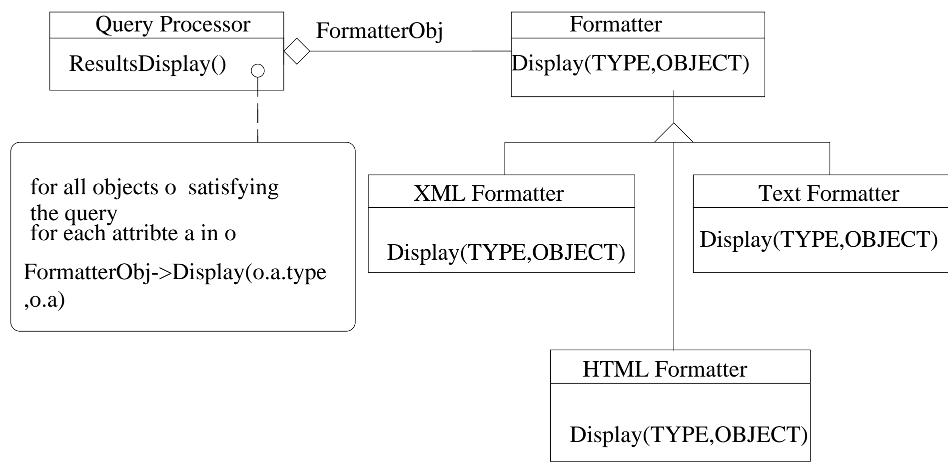


Figure 1.8: Query Processor and Result Formatters

we have extended the method declaration syntax of ODL, to specify a list of attributes on which a derived attribute depends.

Based on this information, whenever an object is created or the attributes of the object are modified, the query processor computes the function and updates the index maintained on it. The following shows a simple example illustrating how to define a function index. The class definition looks like:

```
class Forest (extent Forest_ext){
attribute string  name;
attribute polygon region;
float area() depends on region;
};
```

The OQL syntax to create the index over *area*, looks like:

```
create function index faind on Forest(area);
```

where *faind* is the index identifier.

1.5 Client Interface Framework

As mentioned in Section 1.1, a desirable feature of a bio-diversity data repository system is to support knowledge dissemination, visualization and the ability to exchange data with other similar databases over the Internet. The Client Interface Framework (CIF) component of BODHI provides for these needs.

The CIF consists of a software layer, called *Client Interface Server*(CI-server). that sits on top of the query processor. The CI-Server is responsible for transforming or formatting the query results. The CI-Server is separated from the query processor and can be easily extended to accommodate a new formatter, without having to change or even re-compile the query processor. In order to achieve this extendibility, we use a well known design pattern, *Strategy Pattern*[GHJV95a]. A conceptual class diagram illustrating the usage of this design pattern is given in Figure 1.8. As shown in the diagram, the CI-Server provides a common interface called *Formatter Interface*, from which all other formatter objects are derived. When the Query Processor needs to emit the query results, it requests the CI-Server and obtains an object conforming to the Formatter interface i.e., an object of a formatter class. The implementation of this formatter object is provided in a dynamically loadable library. Since all implementations have a common interface, there is no need for changing the query processor for accommodating a new formatter class.

The implication of separating the data formatting functionality from the query processor is that, during data transformation, no metadata maintained by the Schema Manager is accessible to the CI-Server. But the data transformation needs the meta-data information associated with data. Therefore, a protocol of information exchange between the CI-Server and query processor is implemented for this purpose. This format of data interchange embeds both data and meta-data, and the implementation of Formatter objects takes care of formatting the results according to the transformation rules and returns it to the client. Further implementation details about the CIF are available in [Kon00a].

A useful byproduct of the CIF approach is that it does not distinguish between the two categories of "client"s, viz., the visualization client and databases on the Web that need to exchange data.

1.6 Current Status

Having described the overall design and architecture of the BODHI system, we now overview the current status of our prototype implementation.

The major portion of BODHI is written in C++, the main exception being the query processor, which is implemented in OPTL, the language provided by lambda-DB. Our initial implementation is on a Sun Ultra-1 Workstation running Solaris 2.5. The implementation has been going on for about a year and most of the key components of the system have been completed. The modules of spatial and sequence services have been tested, largely using synthetic data generated by us. Recently, in order to meet our goal of also supporting a lowcost system, we have ported the implementation of all the service modules onto an Intel Pentium-II machine, with GNU/Linux as the operating system. Current work includes, incorporating the genetic sequence similarity algorithms into the query language, implementing the spatial join optimizations and providing bulk loading facilities.

The client interface framework of BODHI was used to develop a browser based graphical visualization client. An XML/DTD, custom designed by combining the relevant portions of *Geography Markup Language* by OpenGIS[OGI] and *ANZMeta DTD Ver1.1*, was used for exchange of information with the client. The visualization client, implemented completely in Java, using the latest visualization features provided by Swing, has capabilities to display the aggregation graphs of the application, and to render the geographical maps using spatial co-ordinates sent by the database server.

We are currently working on populating the database with data already collected by the Center for Ecological Sciences at our institute and to conduct field trials on the useability and performance of the system.

1.7 Related Work

Bio-diversity data consists of both macro-level and micro-level information ranging from ecological information to genetic makeup of organisms and plants. Apart from our work, we are not aware of any other that attempts to combine the complete spectrum of information, though the need for it is highlighted in a recent proposal for *GBIF*(Global Biodiversity Information Facility)[Saa99] by OECD(Organization for Economic Co-operation and Development). This proposal identifies the domain level challenges in building a global, interconnected data repository of bio-diversity information system and notes that the urgent requirement in bio-diversity studies is a suitable information management architecture for handling vast amounts of diverse data.

In the area of macro-level bio-diversity data management, there have been many national level government efforts like *ERIN*[BS94], *INBio*[INB] and global initiatives like *Species 2000*[SPE], *the Tree of Life*[MM98] etc. However, all these efforts have the following drawbacks in their information architectures : (1) Systems are based on the flat relational model, which does not seem to fit well with the bio-diversity data which is naturally hierarchical; (2) Little attention has been paid to supporting data exchange features (a recent exception is the XML/DTD proposal from ERIN, for ecological information exchange[ANZ]); and (3) They do not support ad-hoc querying on their databases by independent clients.
In comparison, our work attempts to develop an architecture that addresses most of these deficiencies.

The micro-level bio-diversity data, or genetic information of various species, has been growing steadily due to a multitude of genome sequencing initiatives. The specific data management issues in handling such data[GRS94, GRS] have been addressed in quite a few proposals. In all of these proposals, the database management architecture has been tailored for the specific purposes of the project. For example, consider the *ACeDB* (A C.elegans Database)[DTM] database system, originally proposed for *C. elegans* genome sequencing project. ACeDB is an object oriented data management tool that has many features that make it a extremely popular software in many sequencing projects[STM99]. ACeDB handles missing data and schema evolution issues, common requirements in a ongoing sequencing projects, in a flexible manner. However, in spite of its popularity in genome sequencing community, it cannot be considered for the larger requirements of bio-diversity data handling due to the following reasons: (1) Its lack of support for geo-spatial data; (2) Weak support for database updates; and (3) The lack of recovery mechanisms necessary in large data repositories.

In BODHI, we have retained the key strengths of ACeDB (its sequencing algorithms and object-oriented basis), and augmented it with strong database functionalities, in addition to other features necessary for a complete bio-diversity information repository.

1.8 Summary

In this report, we have presented the design and implementation of BODHI, a data management system for plant bio-diversity applications. This system, developed in close collaboration with domain experts of bio-diversity studies, is a state-of-the-art solution that provides a generic framework for the requirements of the domain at various levels of data management. Apart from efficiently integrating many datatypes, from genetic to ecological details, BODHI lends itself for flexible data interchange between repositories over the internet. The scalability and the low cost (free) of the system make it suitable for both large and small data collection efforts. The bio-diversity database of flora of

Western Ghats is currently being built over BODHI, intended for use at the *Center for Ecological Studies, Indian Institute of Science.*

We are currently working on improving the efficiency of the genome sequence similarity and pattern matching operations on large volumes of genetic data. In addition, we plan to use BODHI as a platform for further research in various pattern discovery processes over spatial and genome sequence data.

In the remainder of this report, we present in detail each of the components of BODHI. Chapter 2 focusses on Shore, Chapter 3 on taxonomic data, Chapter 4 on spatial data, Chapter 5 on query processing techniques, Chapter 6 on the XML interface, and, finally, Chapter 7 on the system integration and evaluation.

Chapter 2

The SHORE System

SHORE (Scalable Heterogeneous Object REpository) is a 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 **BODHI** database system.

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 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.

2.1 Drawbacks with EXODUS

EXODUS [C+86] is an extensible database system developed by university of Wisconsin prior to the development of SHORE. The main drawbacks of EXODUS are summarized below.

- 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 meta-data.

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.

2.2 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.

2.3 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.

2.4 SHORE Software Components

The main software components of SHORE (Figure 2.1) constitute the SHORE server and the Language Independent Library. We next discuss these components in greater detail.

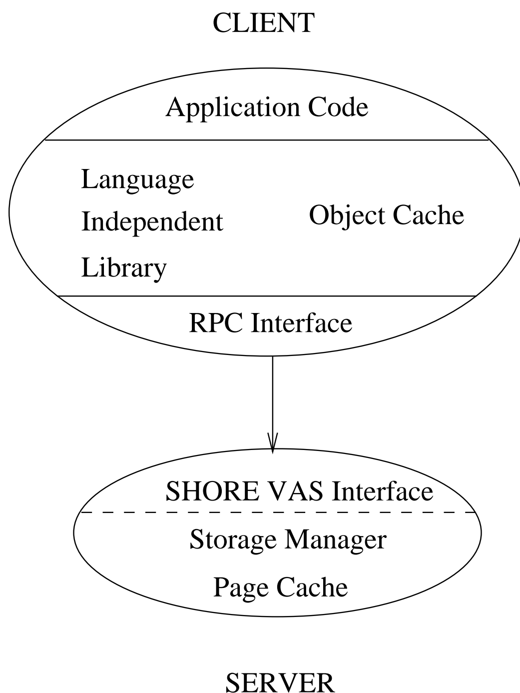


Figure 2.1: Application - Server Interface

2.4.1 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.

2.4.2 The SHORE server

The SHORE server (Figure 2.2) is divided into two main components: a *Server Interface*, which communicates with applications, and the *Storage Manager* (SM), which manages the persistent object store.

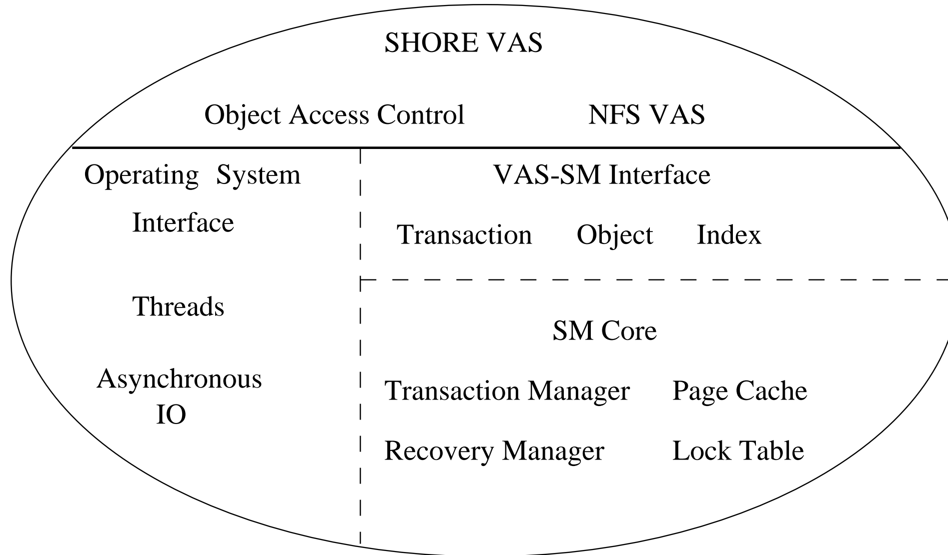


Figure 2.2: SHORE System Architecture

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.

2.5 Value Added Server

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 example for VAS is the NFS file server which is used to *mount* the entire subtree of the SHORE name space 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.

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.

2.6 The VAS-SM Programming Interface

The SHORE Storage Manager (SSM) is a package of libraries for building object repository servers and their clients. These libraries are useful for managing persistent storage and caching of un-typed data and indexes. They also provide disk and buffer management, transactions, concurrency control and recovery. A VAS relies on the SSM for the above capabilities and extends it to provide more functionality. In this section we describe some of the facilities that can be accessed through the VAS-SM programming interface.

2.6.1 Storage Facilities

The SSM provides a hierarchy of storage structures. A description of each type of storage structure is given below.

Devices

A device is a location, provided by the operating system, for storing data. A device is either a disk partition or an operating system file. A device is identified by the name used to access it through the operating system. Each device is managed by a single server.

For each mounted device, the server forks a process to perform asynchronous I/O on the device. These processes communicate with the server through sockets and shared memory.

Volumes

A volume is a collection of file and index storage structures (described below) managed as a unit. All storage structures reside entirely on one volume. A volume has a quota specifying how must large it can grow. Every volume has a dedicated B^+-tree index, called the *root index*, to be used for cataloging the data on the volume.

Files of Records

A *record* is an un-typed container of bytes, consisting of a *tag*, *header* and *body*. The tag is a small, read-only location that stores the record size and other implementation-related information. The header has a variable length, but it limited by the size of a physical disk page. A VAS may store -information about the record (such as its type) in the header. The body is the primary data storage location. A record can grow and shrink in size by operations that append and truncate bytes at the end of the record.

A *file* is a collection of records. Files are used for clustering records and have an interface for iterating over all the records they contain. The number of records that a file can hold is limited only by the space available on the volume containing the file. Methods for creating/destroying files, creating/destroying/modifying records and pinning records for reading and modifying are also provided as part of the interface.

B^+-tree Indexes

The B^+-tree index facility provides associative access to data. Keys and their associated values can be variable length (up to the size of a page). Keys can be composed of any of the basic C-language types or variable length character strings. A bulk-loading facility is provided. The number of key-value pairs that an index can hold is limited only by the space available on the volume containing the index. Routines for creating/destroying indexes, searching and iterating over a range of keys are provided as part of the interface.

R^*-tree Indexes

An R-Tree is a height-balanced tree structure designed specifically for indexing multi-dimensional spatial objects. It stores the minimum bounding box (with 2 or more dimensions) of a spatial object as the key in the leaf pages. The current implementation in SHORE is a variant of R-Tree called R*-Tree [B+90], which improves the search performance by using a better heuristic for redistributing entries and dynamically reorganizing the tree during insertion. All the operations provided for B^+-tree implementation are also provided for R^*-tree.

2.6.2 Transaction Facilities

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

Transactions

A *transaction* is an atomic set of operations on records, files, and indexes. The interface provides methods for beginning, committing and aborting transactions. 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 , the notion of $save - points$ is there. 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 [GR93]. For a file, the hierarchy is volume, file, page, record ; 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.

2.6.3 Crash Recovery Facilities

The crash recovery facilities of the SSM consist of logging, checkpointing, and recovery management.

Logging

Updates performed by transactions are logged so that the 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).

The log is a sequence of log records. The log is stored in Unix files in a special directory. The size and location of the log is determined by configuration options.

Checkpointing

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.

2.6.4 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. Any program using the thread package automatically has one thread. In addition, the SSM starts one thread to do background flushing of the buffer pool and another to take periodic checkpoints. SSM also provides *latches* which 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.

2.6.5 Communication and RPC Facilities

Clients (applications) need a way to communicate with servers. The SSM contains a version of the publicly available Sun RPC package, modified to operate with the SSM's thread package. The SHORE value-added server uses this package.

More details on SSM programming interface can be found in SHORE manual pages.

2.7 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 3

Handling Taxonomy Data

Taxonomy data plays an important role in a biological information system. Object oriented design for handling taxonomy data involves issues such as modeling the taxonomy data and designing efficient access methods for this model. The access methods need to handle aggregation and class hierarchies. Class hierarchy index methods should be able to answer both single class and class hierarchy queries efficiently. Aggregation hierarchy index methods should be capable enough to support general N:M relationships between classes.

In this chapter, we describe the object oriented design and implementation of taxonomy module in BODHI. We have chosen Path Dictionary Index and MT-index as aggregation and class hierarchy indexes respectively. The reasons for these choices are presented and extensions to the Path Dictionary Index method to support N:M relationships are explained.

3.1 Taxonomy Data Model

In this section we look at the analysis and modeling stages of the development of the database management system for managing taxonomy data.

3.1.1 Analysis

In the analysis stage, we study the problem and collect details regarding the types of data that need to be maintained in the database, their constraints and relationships. This information is presented in the form of a problem statement. The following is the statement of the biodiversity data management problem.

The Problem Statement

In taxonomy, each unit of classification is called *taxon*. The four major taxa in the classification of species are *Order, Family, Genera* and *Species*. An order contains several related families, which means the 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. Each taxon has a biological name [Vas74a]. The classification hierarchy of species is shown in Figure 3.1

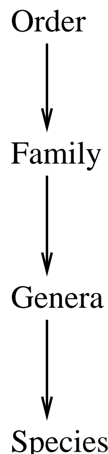


Figure 3.1: The taxonomy of species

Species are identified based on their leaf, stem, flower and fruit characteristics. Hence, these characteristics could be referred to as *identifying characteristics.* There are other characteristics, for example, the characteristic of root, which do not aid in the identification of species. These characteristics could be referred to as *non-identifying characteristics.* A flower is composed of several parts, namely, petals, sepals, inflorosence, 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. Similarly, a leaf consists parts such as blades, sheath, etc. Hence every order, family, genera and species is associated with a set of identifying and non-identifying characteristics which differentiate it from the other orders, families, genera and species respectively. A species may have synonyms, the geographical location details and the textual information associated with it.

The Queries

In the last section, we described the data and the relationships that need to be maintained in the database. We now focus on the query processing requirements of the biodiversity 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*
- *List all species in a given geographic location*
- *List all species with specific characters in a given geographic location*
- *Display the distribution of a species in a geographic location*
- *List all predators of a given species*

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

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

3.1.2 The Object Model

The development of an object model involves various stages such as identifying the classes, associations, attributes and operations, simplifying the classes using inheritance and finally verifying the existence of access paths for likely queries. We have used the Rumbaugh [R+91] notation (given in Appendix A) to pictorially represent the classes, associations and inheritance. We now discuss each of the steps in the object model development of taxonomy data in BODHI.

Identifying Classes

The taxonomy data includes instances of all taxa and their characteristics. The four classes – *Order*, *Family*, *Genera* and *Species* correspond to the four major taxa – the order, the family, the genera and the species. Classes are required to represent the identifying and non-identifying characters. Therefore, we have the classes *Flower*, *Fruit*, *Seed*, *Leaf*, *Inflorescence*, *Sepal*, *Petal*, *Anther*, *Ovary*, *Style*, *Stigma*, etc., to represent the corresponding identifying characters and a class *OtherChar* to represent any non-identifying character.

Identifying Aggregations

Most of the objects dealt with in biodiversity databases are composite, i.e., they exhibit an aggregation hierarchy. The following are some of the aggregations in the biodiversity databases :

- An order *contains* several families. A family in turn *contains* several genera and a genera *contains* closely related species. Since a family cannot belong to two orders, the order-family relationship is modeled as an aggregation rather than an association. Similar arguments hold for genera and species also. Hence, the taxonomy hierarchy is modeled as a four level containment hierarchy in BODHI (Figure 3.2a).
- A flower is an aggregation of inflorescence, sepal, petal, anther, stamen, style and stigma(Figure 3.2b).

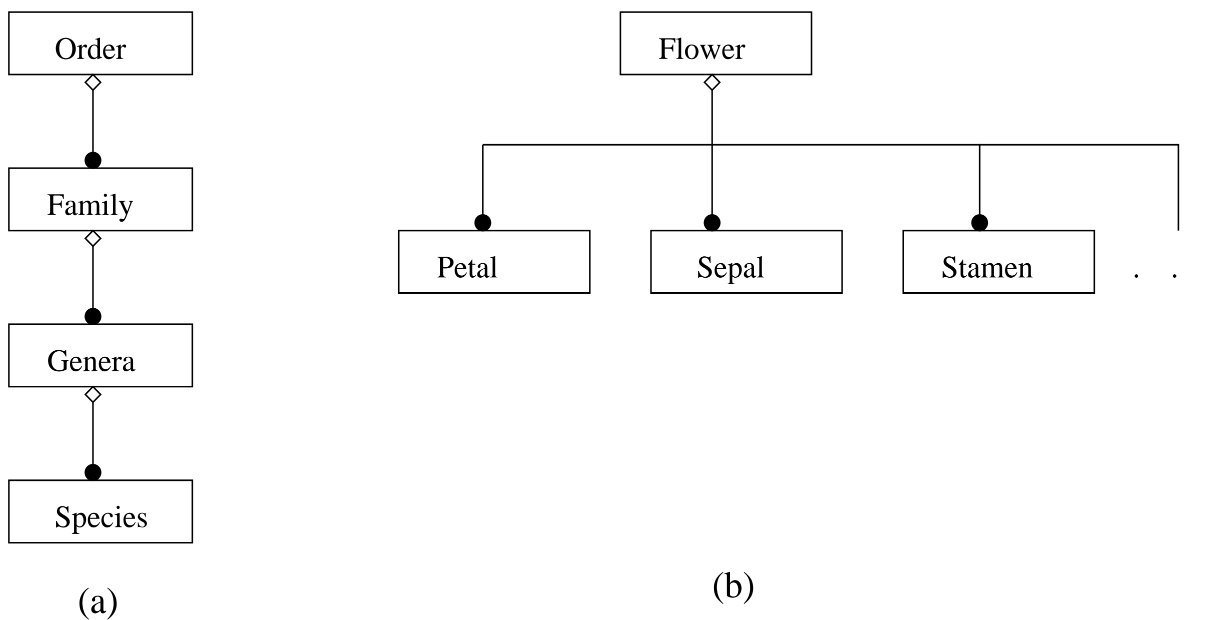


Figure 3.2: Aggregations in taxonomy data model.

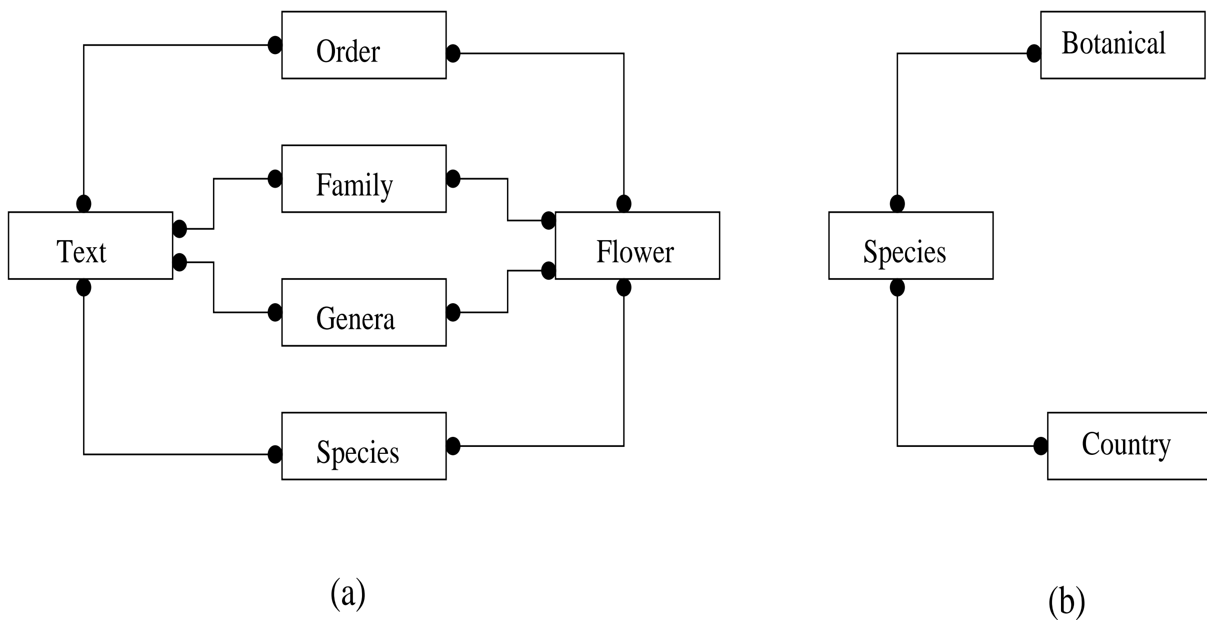


Figure 3.3: Other associations in taxonomy data model

Identifying other Associations

Aggregation is in some sense an association [R+91]. We have seen the various relationships that are modeled as aggregations in BODHI. The other associations between object classes are described below:

- Every order, family, genus and species is associated with some identifying characters. Since taxa exhibit a large amount of sharing of characteristics, an identifying character could be associated with many taxa. Therefore, an identifying character cannot be said as *belonging to* a specific taxon. Hence, the relationship between the taxa and the identifying characters is modeled as an association rather than an aggregation between the corresponding classes. Every taxon is also associated with the text class. The text object models the textual information about any entity (Figure 3.3a).

- A species is found in many geographic organizations and botanical gardens. This feature is modeled as two associations, one between species and geographic organization (could be a country, city, state, etc.), and the other between species and botanical gardens. A geographic organization or botanical garden may have several species associated with it (Figure 3.3b).

- Information about a species is found in many articles and this is represented by the association between the species class and the article class.

Identifying the Attributes and Operations of Classes

Attributes could be viewed as properties of objects that are not objects by themselves
[R+91]. For example, a taxon has *name* as an attribute. A petal has attributes such as *aestivation, colour, apex form, shape, etc.*, and a sepal has attributes such as *symmetry, form, etc.* Most of the methods for the classes are for reading, updating the attributes or objects contained in them (in case of aggregations). The attributes and methods of some of the object classes are shown in the object diagram given in Appendix B.

Organizing and Simplifying Classes using Inheritance

In this phase, we organize the existing classes by identifying those classes that share similar attributes, methods and associations.

The Taxon Class and Ident Taxon Class

Order, family, genera and species are *Taxons*. The Taxon class generalizes order, family, genera and species. A taxon class has *OtherChar* and *text* associated with it. Order, family and Genera have identifying characters. Hence the three are generalized by *Ident Taxon* class. The classes *Tree*, *Climber* and *Shrub* are specializations of species. Tree has additional attributes such as the bark characteristics. Similarly, climber and shrub have additional attributes.

In this section, we have discussed the analysis and modeling of taxonomy data in BODHI. The problem was stated formally and classes, aggregations and associations were identified. The classes were then organized using inheritance.

So far, the model has been analyzed only in the problem domain and we now shift to the implementation domain. In the implementation phase, we mainly concentrate on the implementation of efficient access methods. Different access methods which can be used to answer the queries on class and aggregation hierarchies are discussed in the next two sections.

3.1.3 Class Hierarchy Indexing Methods

In this section, we discuss the class hierarchy indexing methods that can be used to efficiently answer the queries on class hierarchies. We will also explain the reasons behind choosing MT-index as a class hierarchy indexing method in BODHI.

Basic Problems

An implication of the class hierarchy concept in object-oriented data models (OODM), on query evaluation is the *class scope* of the query; that is, the classes over which the query has to be evaluated. An object-oriented query on a class C has two possible interpretations. In a *single class query*, objects are retrieved from only the queried class C itself. In a *class hierarchy query*, objects are retrieved from all the classes in the class hierarchy rooted at C since any instance of a subclass of C is also an instance of C [BO99]. To facilitate the evaluation of such types of queries, a *class hierarchy index* needs to support efficient retrieval of objects from a single class, as well as from all the classes in the class hierarchy.

There are two approaches to class-hierarchy indexing [BO99]:

1. *Class-dimension-based* approach partitions the data space primarily on the class of an object.

2. *Attribute-dimension-based* approach partitions the data space primarily on the indexed attribute of an object.

Space partitioning of both approaches is illustrated in figure 3.4. While class dimension-based approach supports single-class queries efficiently, it is not effective for class-hierarchy queries due to the need for traversing multiple single-class indexes. On the other hand, the attribute-dimension-based approach generally provides efficient support for class-hierarchy queries on the root class, but is inefficient for single-class queries or class-hierarchy queries on a sub-hierarchy of the indexed class hierarchy, as it may need to access many irrelevant leaf nodes of the single index structure.

Related Work

Several class hierarchy indexing methods are proposed in the literature and most of them are based on B^+-tree [Com79] structure. Class Hierarchy Trees (CH-trees) were proposed in [KKD89] and essentially maintains a single B^+-tree index for all classes of the class hierarchy. It clusters the OIDs of objects of all classes in the class hierarchy for a given value of the indexed attribute. H-trees were proposed in [LOL92] and the central idea here is that one B^+-tree index per class in the class hierarchy is maintained but the indexes are nested according to their subclass-superclass relationship. Using a replication scheme for OIDs, the class division approach [RK95] trades storage space and update performance for query performance. hcC-trees [SS94] focus on a more general problem, namely

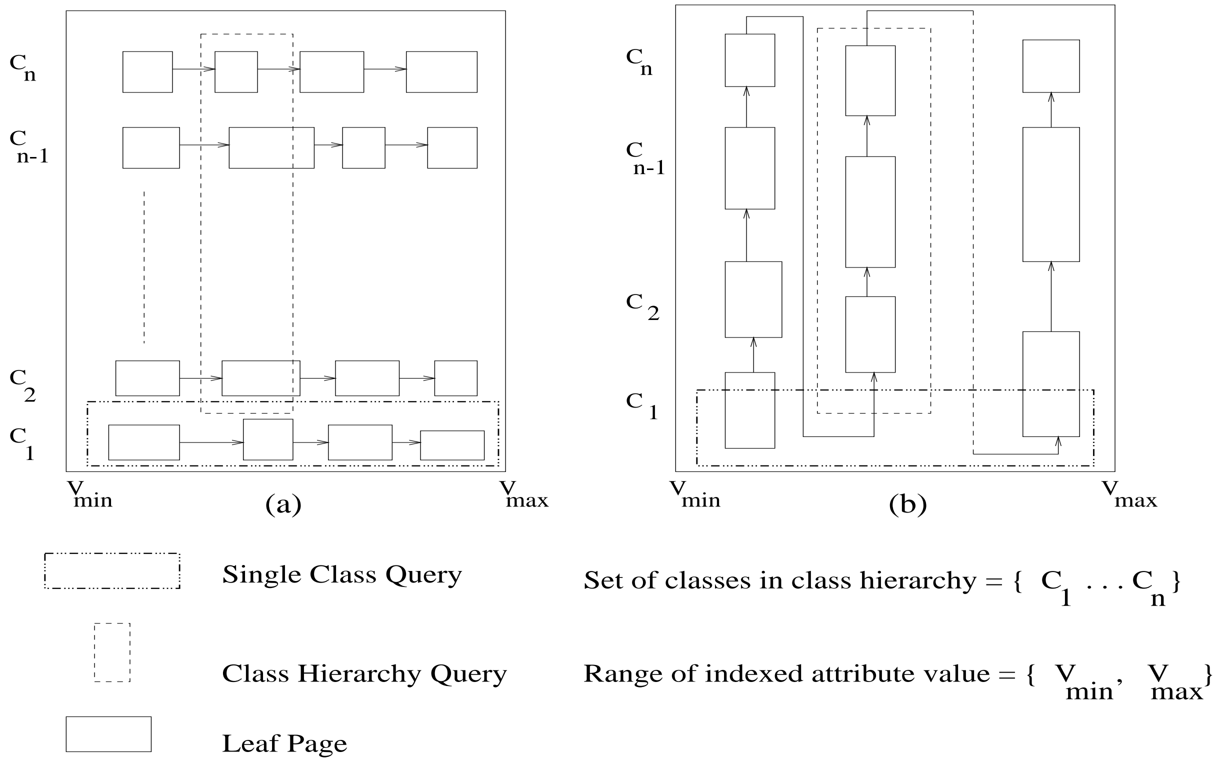


Figure 3.4: Space Partitioning: (a) class-based (b) attribute-based

set membership. It augments B^+-trees by multiple lists to group OIDs with respect to set membership. But the structure results in doubling of space.

Note that CH-trees are an example for the attribute-dimension based approach whereas H-tree are an example for the class-dimension based approach. Class division and hcC-tree approaches occupy the middle ground between these two extremes.

Our Choice: MT-index

To support both single class and class hierarchy types of queries efficiently, the index must support both ways of data partitioning. *Multikey Type index* (MT-index) [MP97b] is an example for this kind of approach.

Linearization Algorithm

The prerequisite behind MT-index is a linearization algorithm which maps type hierarchies to linearly ordered attribute domains in such a way that each sub-hierarchy is represented by an interval of this domain. Using linearization algorithm, MT-index incorporates the type hierarchy structure of a given database scheme into a standard multi-attribute search structure in such a way that the hierarchy is mapped to one of the attribute domains (called *type domain*). The result is an index with $K+1$ keys corresponding to the K indexed object attributes and one additional key representing type membership. Note that by using MT-index, not only single-class and class-hierarchy queries can be supported, but answering multi-attribute queries also becomes straight forward. Since linearization algorithm is independent of the multi-attribute data structure, any multi-attribute or multidimensional data structure such as hB-tree [LS90] or R^*-tree [B$^+$90] respectively, can be used to implement the index.

Optimal Linearization

Assuming an arbitrary linearization, the query range in the type domain may also contain types not qualifying for the query request. Since the resource consumption of a range query is positively correlated with the size of the respective range, a linearization should be in such a way that for each possible type in a query a subspace should be present which does not contain any object identifiers not belonging to the query result. Linearizations satisfying the above property are called *optimal* linearizations.

Necessary and Sufficient Conditions for Optimal Linearization

Sometimes it may not be possible to obtain an optimal linearization of a hierarchy using linearization algorithm. One necessary and one sufficient condition for the existence of the optimal linearization are given in [MP97b]. They are:

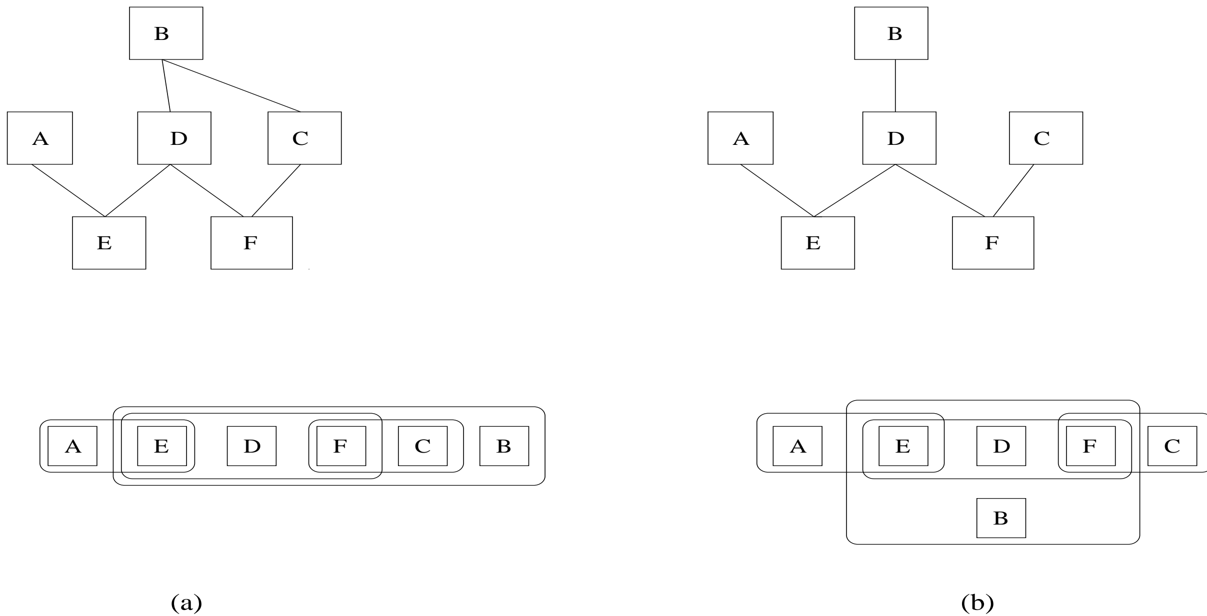


Figure 3.5: Type hierarchies and their corresponding set diagrams (a) with and (b) without optimal linearization.

1. An optimal linearization exists if each type has at most one super-type.

2. An optimal linearization does not exist if any type has more than two super-types.

In the above conditions, first condition is sufficient condition where as second one is only necessary condition. That means, we can always find an optimal linearization in case of single inheritance and we can never find an optimal linearization if degree of multiple inheritance is greater than two. But if the degree of multiple inheritance is two, it may or may not be possible to obtain an optimal linearization. Example for these two cases are shown in figure 3.5(a) and 3.5(b) respectively. Detailed description of the linearization algorithm, necessary and sufficient conditions for optimal linearization, etc., can be found in [MP97b].

3.1.4 Execution of Queries

Given the attribute value and the class name, the single class query becomes a *point query* and the class hierarchy query becomes a *region query* in a two-dimensional data space. Since, the algorithms for evaluating point and range queries varies depending on the multi-dimensional structure that is used in MT-index, they are not discussed here. Searching algorithms for some multi-dimensional access structures can be found in [Che00].

3.1.5 Advantages of MT-index

In nutshell, the advantages of using MT-index can be stated as follows:

- Using two way partitioning of data space, MT-index can answer both single and class hierarchy queries with equal efficiency.

- By extending the number of dimensions, multi-attribute queries can be easily supported with a single index structure. This not only reduces the space overhead, but also improves the retrieval time.

- The linearization algorithm assures good performance by minimizing the query space on which the query is being evaluated.

The above advantages and SHORE support for the R^*-tree have motivated us to decide MT-index as a choice for class hierarchy index method in BODHI.

3.2 Aggregation Hierarchy Indexing Methods

In this section, we discuss the aggregation hierarchy indexing methods that can be used to efficiently answer the queries on aggregation hierarchies. We will also explain the reasons behind choosing Path Dictionary Index as an aggregation hierarchy indexing method in BODHI.

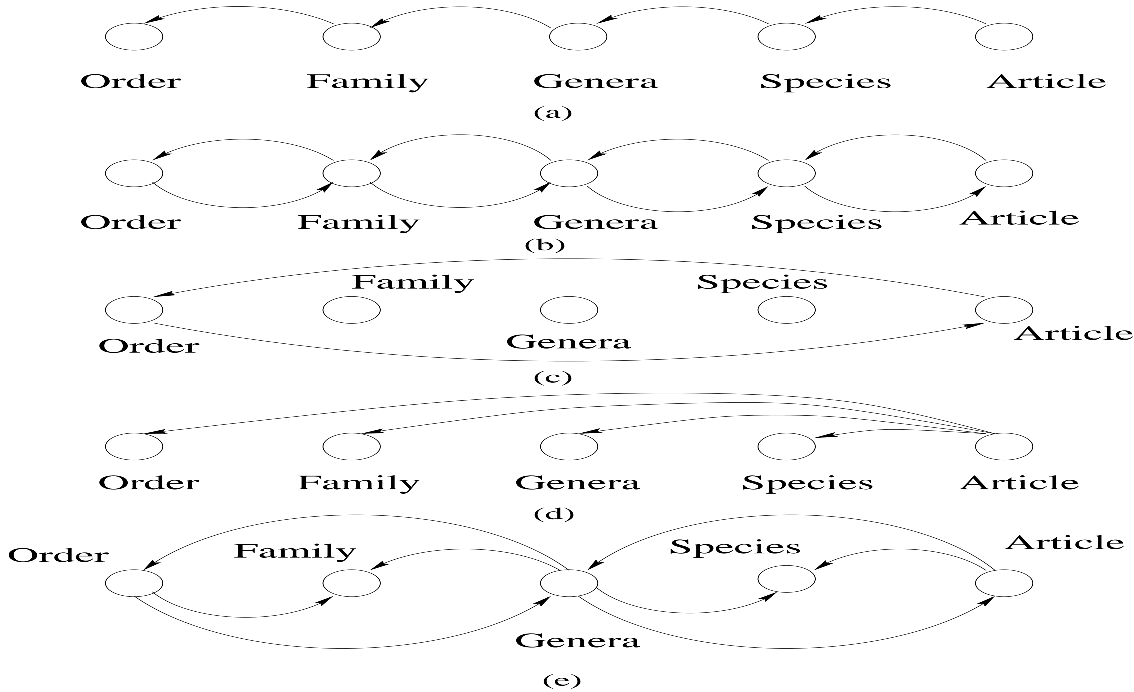


Figure 3.6: Indexing graphs: (a) multi-index; (b) join index; (c) nested index; (d) path-index; (e) path splitting.

3.2.1 Basic Problems

The queries that can be asked on aggregation hierarchies can be classified into the following categories [LL98b]:

- TP: The target class is an ancestor class of the predicate classes.
- PT: The target class is a nested class of the predicate classes.
- MX: The target class is an ancestor class of some predicate class and a nested class of some predicate class.

where, *target class*es are the classes from which objects are retrieved and *predicate class*es are the classes involved in the predicates of the query.

There are three basic approaches to evaluating a nested query: *top-down, bottom-up,* and *mixed* evaluations. The top-down approach traverses the objects starting from an ancestor class to a nested-class. Since the OID in a parent object leads directly to a child object, this approach is also called a *forward traversal* approach. On the other hand, the bottom-up method, also known as *backward traversal*, traverses up the aggregation hierarchy. A child object, in general does not carry the OID of (or an inverse reference to) its parent object. Therefore, in order to identify the parent object(s) of an object, we have to compare the child object's OID against the corresponding complex attribute in the parent class. Mixed evaluation is a combination of the top-down and bottom-up approaches, which is often required for complex queries.

Note that top-down approach is more effective for PT queries. The bottom-up approach, in contrast, no matter where the predicate classes are located, will always scan through all objects in the classes on the path (assuming no reverse references are present).

3.2.2 Related Work

Many of the indexing structures proposed are based on precomputing traversals along aggregation hierarchies. Figure 3.6 illustrates well-known aggregation hierarchy indexing schemes by using the notion of *indexing graph* [BO99]. An indexing graph contains a node for each class in the indexed path; it moreover contains an edge from the node representing a class C to the node representing a class C' if there is an indexing relationship from class C to class C'. An indexing relationship from class C to C' means that an entry in the index contains as key values the identifiers of the instances of class C and it associates with each such key value the identifiers of the instances of class C' that reference, directly or indirectly, the key value. The multi-index [MS86] approach maintains an inverted index for each edge. As can be seen from figure 3.6(a), only backward traversal is supported. The second approach is that based on the well known join index [Val87], where two B^+-trees are maintained for join pairs between two classes. Both forward and backward traversals are supported, but the update cost and storage cost is much higher than before (see figure 3.6(b)). For both methods, the number of indexes that need to be accessed for navigational access is proportional to path length. To alleviate this problem, direct association between objects of the first class and

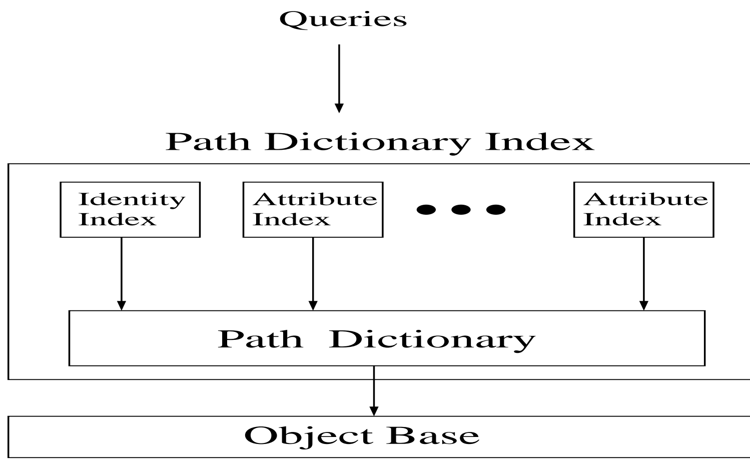


Figure 3.7: Path Dictionary Index

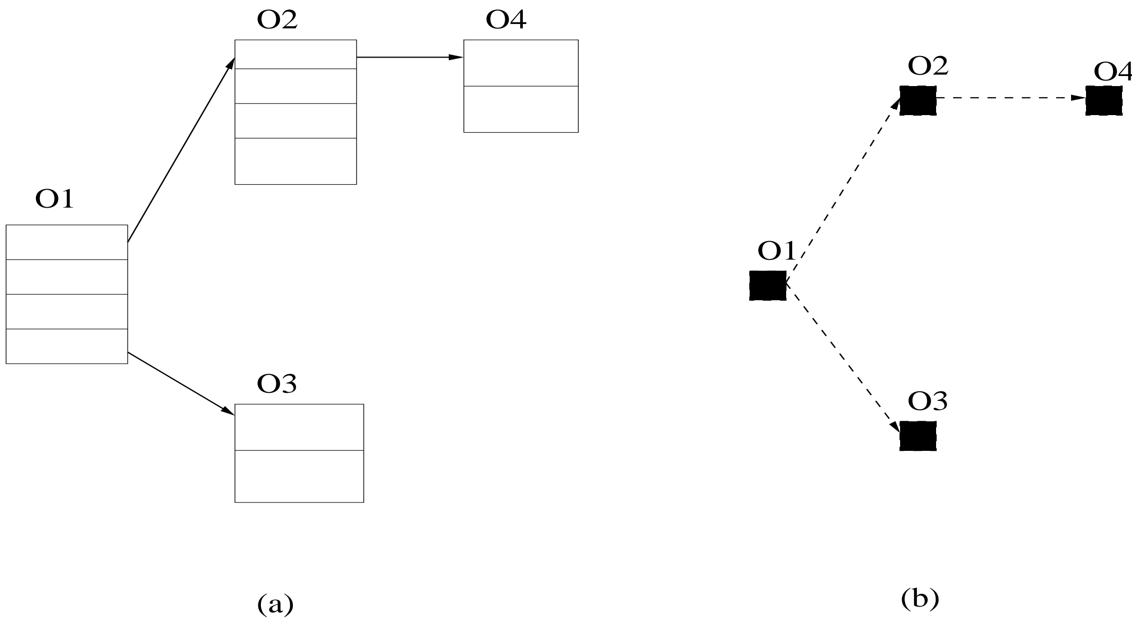


Figure 3.8: (a) A database instance; (b) Path information.

last class along the path are maintained in a single nested index [EK89]. The major problem of such an index is update operations that require access to several objects in order to determine the entries that need update (see figure 3.6(c)). The nested index can be further generalized to support access from the objects to all parent objects [EK89], as shown in figure 3.6(d). To reduce update overhead and yet maintain the efficiency of path indexing structures, paths can be broken into sub-paths, which are then indexed separately [Ber94]. Figure 3.6(e) shows example where a path is split into several sub-paths and different indexing techniques are allocated on each sub-path.

3.2.3 Our Choice: Path Dictionary Index

Path Dictionary Index (PDI) is recently proposed as an aggregation hierarchy indexing method in [LL98b]. The PDI consists of three parts: the *path dictionary* supports efficient object traversal and the *identity index* and *attribute indexes* support associative search. The identity and attribute indexes are built on top of the path dictionary. Figure 3.7 illustrates the overall architecture of the path dictionary index.

Upon receiving the query, the query processor will evaluate the predicates if any, using the attribute indexes and then traverse to the target classes using the path dictionary. Each of these structures are explained in detail below.

Path Dictionary

An object-oriented database may be viewed as a space of objects connected with links through complex attributes. Figure 3.8(a) shows some object instances corresponding to an aggregation hierarchy. Figure 3.8(b) is a *conceptual*

path dictionary storing the connections among the objects in the database. General speaking, the Path dictionary extracts the complex attributes from the database to represent the connections between objects. Since primitive attributes are not stored in the path dictionary, it is much faster to traverse the nodes in the path dictionary than objects in the database. Another advantage of this access structure is that it provides shortcuts for both forward and backward traversals of the objects on a given path. As a result, it is suitable for general queries, whether they imply top-down or bottom-up evaluation.

Attribute Index

While path dictionary supports fast traversal among objects, it by itself will not help predicate evaluation which involves finding objects meeting certain conditions specified on their attribute values. To facilitate associative search, the PDI provides attribute indexes which map attribute values to the OIDs in the path dictionary corresponding to the attribute values. As many attribute indexes as necessary can be built on top of path dictionary.

Identity Index

Since OIDs are used to describe the path information among objects, it is often necessary to obtain from the path dictionary path information associated with a given OID. In order to efficiently support this operation, an identity index is provided to map OIDs to the locations in the path dictionary where the OIDs can be found. Since identity search is important for retrieval and update, the identity index significantly reduces the cost for retrieval and update operations.

s-Expression Scheme

Implementation of path dictionary using s-expression schemes is suggested in [LL98b]. The s-expression scheme encodes into an expression all paths emanating from the same object. The s-expression for the path $C_1, C_2, ..C_n$ is defined as follows:

- $S_n = \theta_n$, where θ_n is the OID of an object in class C_n or null.
- $S_i = \theta_i(S_{i+1}[, S_{i+1}])$ $1 \leq i < n$, where θ_i is the OID of an object in class C_i or null, and S_{i+1} is an s-expression for the path $C_{i+1} \dots C_n$.

S_i is an s-expression at i^{th} level in which the list associated with θ_i contains recursively the OIDs of all descendant objects of θ_i. Except for the objects in C_n, every object on the path has a descendant list, which may be empty.

An advantage of the s-expression scheme is that every object on the path appears only once in the path dictionary, thus avoiding redundant partial path information introduced in other schemes. Due to the inherent tree structure, the s-expression scheme supports naturally 1:1 and 1:N relationships.

Extensions to Path Dictionary Index

The implementation of path dictionary given in [LL98b] works only for either 1:1 relationships or 1:N relationships. But in BODHI, the possibility of N:M relationships can not be overlooked. This can be observed from the relationship between the species and their available geographical locations. Note that a species can be found in any number of geographical locations and a single geographical location may contain number of different species. This forms an N:M relationship between species and their geographical location.

Similarly different types of relationships between objects are possible in BODHI, such as *sets, bags* and *sequences* where set is a collection of references, bag is a collection of references with possible duplicates and sequence is an ordered collection of references. Each of these relationships need different treatment at the implementation level to preserve the correct relationship and at the same time to reduce redundancy without compromising on retrieval time.

We extended the implementation given in [LL98b] to support each of the above discussed additional requirements with some modifications at the data structure level. The main idea behind the modification is to break the graph of N:M relationship into multiple 1:N relationships. By doing this, we can continue using the s-expression scheme to represent each of the 1:N relationships. But this straight forward modification introduces lot of problems, which are described below in detail with relevant examples. Their corresponding solutions are also proposed.

Extensions to support N:M relationships with sets and references

Look at the example N:M relationship graph shown in figure 3.9(a). If we break this into multiple 1:N relationships, the graphs and the corresponding s-expressions looks like in figure 3.9(b). Note the redundancy in these s-expressions. The children of B_1 are replicated in both of the s-expressions of A_1 and A_2. This problem can be solved by using a flag in the entries of s-expression. The flag denotes whether the entry is a direct reference or an indirect reference. All the descendant entries of an OID will be stored only in the entry which contains direct reference to that OID. This modification is shown in form of a graph in figure 3.9(c) with their corresponding s-expressions. Note the suffix

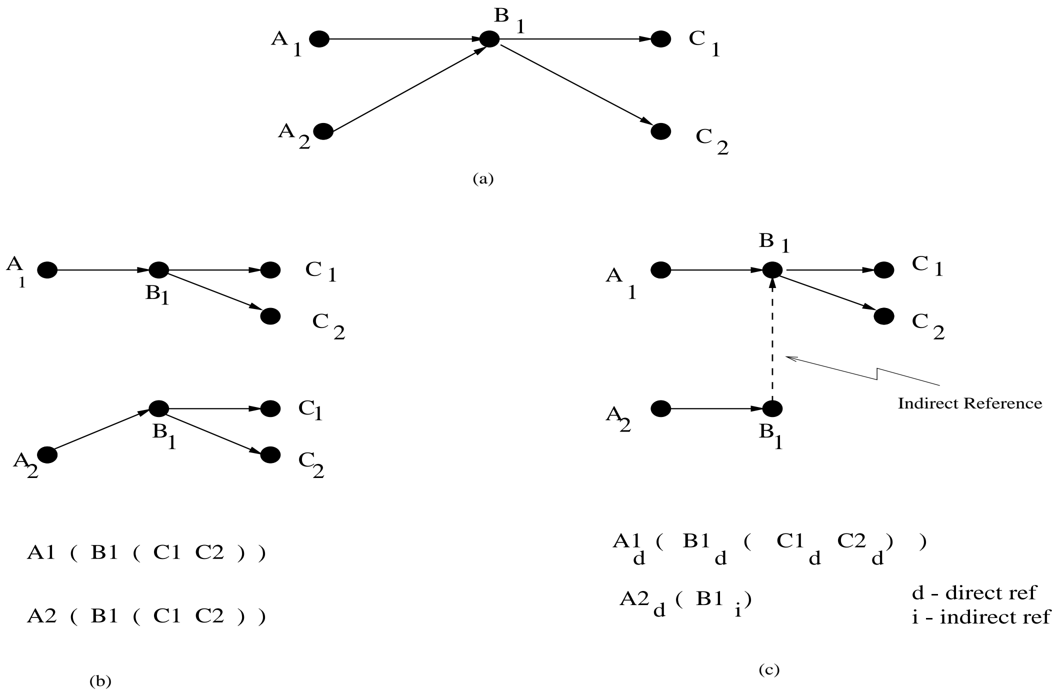


Figure 3.9: Representing N:M relationships and Problems: (a) N:M relationship (b) Equivalent 1:N relationships (c) Equivalent 1:N relationships with indirect references.

for each entry which denotes whether it is a direct reference or an indirect reference. With this modification, the B_1 entry is duplicated (with different flag values), but we are avoiding duplicating the children of B_1 which saves lot of space. This modification immediately affects the structure of the identity index. In original implementation, since only 1:N relationships are supported, with s-expression scheme it is guaranteed that each OID appears in only one s-expression. But with the above modification, now a single OID can appear in more than one s-expression (for ex: B_1). So, the identity index should be modified in such a way that it keeps track of addresses of all the s-expressions in which a given OID appears.

Extensions to support N:M relationships with bags

The above modification works fine for storing ordinary references and sets. But still redundancy is possible if bags are represented with the above implementation. The example for this is shown in figure 3.10. The number on edge from a to b denotes the number of times b appeared as reference in the bag of a. The corresponding s-expressions for this graph using above implementation are given in figure 3.10(b). Note that the entry of B_1 is repeated n number of times in each expression, where n denotes the number of times reference to B_1 is contained in parent object. This replication can be eliminated by introducing one more field in the entry of s-expression which stores this replication count. This reduces the storage overhead for storing bags since OIDs are not duplicated. The s-expressions with this modification are shown in figure 3.10(c).

Extensions to support N:M relationships with sequences

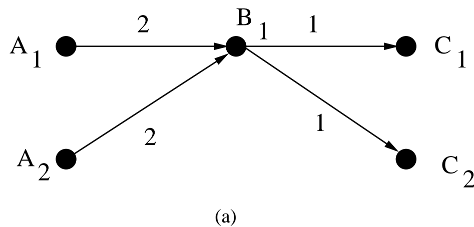
The implementation can support sequences also if the order of the children of a given parent is maintained in the s-expressions.

3.2.4 Implementation of the Data Structures

In this section, we describe the physical structures by which the s-expressions, attribute and identity indexes can be implemented.

s-Expression

Figure 3.11(a) illustrates the data structure of an s-expression for $C_1C_2\dots C_n$. SP_i in the header points to the first occurrence of θ_i in the s-expression. Following the SP_i fields is a series of $< Prev, OID, Next >$ entries. The data structure mimics the nesting structure in the s-expression. The OIDs in the data structure are in the same order as the OIDs in the unwrapped s-expression. The $Prev$ and $Next$ values associated with θ_i points to the previous and



(a)

$A1_d$ ($B1_d$ ($C1_d$ $C2_d$) $B1_i$)

$A2_d$ ($B1_i$ $B1_i$)

(b)

A1 <d 1> (B1 <d 2> (C1 <d 1> C2 <d 1>))

A2 <d 1> (B1 <i 2>)

(c)

Figure 3.10: Representing N:M relationships and Problems: (a) N:M relationship (b) Equivalent 1:N relationships with just indirect references (c) Equivalent 1:N relationships with indirect references as well as replication count

next occurrence of θ_i in the s-expression. Using the SP_i, $Previous$ and $Next$ values, we can easily trace the nested relationship among the objects in an s-expression. For example, to obtain the descendant list associated with θ_i, we simply collect the OIDs stored after θ_i until we reach the OID pointed to by θ_is $Next$ value. An s-expression and its representation are shown in Figure 3.11(b) and 3.11(c) respectively.

Note that additional information such as *direct/indirect flag* and *replication count* can also be stored in the entries of s-expression. But they are not shown in the figure to make it simple.

An advantage of this representation is that it allows fast retrieval of OIDs in the same class. To retrieve all OIDs for class C_i, we start with SP_i, which will lead us to the first θ_i in the s-expression. Following the associated *Prev* and $Next$ pointers, we can easily traverse the list of OIDs belonging to the class C_i, skipping the OIDs of irrelevant classes.

Identity Index

Figure 3.12(a) and Figure 3.12(b) show the structures of a leaf node record and non-leaf node page for B^+-tree implementation of the identity index, respectively. In the identity index, OIDs are used as the key value. The $s - address$es in the leaf node record are the addresses of the s-expressions corresponding to the OID in the same leaf node record. The page pointers in a non-leaf node are pointing to the next level of non-leaf nodes or to the leaf nodes. Additional information can be stored in the leaf nodes depending on the requirements.

Attribute Index

Figure 3.12(c) and Figure 3.12(d) show the structures of an attribute index's leaf node record and non leaf node page. In the attribute index, attribute value is used as the key. The OIDs in the leaf node record are the object identifiers of the objects which are having the attribute value in the same leaf node record. The page pointers in a non-leaf node are pointing to the next level of non-leaf nodes or to the leaf nodes.

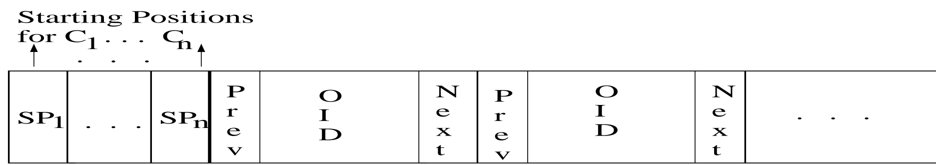
3.2.5 Execution of Queries

We next discuss some sample TP, PT and MX queries and see how they are solved efficiently using the Path Dictionary Index. We assume that path dictionary index for the path *Order - Family - Genera - Species - Article* has been created.

TP Queries

Given the name of the species, find its order.

Assuming an attribute index is available on attribute *species.name* , first the species name is used as key to find all the OIDs of class *species* which satisfies the predicate. These OIDs are then used as keys in the identity index to find the corresponding s-expression addresses in the path dictionary. These s-expressions are then retrieved and scanned to find the OIDs of parent class *order*. Since we are eliminating the need for traversal of intermediate objects



(a)

Order[1] (Family[1] (Genera[1]) , Family[2] (Genera[2] , Genera[3]))

(b)

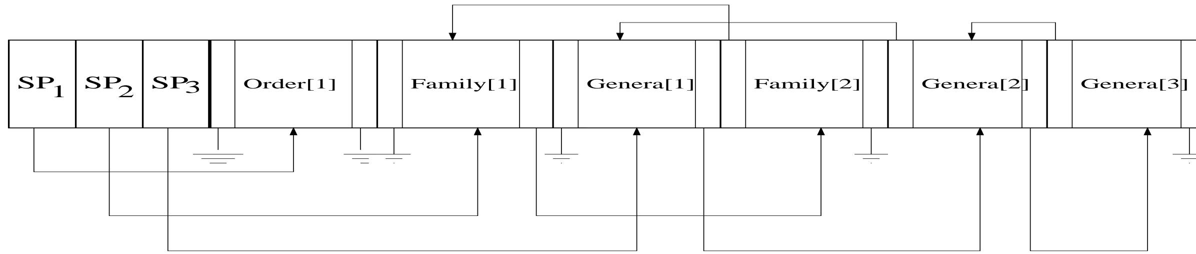


Figure 3.11: Data structure of an s-expression.

belonging to classes *family* and *genera*, the time for evaluating query will get reduced. Due to the structure of path dictionary, the backward traversal of the aggregation hierarchy also becomes easy.

PT Queries

Given the name of the order, find all the species under it.

Assuming an attribute index is available on attribute *order.name*, first the order name is used as key to find all the OIDs of class *order* which satisfies the predicate. These OIDs are then used as keys in the identity index, to find the corresponding s-expression addresses in the path dictionary. These s-expressions are then scanned to retrieve all the descendant OIDs belonging to the class *species*. The *previous* and *next* pointers will facilitate the traversal of the class lists in the s-expressions.

MX Queries

Given the name of the order and the name of the article, find all the species which belong to that order and appear in the article specified.

There are different ways to evaluate the mixed queries. Here, we will explain one simple way of evaluating this query. First, *order.name* is used to retrieve all the species under that order. Evaluation of this is similar to the evaluation of PT query discussed in previous section. Then, *article.name* is used to retrieve all the species which appeared in the article specified in the predicate. Evaluation of this is similar to the evaluation of TP query discussed above. Then the intersection of these two sets of OIDS are taken to find the answer to the query.

3.2.6 Advantages of Path Dictionary Index

The main advantages of the Path Dictionary Index method can now be summarized as follows:

- Since the path dictionary stores the structural relationships and attribute indexes hold the values, several attribute indexes can be built on top of the same path dictionary, there by reducing the space overhead.

- Since both forward and backward traversals are supported by the path dictionary, it is possible to evaluate both PT and TP queries with equal efficiency.

- Since intermediate objects need not be traversed, the path dictionary improves the retrieval time.

- Identity index reduces the update overhead by directly pointing the s-expressions where update is to be done.

- Path Dictionary Index can be used to store N:M relationships with the modifications suggested.

The above advantages have motivated us to decide Path Dictionary Index as a choice for aggregation hierarchy index method in BODHI.

OID	s-address

(a)

OID	Page Pointer	. . .	OID	Page Pointer

(b)

Key length	key value	no. of entries	<OID> , . . . , <OID>

(c)

key value	page pointer	key value	page pointer	. . .	key value	page pointer

(d)

Figure 3.12: Path Dictionary Index: (a) leaf node record of the identity index; (b) non-leaf node of the identity index; (c) leaf node of an attribute index; (d) non-leaf node of an attribute index.

3.3 Implementation Issues

In this section we discuss the issues involved in implementing the taxonomy data model and in writing the value added servers for path dictionary index and MT-index.

3.3.1 Data Model

The data definition language for BODHI is based on Object Definition Language (ODL) [Cat93]. Any data model to be represented in BODHI should be written in this language. The ODL representation of the taxonomy data model detailed in the section 3.1 will be converted into internal representation by the Query Processor module of BODHI. This information is then used by the Query Processor to answer the queries written in OQL (Object Query Language) as shown in figure 3.13. The detailed diagram and discussion on these conversions are beyond the scope of this paper. More details on query processing in BODHI can be found in [Kon00b].

3.3.2 Path Dictionary Index

As mentioned earlier, SHORE supports the notion of "value-added" server (VAS). The server code is modularly constructed to make it relatively simple for users to build application-specific servers. The Path Dictionary Index (PDI) is implemented using this VAS programming interface (see figure 3.13).

SHORE storage manager is responsible for providing persistence of data and recovery mechanisms. The PDI server interacts with the storage manager by using some predefined function calls. Communication between the executable query and PDI server is implemented using RPC mechanism. Executable query sets up a connection and works as a client under PDI server. Different commands are implemented such as creation and destroying indexes, retrieval functions and iterators. These commands can be called by the executable query using RPCs which will then be executed by the PDI server and the results are returned back to the executable query.

The path dictionary is implemented as a file in SHORE storage manager and each s-expression is stored as a record in that file. SHORE supports two types of B^+-tree implementations namely *unique* and *non-unique* B^+-trees. Identity index is implemented using non-unique B^+-tree implementation, because more than one element (i.e., s-expression address) might exist for a given key (i.e., OID). Attribute indexes can use any of the above implementations, depending on the attribute on which the index is being built.

3.3.3 MT-index

SHORE supports a multi-dimensional index data structure called R^*-tree [B+90]. As discussed in section 3.1.3, any multi-dimensional data structure can be used along with linearization algorithm. To simplify the coding effort, we decided to use already available R^*-tree as multi-dimensional data structure in the implementation of MT-index.

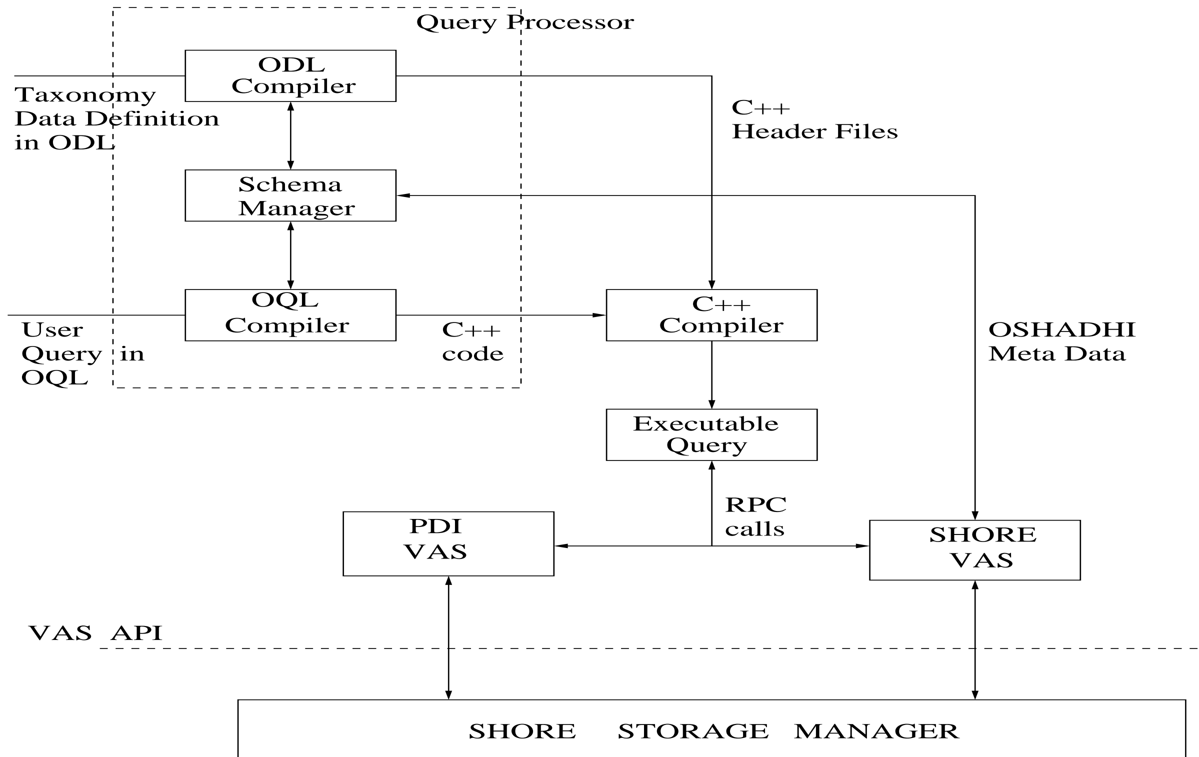


Figure 3.13: The high level diagram of data flow in query processing.

3.3.4 Summary

On the one hand, object oriented modeling framework gives rich modeling power, but, on the other hand, it leads to arbitrary deep aggregation and class hierarchies, which need indexing schemes different from those found in traditional database systems. In this thesis, we have focused on two such indexing schemes viz., MT-index and Path Dictionary Index, within the preview of object data model of taxonomy data in BODHI.

Path dictionary index method is chosen as aggregation hierarchy index, because of the separation between the structure and content (i.e., path dictionary stores the structural relationships while the attribute indexes hold values), which makes it possible to use different attribute indexes on the same path dictionary, there by reducing space overhead as well as improving the retrieval time. MT-index is chosen as class-hierarchy indexing method, because it provides two way data partitioning of space, which makes it to support both single class and class hierarchy queries with reasonable efficiency. The linearization algorithm improves the performance of MT-index by minimizing the query space on which the query is being evaluated.

It would be useful to implement mining of taxonomy data to find some useful relationships among various species. For example, we might be able to find all species that can grow together in a particular set of climatic conditions.

Chapter 4

Handling Spatial Data

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 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 [KF94b] index among the variants of R-tree based methods, because of the reasons that will be explained later in section 4.2. All these issues are discussed in detail in the remainder of the thesis.

4.1 Modeling of Spatial Data

In this section, we look at the modeling of spatial data in Bodhi. 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.

4.1.1 The Object Model

Figure 4.1 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 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.

4.1.2 Spatial Data Types and Relationships

Bodhi 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.

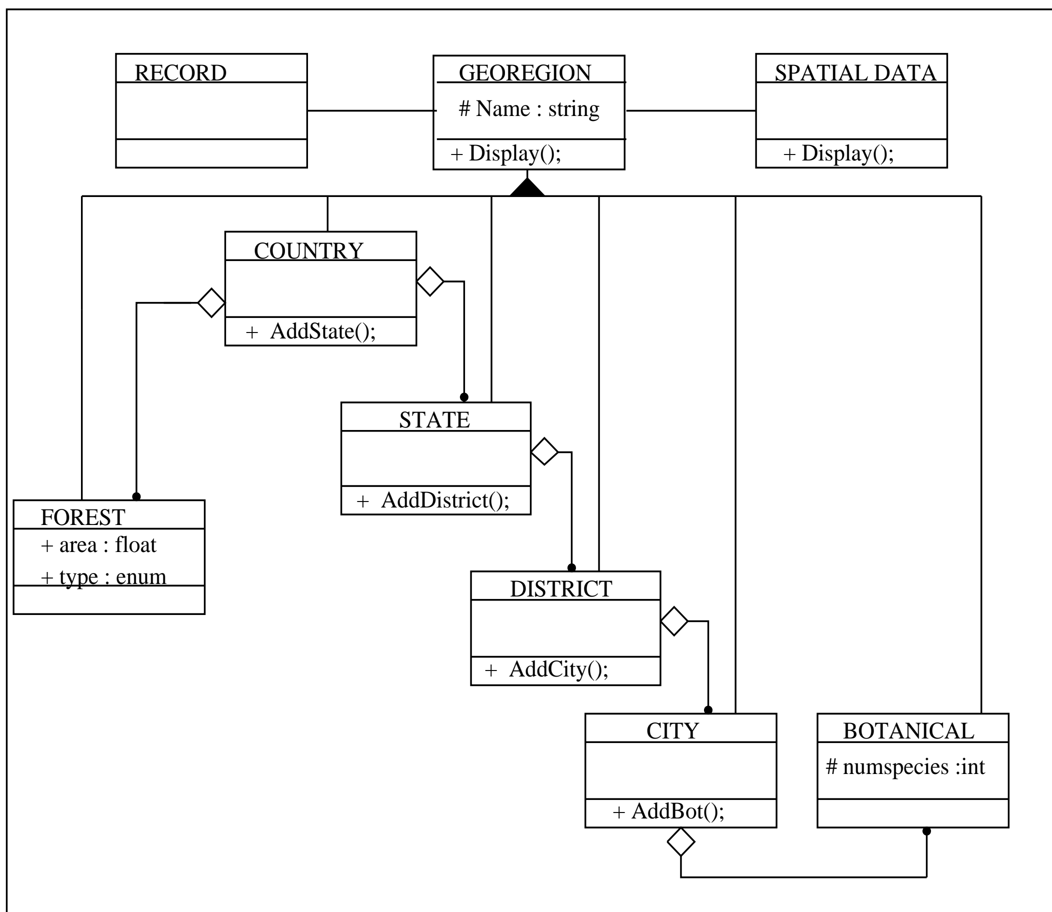


Figure 4.1: Object Model of the geo-spatial data observed in biological applications.

Primitive Data Types

For modeling single objects, three primitive data types viz., point, line, and polygon as suggested in [Güt94a] are offered. Figure 4.2 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).

Compound Data Types

For modeling spatially related collections of objects two compound data types, layer and network, are offered as suggested in [Güt94a]. Figure 4.3 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 Bodhi, layers can be used to represent thematic maps (for example, map of Western Ghats with all districts).
- 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 Bodhi 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 4.4. 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.

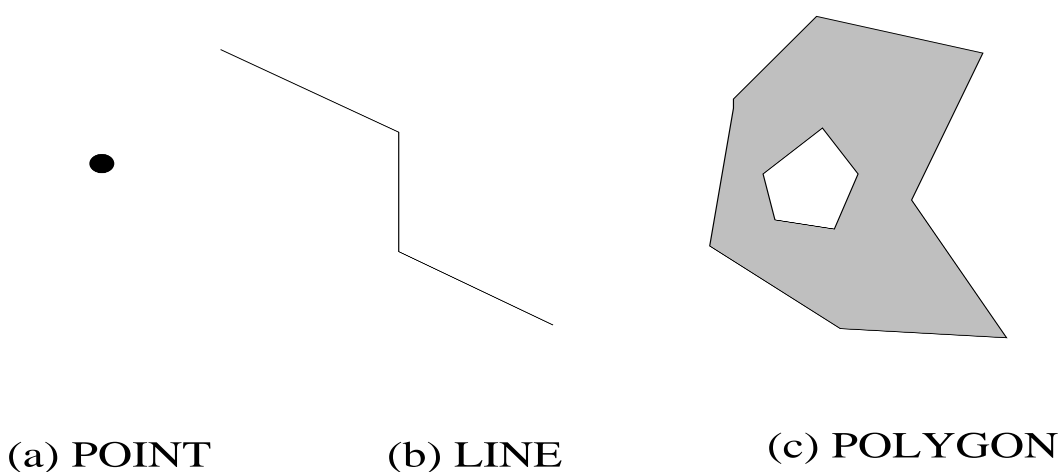


Figure 4.2: Three basic abstractions to represent single spatial objects

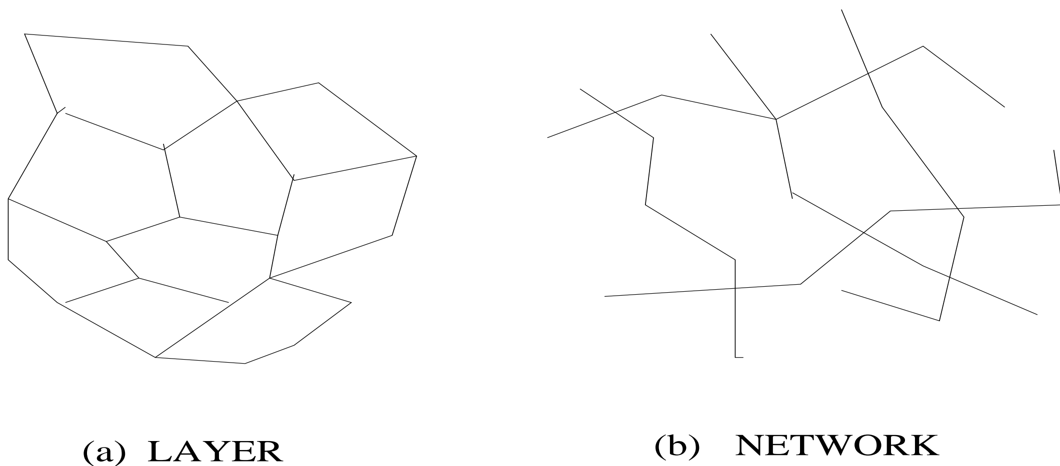


Figure 4.3: Two compound spatial data types to represent spatial collections

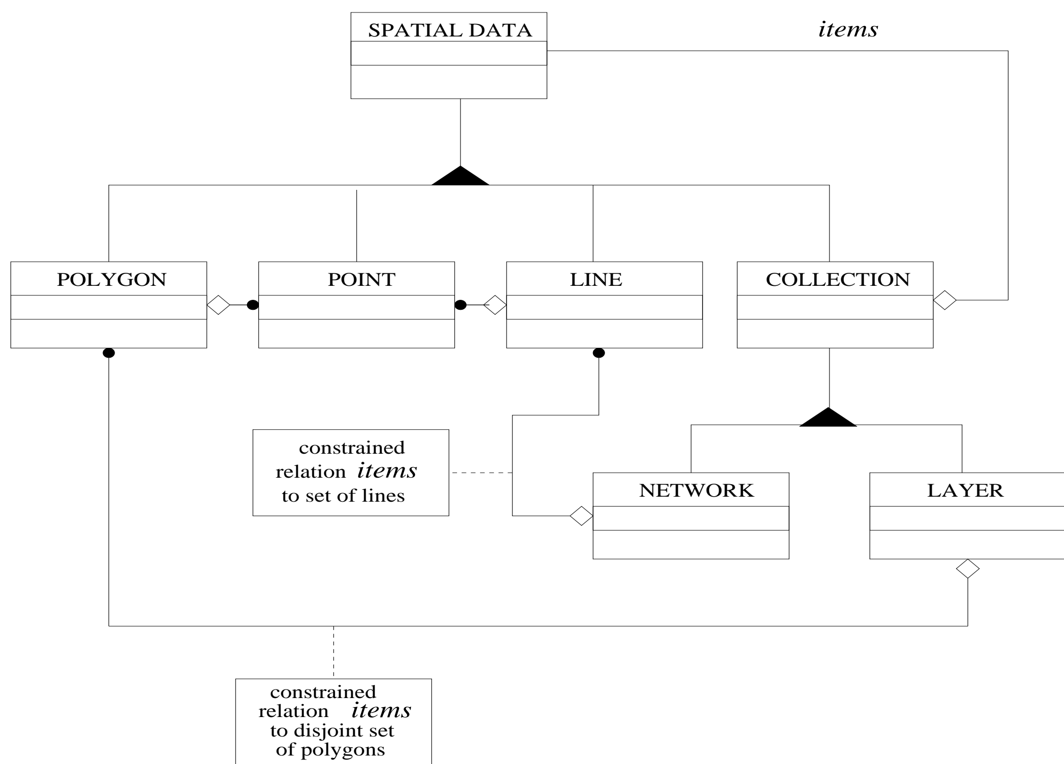


Figure 4.4: Object Model of Bodhi Spatial Data Types

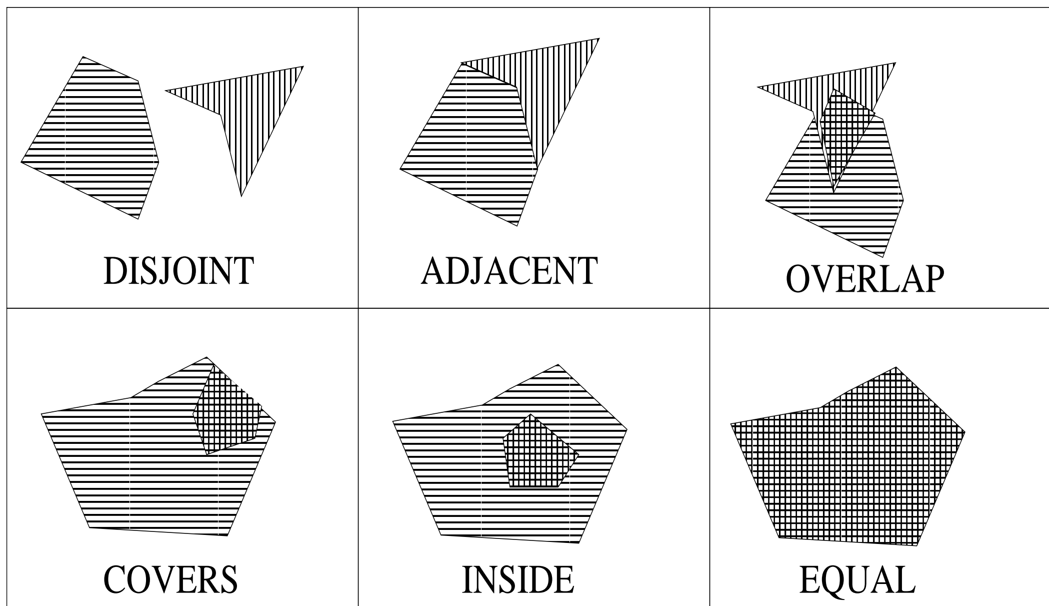


Figure 4.5: Spatial Relationships

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

Among the relationships mentioned above, according to [Güt94a], there are mainly six different relationships, called *disjoint, in, touch, equal, cover* and *overlap* are shown in Figure 4.5, which are important for geo-spatial applications. All these relationships are being supported in Bodhi.

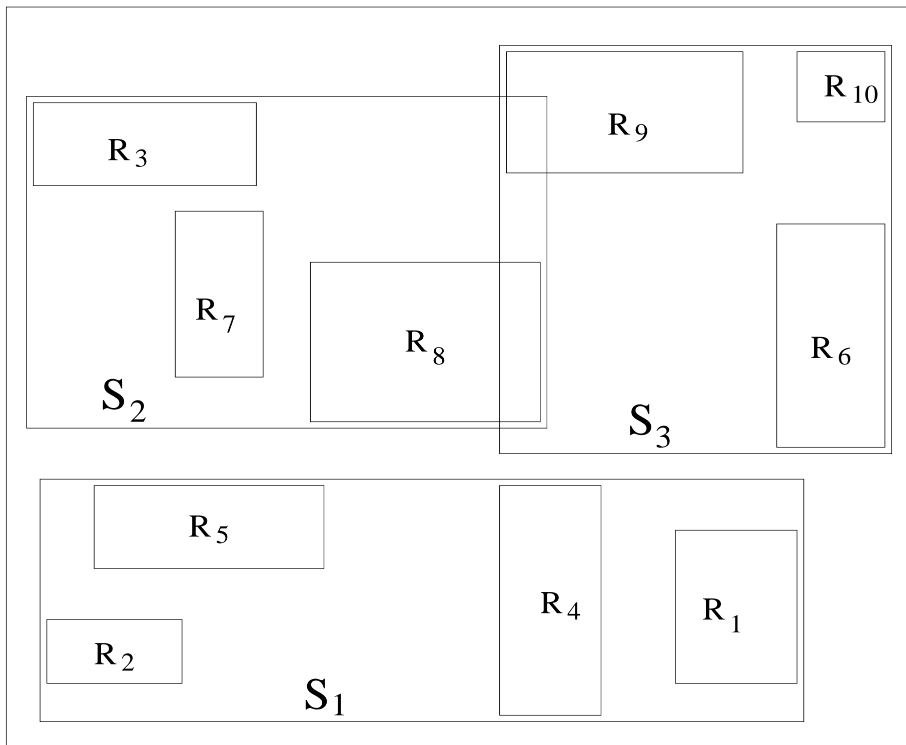
In this section we have seen the modeling of the spatial data in Bodhi. The support for spatial data by offering primitive and compound data types are discussed. The important relationships among these data types are also identified. This completes modeling phase and in the next section, we will look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

4.2 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 Bodhi system for handling spatial queries.

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.



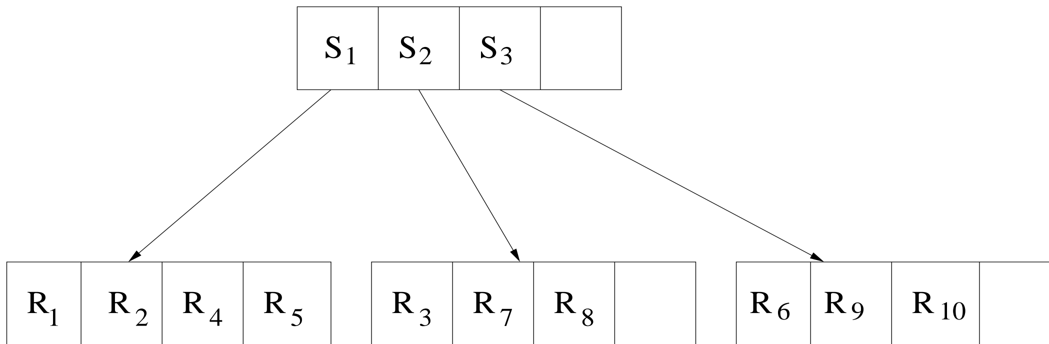


Figure 4.6: The file structure of R-tree.

4.2.1 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 4.6 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 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.

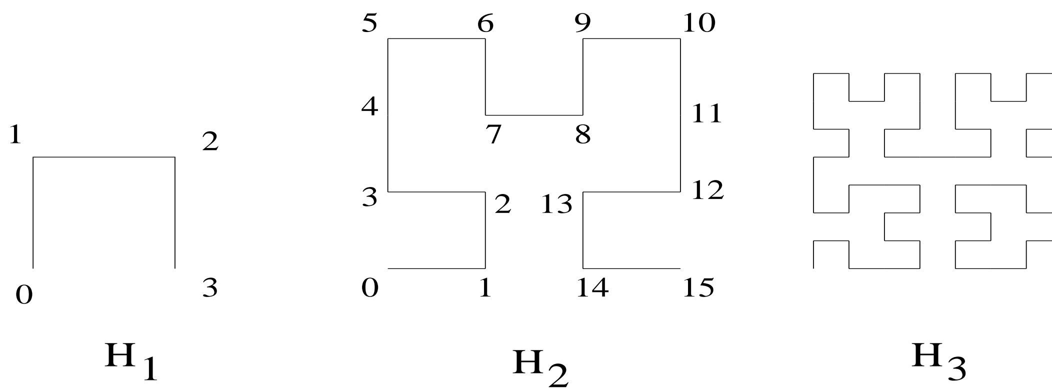


Figure 4.7: Hilbert Curves of order 1, 2 and 3.

- *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 [B^+90], Hilbert R-tree [KF94b], 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%.

4.2.2 Hilbert R-tree

Hilbert R-tree is a variant of R-tree proposed by [KF94b]. 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. [FR89b] 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 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%.
.

Hilbert Curve

The basic Hilbert curve on a 2 × 2 grid, denoted by H_1. is shown in Figure 4.7. 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 4.7 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 [KF94b]. The Hilbert curve can be generalized for higher dimensionalities. Algorithms to draw the two-dimensional curve of a given order, can be found in [Jag90b].

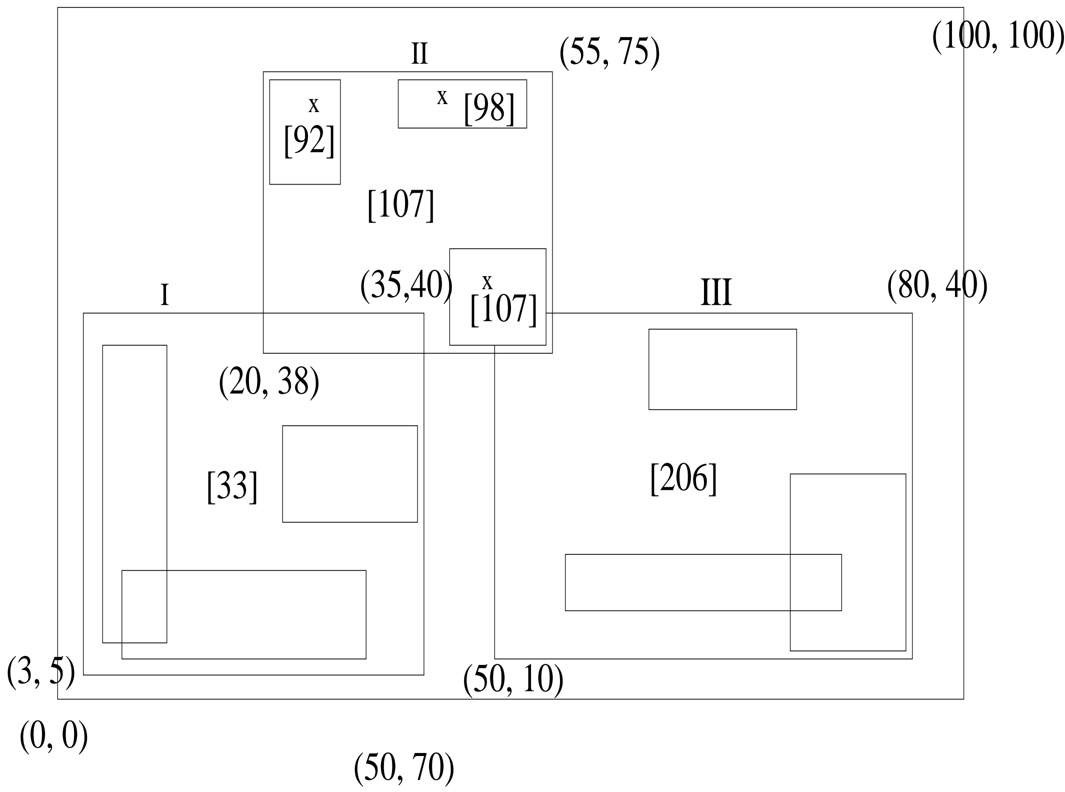
The path of a space filling curve imposes a linear ordering on the grid points. Figure 4.7 shows one such ordering for a 4 × 4 grid (see curve H_2). For example the point (0,0) on the H_2 curve has a Hilbert value of 0, while the point (1,1) has a Hilbert value of 2.

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
- 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.



LHV	XL	YL	XH	YH		LHV	XL	YL	XH	YH		LHV	XL	YL	XH	YH	
33	3	5	35	40		107	20	38	55	75		206	50	10	80	40	

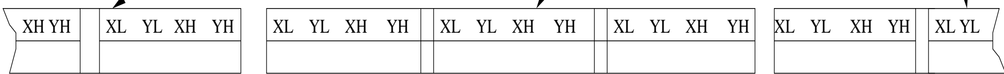


Figure 4.8: An example of the file structure of Hilbert R-tree.

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 4.8 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.

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.

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.

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)**.

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.

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 Bodhi.

This section finishes the design phase of the project and in the following sections 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.

4.3 Implementation Details

The implementation of the spatial module is discussed in this section. The object model discussed earlier in section 4.1 is implemented using Shore's support for persistent objects. The Hilbert R-tree index described in section 4.2 is then implemented inside the Shore storage manager (SSM). Bulk-loading and iterator facilities to the index are also provided.

4.3.1 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 D 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 Bodhi 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 [Kon00b]. 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 4.9.

4.3.2 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 [Kon00b].

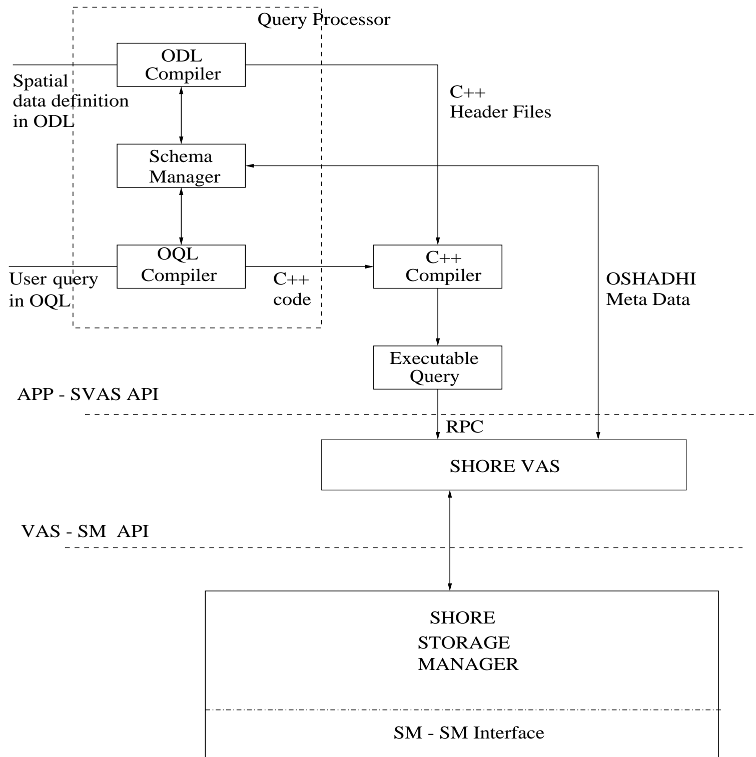


Figure 4.9: The data flow in Bodhi

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 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. The list of operations supported in Bodhi along with examples are discussed in Appendix E.

4.3.3 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 Bodhi 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 4.9, 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 4.9. Only modules inside the storage manager can access this interface.

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

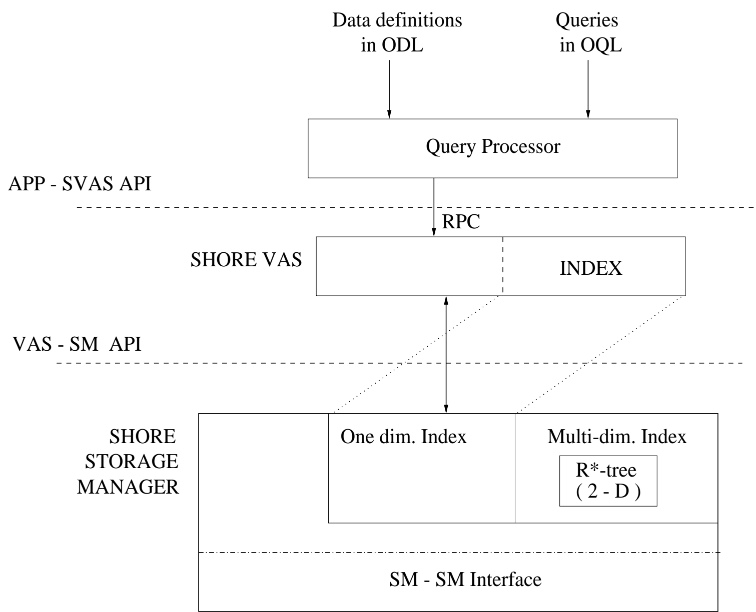


Figure 4.10: System Before Implementation

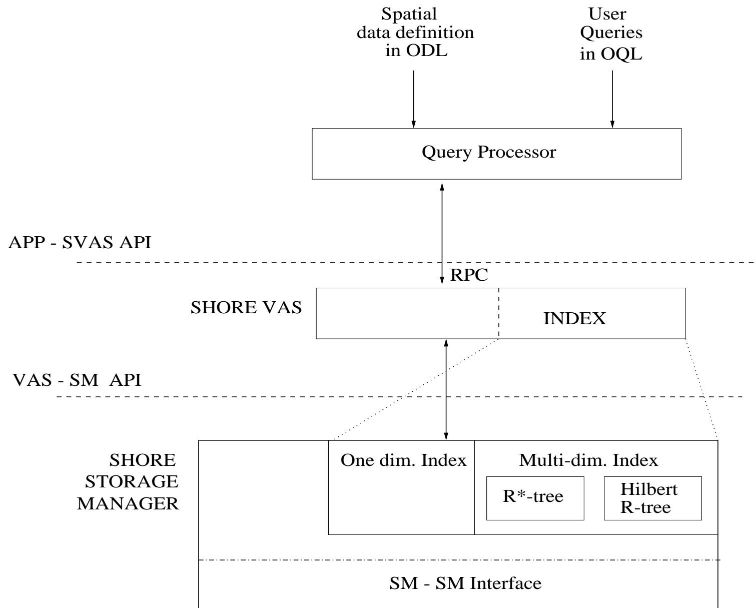


Figure 4.11: System After Implementation

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 4.10.

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 4.10. Since Bodhi 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.

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 4.10. 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 4.11. This required modifying the code of Shore VAS by adding some more functions to it.

In this section we have discussed the implementation details of the spatial data model and indexing mechanisms in Bodhi. We have also stated the choices and the decisions made regarding the level (layer) at which all these implementations were done.

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 Bodhi. Also the spatial data types can be extended to support images also.

4.4 Summary

Spatial data management is a major component in any biological information system. In this project, we have addressed how Bodhi, 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.

Chapter 5

Query Processing and Optimization

This chapter discusses the query processor and optimizer for Bodhi.

5.1 Design Issues

This Section discuss about issues in processing and optimization of queries in object-oriented databases and why special consideration is to be given to process queries on spatial data.

5.1.1 OODBMS Query Optimization Issues

The presence of user-defined types and user-defined functions in OODBMS make query processing and optimization difficult [DÖBS94]. The following are some of the issues in query processing and optimization in OODBMS.

Path Expression

Due to the presence of class aggregation, queries can be issued on any object deep in the aggregation hierarchy. So an user may need information of all objects which satisfies the predicate $C_1.a_1.a_2.....a_m = C_2.b_1.b_2......b_n$. This turns around to be a complex operation because each object in the class C_1 has to be fetched and its attribute a_1 has to be fetched and then from it, attribute a_2 and so on upto a_m. Same is the case with class $C2$. Path expressions are common in typical object-oriented applications, so the optimizer for an object-oriented query language has to optimize this sort of expressions [DÖBS94].

Derived Attributes

Derived attribute is any attribute whose value is dependent on some set of attributes within the class. While converting these the data model into a class definition there is always a decision to be made to whether these derived attributes should be functions returning a value or an attribute storing the value and there is a space/time trade off in doing it either way. An example is area of a spatial object. If it is an attribute usual optimizations will do, but if it is a function and it appears in the query then instead of fetching objects and then applying area function to the objects, we can have an index maintained on the values returned by this function and fetch the objects using the index. Method materialization and function-based indexing [Hwa94] are generalizations of this concept.

5.1.2 Spatial Query Optimization Issues

The characteristics of spatial data is that data can be present in a multi-dimensional space and needs special way to represent them and special operators like overlap, contain etc. to operate on them [Güt94a]. Some of the issues in spatial query processing/optimization are

- The basic problem in spatial data types of multiple dimensions is that there is no total ordering in the data space of the spatial objects which preserves the spatial neighborhood in the space. So the existing index structures like B+ trees cannot be used which expects a total ordering on the key value on which index is built. So, separate indexing mechanisms are used which take care of the spatial nearness (for more details see [Che00]) and the query processor has to make use of the presence of these spatial indices.

- Conventional join algorithms may not do better in the presence of spatial data because the join predicate will be an operation like overlap, contain etc. [Güt94a]. There are several spatial join algorithms which do join efficiently on a particular spatial access structure. So, the query processor has to make use of the presence of these spatial join algorithms.

5.2 The Bodhi Language Interface

This section discusses about the DDL and the query language for Bodhi. Examples are given to demonstrate the language features.

5.2.1 Data Definition Language

This section demonstrates the DDL of Bodhi with the help of an example which is used in describing the query language. The DDL of Bodhi is adopted from ODL (Object Definition Language), specified by ODMG (Object Data Management Group) [Cat93]. As ODL is fairly standard [Cat93], it is not explained in detail. Some of the modifications that are done to ODL are:

- In ODL the namespace is a set of modules and each module consists of a set of classes and there is no way in which one can refer one module from another. We have extended this concept to allow a module to be able to use any other module as it greatly will enhance the reuse of code.
- In ODL there is a concept of nested modules wherein the module definitions can be nested. Since, ODL declarations are converted into SDL declarations (see figure 5.2 for the flow of an ODL declaration) and as SDL does not support nested modules, this feature has been removed from ODL.
- ODL is extended to support the indexing to be done on derived attributes (more details in 5.5.2).

The following example (shown in Table 5.2.1) is written in DDL of Bodhi. Each of the class definition has an *extent* declaration which specifies the table where all the objects of that class are to be stored. The declaration *key name* states that attribute *name* is the primary key. The *import* statement states that all the classes defined in the imported module are accessible in this module. The *SpatialModule* is a module, containing the classes like point, polygon and polyline etc, whose definition is not shown here. Each attribute is declared by the keyword attribute followed by type and then the attribute name. The rest of the example is self-explanatory. The grammar for the DDL of Bodhi is given in the Appendix F.

5.2.2 Query Language

OQL is a declarative query language standard proposed by ODMG and has been widely accepted as a query language for OODBMS [Cat93]. This section gives a flavor of the query language designed for Bodhi, which is an extension of OQL.

OQL, by itself, is not computationally complete [Cat93]. However queries can invoke methods written in the host language and any host language methods can include OQL queries.

OQL supports only B+ tree index structures, which is not enough for Bodhi. So, OQL has been extended to support several index structures. The grammar for the query language of Bodhi is given in Appendix G. The following shows some example queries written in the query language adopted by Bodhi.

1. **Object Creation:**

   To create a persistent object, say of type City:

   po := City(location:p, belongsToState:s);

   where s and p are state and point objects created similarly. po is the newly created City object. All the indexes which are present on attributes of the class will be automatically updated by the query processor.

2. **Simple query:**

   To print the details of a plant say "neem" :

   ```
   FOR EACH i IN (
                   SELECT *
                   FROM p IN plants
                   WHERE p→name = "neem")
   DO i→Print();
   ```

3. **Query on a Class hierarchy:**

   To print all characteristics of a plant, say "neem", the query will be:

   ```
   FOR EACH i IN (
                   SELECT c
                   FROM p IN plants ,
                        c in p.specs
   ```

```
    module BioDiversity
    {
        import SpatialModule;
        class State (extent states key name){
            attribute string name;
            relationship City capital ;
            attribute polygon border;
              Print();
        };
        typedef enum LF_TYPE{BI_FOLIATE,TRI_FOLIATE,UNDEF_LF}LEAF_TYPE;
        typedef enum CLR {GREEN, YELLOW, UNDEF_CLR } COLOUR;
        typedef enum WBType { RIVER, LAKE, SEA} WATERBODYTYPES;
        class WaterBody (extent waterbodies) {
            attribute string name;
            attribute polyline shape;
            attribute WATERBODYTYPES type;
              Print();
        };
        class Forest( extent forests ) {
              attribute string name;
            attribute polygon boundary;
            attribute set <Plant>  plants;
              Print();
        };
        class City(extent cities) {
              relationship State belongsToState ;
            attribute point location;
              Print();
        };
        class Plant (extent plants) {
            attribute string name ;
            attribute set <State>  statesFound;
            attribute list <Characteristic> specs;
            attribute set<string> medicineFor;
            boolean isFoundinState(in string stateName);
             Print();
        };
        class Characteristic (extent characteristics) {
            attribute string chName;
              Print();
        };
        class Flower: Characteristic (extent flowers) {
            attribute COLOUR colour;
            Print();
        };
        class Leaf: Characteristic (extent leaves) {
            attribute LEAF_TYPE typ;
            Print();
        };
};
```

Table 5.1: **Example code illustrating DDL in Bodhi**

WHERE p.name = "neem"

)

DO i→Print();.

This will generate the code for Print method to be called on all the objects present in the extents characteristics, flowers and leaves which belongs to neem plant. This query also demonstrates the use of an attribute of collection type as the generator of data in the FROM clause.

4. **Query with a Join:**

   To find all lakes which flow through the city Bangalore, the query is:

   FOR EACH i IN (

   *SELECT **

   FROM c IN cities, w IN waterbodies

   WHERE c.name = "Bangalore"

   AND c.location.overlap(w.location)

   AND w.type = LAKE

)

   DO cout<< i →w→name<<endl;

5. **Index Creation:**

   To create an index, say Hilbert R-tree, on the boundary attribute (of polygon type) in the Forest class:

   CREATE HILBERT INDEX hilID1 ON Forest(boundary);

   where *hilID1* is the index identifier. This index identifier is to be used when deleting the index.

6. **Object updation:**

   The following examples demonstrate how to update the value of an attribute. Query processor will update all indices that need to be updated to maintain the consistency.

   (a) **Simple attribute assignment:**

   In this case, a simple attribute (of basic or compound types but is not a collection) is assigned a new value.

   To update the name of the city , say Bombay, from "Bombay" to "Mumbai". Assuming that *bombay* is the object holding the Bombay city object, the OQL statement which does the update is:

   bombay.name := "Mumbai";

   (b) **Collection Attribute Insertion:**

   In this case, a collection attribute is updated and an element is inserted into it.

   Suppose, Thulasi plant is found to cure the heart attack. Then we want to insert this "Heart Attack" into an attribute medicineFor, in class Plant, which is a collection. Assuming that *thulasi* is the variable containing the Thulasi plant object. The query will be:

   thulasi.medicineFor += "Heart Attack";.

   This $+=$ operator will insert an element into the attribute which is a collection.

   (c) **Collection Attribute Deletion:**

   In this case, a collection attribute is updated and an element is deleted from it.

   Similar to $+=$ operator to insert into an attribute of collection type, there is a $-=$ operator which will delete an element from the attribute which is of collection type. The query to remove an element from an attribute which is a collection type is:

   thulasi.medicineFor −= "Heart Attack";.

5.3 Query Processing and Optimizations in Lambda-DB

Bodhi's query processor and optimizer is built by extending Lambda-DB, a rule-based query processor and optimizer. This section describes the salient features of Lambda-DB.

Most of the material in this section is taken from [FM95], [Feg95], [Feg97b], [Feg98a] , [Feg98b] and Source Code Documentation.

5.3.1 Introduction

Lambda-DB works with OQL and ODL as its query language and DDL respectively. A query passes through several phases before the equivalent C++ code is generated for the query. This code after execution produces the results. Each of these phases are described in more detail in later parts of this section. An ODL declaration is converted into a SDL declaration. As ODL and SDL are more or less similar, the conversion is straight forward and the details are not discussed. Lambda-DB provides OPTL language [Feg97b] using which all the rules and actions, for the query processing and optimization, are written. OPTL is described briefly in Section 5.3.4.

Lambda-DB uses both a calculus and an algebra as intermediate forms because calculus closely resembles OQL and is easy to optimize by query-rewriting, while the algebra is low level and can be directly translated into the execution algorithms supported by database [Feg97b].

The next section discusses about the calculus of Lambda-DB, the monoid comprehension calculus, which is used as the intermediate form of query representation. The monoid algebra, similar to the nested-relational algebra, serves as an intermediate form between comprehensions and physical plans and the algebraic operators are discussed in Section 5.3.3.

5.3.2 The Monoid Comprehension Calculus

Lambda-DB uses monoid comprehension calculus, which is a generalization of list comprehension familiar in the functional languages, to represent the queries in an intermediate from [FM95].

A monoid can be primitive or a collection and is used to represent the types. All operations are represented as homomorphisms which maps a collection monoid into a collection or primitive monoid. Monoids are represented in the form of $\mathcal{M}$(t,z,u,m) where

- $\mathcal{M}$ is the monoid,
- t is the type of the result,
- z is the identity element of the monoid,
- u is the unit function which takes an element e and returns a monoid $\mathcal{M}$ of type t having only that element e and
- m is the merge function used in combining two monoids.

Examples of primitive monoids are sum(int, 0, unit(a)=a, +), some(bool, false, unit(a)=a, $\vee$). Example of collection monoids is list(list(T),[ ], $unit(a) = [a]$,list_append). Here list is a list of objects of type T and a is an element of type T . A monoid comprehension calculus is defined over a monoid $\mathcal{M}$(collection or primitive) and takes the form $\mathcal{M}\ \{e|q\}$, where e is the head(projection) of the expression and q is $q_0.q_1 \ldots .q_m$ where $m \geq 0$ and each $q_i$is either

- a generator of the form v $\longleftarrow$e where v is a variable and e is an expression or an extent name.
- a predicate on any of the variables defined to the left of this predicate.

5.3.3 The Monoid Algebra

This section discusses about the algebra used in Lambda-DB. The following are the algebraic operators used by lambda-DB and the conversion process from the calculus to these algebraic operators is discussed in the Section 5.3.8. An example demonstrating these algebraic operators is given in this Section.

1. **get(monoid, extent_name, range_variable, predicate):**

   This operator captures the OQL query: SELECT * FROM *range_variable* IN *extent_name* WHERE *predicate*. It creates a collection, where each tuple has only one component, *range_variable*, bound to an object of this *extent_name*. The *predicate* has the form $and(p_1, \ldots, p_n)$ to indicate the conjunction of the predicates p_i. The *predicate* is not allowed to contain nested queries (all forms of query nesting are eliminated before this stage).

2. **reduce(monoid, expr, variable, head, predicate):**

   This operator generalizes the projection operator. For each tuple in the collection *expr* that satisfies the *predicate*, it evaluates the *head*. Then it reduces the resulting collection to a value (bound to *variable*) or a collection (where each tuple binds *variable* to the value of *head*), depending on the output *monoid*.

3. **join(monoid, left, right, predicate, keep):**

   This is the relational join operator. It concatenates the tuples of the *left* and *right* collections if they satisfy the *predicate* and creates a new collection out of the new tuples. *keep* indicates which variables to keep from *left* (if any) in case of a left outerjoin (if no tuple is joined with the value of a *keep* variable, the latter is joined with null).

```
SELECT DISTINCT STRUCT( MyPlant: e.name,
                        Spec: ( SELECT c.name
                                FROM c IN e.specs
                                WHERE some_predicate(c)
                                ) )
       FROM e IN plants
       WHERE e.name = "neem";
```

Table 5.2: **An Example in OQL to Demonstrate Monoid Algebra**

```
REDUCE(SET,
    NEST(BAG,
        UNNEST(BAG,
            GET(SET,
                plants,
                e,
                AND(EQ(PROJECT(e,name),"neem"))
                ),
            c,
            PROJECT(e,specs),
            AND(some_predicate(c)),
            e
            ),
        x,
        PROJECT(c,name),
        e,
        AND()
        ),
    y,
    STRUCT(BIND(MyPlant,PROJECT(e,name)),BIND(Spec,x)),
    AND()
    )
```

Table 5.3: **An Example Query in the form of Monoid Algebra**

4. **unnest(monoid, expr, variable, path, predicate, keep):**

   It unnests the inner collection *path* of the outer collection *expr* and filters out all tuples that do not satisfy the *predicate*. For example, if *expr* is of type set(<p:Plant>) and *path*=p.specs, then the output of this operation has type set(<p: Plant, new_variable: Characteristic>). *keep* plays the same role as for joins.

5. **nest(monoid, expr, variable, head, groupby, predicate):**

   It groups the *expr* by the variables in *groupby*. Then, for each different binding of the *groupby* variables, it creates a tuple extended with the binding of *variable* to the value reduce(monoid, plan, variable, head, predicate).

6. **merge(monoid, left, right):** Merges the (union-compatible) input expressions.

Let us consider an example to demonstrate how an OQL query in the algebraic format looks like. The query is, say, to display all the characteristics of neem plant which satisfies the predicate some_predicate. Then the corresponding OQL query is shown in the Table 5.2 and the corresponding algebraic expression is shown in Table 5.3. x and y are temporary variables generated by query processor and are used as temporary data holders.

Each of the algebraic operations present in Table 5.3 yields values of the following types:
get → set(< e: Plant >)
unnest → bag(< e: Plant, c: Characteristic >)
nest → bag(< e: Plant, x: bag(string) >)
reduce → set(< MyPlant: string, Spec: bag(string) >)
Notice that

1. The unnest operation has keep=e, i.e., it keeps every plant even if a plant does not satisfy the some_predicate(). In that case, c is bound to null. This null value is converted to the empty bag (the zero of the bag monoid) during the nest operation.

2. The operators does not have selection as an operator because every operator can take predicate as an argument.

```
Expr  x = #<join(a,b,p)> ;
Expr  y = #<select(‘x,q)>;
list< Expr > *r = Cons(#<‘x,Cons(#<select(c,q)>,Nil));
// Here Cons is a list append operator.
Expr  z = #<join(...r,pred)>;


then y is set to #<select(join(a,b,p),q)>;
and z is bound to #<join(join(a,b,p),select(c,q),pred)>;
```

Table 5.4: **Examples of OPTL Expressions**

```
Expr  switch_join_args( Expr  exp)
{
#case exp
|‘f(...r, join(‘x, ‘y), ...s) : x → eq(y) ⇒ return # < ‘f(...r, ‘y, , ...s) >;
|‘f(...r, join(‘x, ‘y), ...s) ⇒ return # < ‘f(...r, join(‘y, ‘x), ...s) >;
|‘x ⇒ return x;
#end;
};
```

Table 5.5: **Example of OPTL Case Statement**

3. Also a generalization of the projection operator, called reduce, is over a collection and at the same time it maps the collection to a (possibly different) monoid (such as aggregation or a quantification monoid). This is the way to do aggregation or quantification in the algebra.

5.3.4 OPTL

This section briefly discusses the OPTL language which is a pattern based language used for query rewriting. Most of the code of Lambda-DB and Bodhi's query processor and optimizer is written in OPTL. OPTL is an embedded C++ language in which we represent the rules in terms of patterns in the query and actions in C++/OPTL. OPTGEN is a C++ preprocessor that maps OPTL specification into executable code. So, all OPTL code is processed by OPTGEN which converts this OPTL expressions into C++ Code.

In OPTL, Expr is the basic data type which is a tree-like data structure used in representing both plans and algebraic forms. To ease the process of writing the code in OPTL, it has extended C++ with the syntactic form #<>. For example, #<join(R,S,eq(A,B))> captures the algebraic term, in tree form, which is join(R,S,eq(A,B)), which represents equijoin $R \bowtie_{A=B} S$.

Term:

Expr is a tree of Terms. Each term in the Expr is either

- string denoting the data or
- Expr represented by escaping with a backquote character(‘) or
- List of Expr represented by escaping with a dot-dot-dot (...).

See the Table 5.4 for an example demonstrating the OPTL expressions.

Case Statement:

OPTL provides a case statement for pattern matching. Case statement has an expression *exp* and a list of options as templates which are probable candidates to match with the expression *exp*. The options are also of type Expr and can be escaped with backquote($'$) or ... as mentioned before. if a variable is escaped in the option then that variable is bound from an appropriate term from the expression *exp* (like 'x will be matched to a term and $\dots r$ will be matched to a list of terms.

As an example of case statement in OPTL, consider the function shown in Table 5.5.

Here, if it exp is any expression having a function with one of the arguments being a join expression, then it swaps the arguments of that join expression iff the arguments are not same. Here f is bound to the function name, r and s is bound to the list of arguments to the left and right of the argument having the join expression. Options expressed can have gaurds, i.e., after pattern matching, the action is executed only if the gaurd is satisfied. Gaurd can be any C++ predicate. In the Example shown in Table 5.5, the predicate $x \rightarrow eq(y)$ is the guard and this does the job of removing the join expression if there is a join of same tables. (Note that the order of the statements are important. There can be expressions which can match so many template options in the case statements, then in that case only the first one's action is executed.)

The following shows several inputs and the outputs of the function swith_join_args so that its behaviour can be understood.

$\# < myFunc(a, b, join(c, d), e, f, g >$ will return $\# < myFunc(a.b.join(d, c), e, f, g >$

Here f will be bound to myFunc, r will be bound to list(a,b), s will be bound to list(e,f,g), x will be bound to c and y to d.

$\# < myFunc(a, b, join(c, c), e, f, g >$ will return $\# < myFunc(a.b.d, e, f, g >$

$\# < myFunc(a, b, c, d, e, f, g >$ or $\# < a, b, c, d >$ or any other which does not match the first two patterns return the same expression.

For more details on OPTL language one can refer [Feg97b].

5.3.5 Phases of Lambda-DB Optimizer

This Section gives a brief introduction to the different phases that are present in query processing/optimization in Lambda-DB and the subsequent sections discusses each of these phase in detail.

1. **Parsing and type checking:** During this phase, query in OQL is translated into the comprehension calculus and it is checked for type errors.
2. **Normalization:** During this phase, some forms of nested queries are unnested using the *normalization* algorithm described [Feg97b]. During *normalization* , the generator domains are reduced into simple paths.
3. **Translation into an algebraic form and query unnesting:** After comprehensions are normalized they are translated into nested-relational operators which are extended to include multiple collection types and aggregation.(These operators are described in detail in Section 5.3.3). At the same time, nested comprehensions are unnested, by converting joins into outer-joins and by grouping together the results.
4. **Materialization of path expressions:** During this phase, if a query contains a path then it is materialized.
5. **Operator Ordering:** During this phase, the best join processing order is chosen.
6. **Physical plan generation:** The last phase considers the available access paths (indices) of each extent and the available alternative physical algorithms to generate different plans. Finally, the best plan is selected and code is generated for that.

The following sections discusses each of these phases in detail.

5.3.6 Parsing And TypeChecking

In this phase a query in OQL is translated into a comprehension calculus and it is checked for type errors. This process is straightforward and it is described in detail in [FM95]. A select-from-where OQL statement is of the form

SELECT e
FROM x1 IN e1 , ... ,xn IN en
WHERE pred;

and is translated into

$BAG\{e|x1 \leftarrow e1, \ldots, xn \leftarrow en, pred\}$

in the monoid comprehensions form.

Suppose the query is to **find all plants in the forest Mudhumalai which can be useful in treating Asthma**, Then the OQL query is

```
SELECT p.name
        FROM p1 IN ( SELECT f.plants
                        FROM f in forests
                        WHERE f.name ="Mudhumalai"
                        ),
        p in p1,
        WHERE EXISTS r IN p.medicineFor : r = "Asthma";
```

This query is expressed in comprehension calculus as follows:

$$BAG\{p.name|p1 \leftarrow BAG\{f.plants|f \leftarrow forests, f.name = "Mudhumalai"\},$$
$$p \leftarrow p1, some\{r = "Asthma"|r \leftarrow p.medicineFor\}\}$$

5.3.7 Normalization

Another claim for monoid calculus is that it supports easy manipulation of query expressions which is amenable to pattern-based rewriting [FM95]. This monoid calculus expressions can be put into a canonical form by an efficient rewrite algorithm called the *normalization* algorithm. (The *normalization* algorithm is given in [FM95], the algorithm in the form of OPTL rules is given in [Feg97b] and its proof of correctness is found in [Feg94]). It generalizes many optimization techniques, such

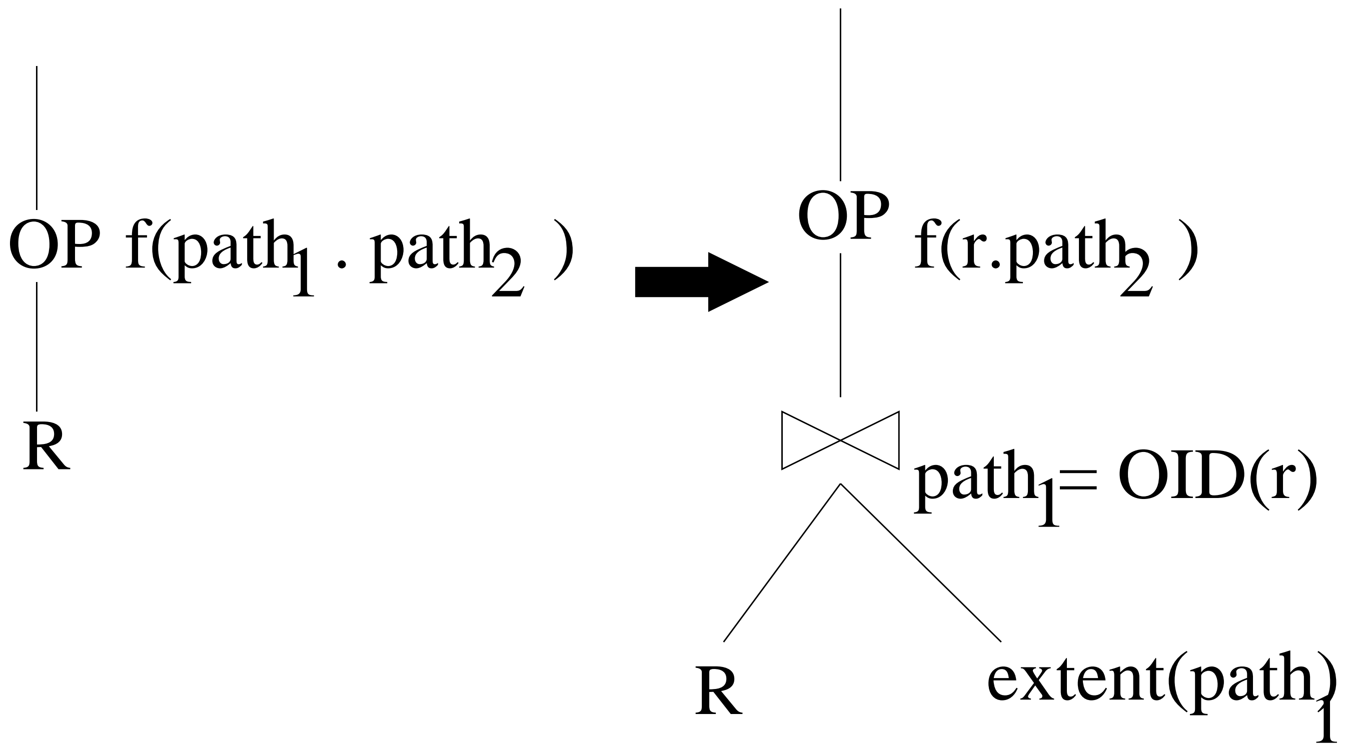


Figure 5.1: **Path Materialization**

as pushing a selection before a join. The evaluation of these generally produces fewer intermediate data structures than the initial unnormalized programs.

For the query given in Section 5.3.6, that finds all plants in the forest Mudhumalai which can be useful in treating Asthma, the OQL query and the calculus form is shown in the Section 5.3.6. The normalized form of that query is:

```
BAG{p.name |f<-forests,f.name="Mudhumalai",
           p<-f.plants, r<-p.medicineFor,r.name="Asthma"}
```

Thus after normalizing the expression we will get a query that do not materialize the intermediate results.

As an example showing predicate pushdown, The following transformation, which is valid for any monoid $\mathcal{M}$, pushes a selection before a join if predicate does not depend on v:

$\mathcal{M}\{$ e | q,v$\longleftarrow$e,p,r$\} \longrightarrow \mathcal{M}\{$ e | q,p,v$\longleftarrow$e,r$\}$

5.3.8 Translation into an algebraic form and Unnesting

The next stage after normalizing is the translation of the comprehensions into the algebraic operators. The generators of a comprehension are translated from left to right, generating either a join (if the domain is an extent) or an unnest (if the domain is a path). Refer to Section 5.3.3 for an example demonstrating the conversion of an OQL query into algebra. If the query has any nested queries then they are unnested in this phase. (For more details on unnesting the query one can refer to [Feg98b]).

5.3.9 Materialization of Path Expressions

One of the most common technique for OODBMS is converting path expressions into joins [Feg97b, BKG93]. It is also known as materialization of path expressions [CD92].

The Figure 5.1 shows how a path expression is converted into a join. The Figure 5.1 indicates that if op is an unary operator (such as unnest) that depends on path expression $path_1.path_2$, where $path_1$ and $path_2$ are paths and R is any input stream to the operator op, f is any predicate then this can be converted into a join of the first extent with that of the input stream R. The resultant is shown in the right part of the Figure 5.1 The detailed algorithms and rules in OPTL for this transformation is given in [Feg97b].

5.3.10 Operator Ordering

This phase re-orders the order of the join processing so that the cost of join is minimum. Because of a number of OODB optimizations implemented in Lambda-DB, such as translation of path expressions into joins and query unnesting, it may generate a large number of implicit joins even for simple queries. So an efficient join order processing algorithm has gained importance in this context [Feg97b]. The join processing ordering algorithm used in Lambda-DB is a polynomial time (of order $\mathcal{O}(n^3)$ where n is the number of relations) greedy algorithm called as GOO(Greedy Operator Ordering).

Greedy Operator Ordering Algorithm

The GOO algorithm is a bottom-up algorithm, similar to Kruskal's minimum spanning tree algorithm. The main data structure is the query graph where each relation r_i corresponds to a node i and each join predicate P_{ij} corresponds to an edge between i and j. Each node i is labelled by the size of r_i and each edge between i and j is labelled by the selectivity S_{ij} (the selectivity of predicate P_{ij}). At each step of the algorithm, it selects two nodes i and j that have minimum value of size $(r_i)\times$size$(r_j)\times S_{ij}$. Then these two nodes are merged into a new node ij. The size of node ij is size$(r_i)\times$size$(r_j)\times S_{ij}$ and its operator tree is $r_i \bowtie_{P_{ij}} r_j$. In addition for each node k connected to both i and j, the weight of the edge between the nodes k and ij becomes $S_{ik} \times S_{jk}$. For a full description of the algorithm see [Feg98a]. The GOO algorithm is not optimal always as it is a greedy algorithm based on heuristics. Refer to [Feg97b] for an example for which GOO is sub-optimal.

Ordering OODB Operations

This GOO algorithm mentioned above is extended to order some of the OODB operations like nest, unnest which commute with join. For this, the nodes of the query graph are range variables in the query,associated not only with extents, but with nest and unnest operators as well. A dependency between two range variables is denoted by an edge in the query graph. This algorithm groups two nodes if these do not depend on any other node (but may depend on each other). If one node depends on the other then appropriate OODB operator (nest or unnest) is generated as the operator of the node; otherwise a join is generated as the operator of the node.

5.3.11 Physical Plan Generation

This phase is concerned with the generation of physical plans corresponding to the algebraic operators present when the query comes to this stage.

The following are the physical plan operators used by lambda-DB:

1. **TABLE_SCAN(monoid, extent_name, range_variable, predicate):**

   This operator implements the OQL query: SELECT * FROM *range_variable* IN *extent_name* WHERE *predicate*. It creates a stream of tuples, where each tuple has only one component, *range_variable*, whose value is an object from the extent *extent_name*.

2. **INDEX_SCAN(monoid, extent_name, range_variable, predicate, index_name, low, high):**

   This operator is same as TABLE_SCAN, but it uses the index *index_name* to deliver all the tuples between (and including) the keys *low* and *high*.

3. **REDUCE(monoid, plan, variable, head, predicate):**

   For each tuple of the input stream of *plan* that satisfies *predicate*, it evaluates the *head*. Then it reduces the entire stream to a value (a stream of one value bound to *variable*) or a collection (a stream of tuples, where each tuple binds *variable* to the value of *head*), depending on the output *monoid*.

4. **NESTED_LOOP(monoid, left_plan, right_plan, predicate, keep):**

   Implements the relational join operator. It concatenates the tuples in the input streams of the *left_plan* and *right_plan* if they satisfy the *predicate*. If *keep*=none, the join is a regular join. Otherwise, it is a generalization of the left outer join and *keep* specifies which variables to keep from the left stream. For example, if *keep* = pair(e,d), then if all the tuples from the left stream that have names equal to e and d are not joined with any tuple from the right stream,

then this operator delivers one of these tuples concatenated with null values. If *keep* contains all the variables of the left stream, then this outer join is the same as the left-outer join in relational databases.

5. **INDEXED_LOOP(monoid, left_plan, right_plan, predicate, keep, index_name, key):**

   This operator is same as NESTED_LOOP but it uses indexing to retrieve tuples from the right (inner) stream. *index_name* is the name of the index while *key* contains the index attributes.

6. **MERGE_JOIN(monoid, left_plan, right_plan, predicate, keep, left_key, left_sort_order, right_sort_order):**

   This is same as as NESTED_LOOP, but it requires the *left_plan* be sorted by *left_sort_order* and the *right_plan* by *right_sort_order*. It merges the two streams using the sort orders. It requires that at least one of the streams has the sort order as a key (determined by the boolean value *left_key*).

7. **SORT(plan, sort_order):**

   Sorts a *plan* by the *sort_order*.

8. **UNNEST(monoid, plan, variable, path, predicate, keep):**

   For each tuple in the the input stream of *plan*, it concatenates the binding of variable to all possible values of *path*, filtering out those tuples that do not satisfy the *predicate*. If there are no values, or no value satisfies the *predicate*, and *keep* is not equal to none, then the tuple is padded with a null value.

9. **NEST(monoid, plan, variable, head, groupby, predicate):**

   It groups the input stream of *plan* by the variables in *groupby*. Then, it extents the input stream of *plan* with the binding of *variable* to the value REDUCE(monoid, plan, variable, head, predicate).

10. **MAP(monoid, plan, variable, pred, keep, path):**

    It maps the *path* to every tuple of the input stream *plan* and concatenates the result to the tuple.

11. **MERGE(monoid, left_plan, right_plan):**

    Merges the (union-compatible) streams of *left_plan* and *right_plan*.

After the query is rewritten in terms of these physical operators, C++ code is generated for the query in terms of these physical operators. The code generated is such that the data is pipelined between the operators rather than storing the intermediate results in a temporary extent.

5.4 Query Processing

This section discusses about the query processing in Bodhi in detail. This focuses on the data flow in Bodhi regarding the processing of ODL declaration before generating the C++ header file and how an OQL query passes through several phases before getting executed. This section also discusses the issues in interactions with Visualization and the implementation details and how extensibility (supporting of different formats) is possible and then the schema manager is discussed and the format that is used in exporting the meta-data to the Visualization is presented.

5.4.1 Data Flow in Bodhi

Figure 5.2 shows different components of query processor and the flow of a query, in Bodhi, starting from the user to the displaying of results.

ODL Declaration:

Since all objects are to be stored in SHORE, we have to use SDL as the DDL at some level. An ODL declaration is therefore parsed and converted into an equivalent SDL declaration and the SDL compiler generates an equivalent C++ binding for the SDL declarations. While Converting ODL Declarations to SDL these declarations are stored by the Schema manager for further use in optimization. ODL compiler also generates some C++ classes, in addition to SDL declarations. ODL compiler generates one class for each class and module. These are called as ClassController classes and ModuleController classes. The main use of these Controller classes is given in Section

Name	Type	ClassName	Statistics	DeclarationExpression
specs	attrib-dcl	State		attribute(specs,list(characteristics))
states	extent-dcl	State	50	extent(states)

Table 5.6: **Meta-data Storage Format**

```
Plant  (    ATTRIBUTE(name,STRING),
            ATTRIBUTE(specs,LIST(Characteristic)),
            METHOD(isFoundinState,BOOLEAN,
                    PARAMS(in (STRING,stateName))
                   )
      )
```

Table 5.7: **Meta-data Export Format Example**

5.4.3. But the use of this Controller classes can be extended to a greater extent. For example to initialize all the attributes of the objects with default values , to handle the exceptions raised by the C++ member functions etc. This Controller classes design is actually a restrictive case of MVC(Model-View Controller), one of the mostly used design pattern in software architecture, where an object will be controlling a set of objects (for more details on what a design pattern is and on MVC pattern see [GHJV95b]).

OQL Query: An OQL query is received from the Visualization and is first parsed for correctness and then optimized then the physical plans were generated and the C++ code is produced for the query in terms of these physical plans. The C++ code is linked with the Bodhi Library, C++ header file generated from ODL declarations and an executable is generated. This executable has code to interact with SHORE, Taxonomy and Visualization servers and display the results.

5.4.2 Schema Manager

Schema Manager is responsible for the maintenance of all the meta-data in Bodhi. All optimizations and type checking depends on the meta-data stored by this Schema Manager.

Storage Format

The storage structure of meta-data is as shown in the Table 5.6. The *Name* field stores the name of the declaration (like attribute name, function name etc.), *Type* is the type of the declaration (like index-dcl, attrib-dcl, method-dcl, extent-dcl etc.), *ClassName* is the name of the class to which this declaration belongs , *Statistics* stores the statistics (like the size, cardinality etc) and *DeclarationExpression* contains the expression of the declaration in a string form as it may contain more properties which varies among the declarations (like method can have a list of parameter types and path dictionary have a path expression to be stored).

Meta-data Export Format

The optimizer and the Visualization module needs to access the meta-data for their processing. Optimizer may need, say, information about the presence of indices to use in it's optimization. Similarly Visualization needs the type information in forming the query. So, Schema Manager exports the meta-data in the format whose grammar, in BNF, is shown in Appendix H. An example showing this export format for the class Plant shown in Table 5.2.1 is shown in the Table 5.7

5.4.3 Interface With Visualization

Bodhi consists of a web-enabled front end. The query processor forms the back-end which computes the query and sends the results back to the Visualization. Visualization module comprises of a GUI-server and a GUI-client (for more details on Visualization in Bodhi see [Gho00]).

Issues in Interface with Visualization

The query language adopted for Bodhi can be used only in fetching the objects satisfying the query and does not have any features for displaying the object. User has to write the C++ code in the action part of the query to display the objects. But, since the user can write the code in his own way, a question arises regarding how the Visualization module interprets these results. GUI-Server needs the type of the results because each type may need some special treatment for displaying them(For example, output of a query can be a set of numbers which can be representing the points of a polygon or salaries of employees and if the type is polygon then Visualization has to render the points and display the polygon. But if it is employees salary it can just print them). So, the results must have meta-data also describing the structure of the output. But an issue is who will generate the meta-data. There are two solutions to this:

1. The user writes the code for displaying the object's attribute values and the meta-data in the action part of the query in a format acceptable by GUI-server.
2. Bodhi will provide a function which does all this chores of displaying the data and meta-data and the user can just call this with the object, that is to be displayed, as parameter.

In Bodhi we have followed second approach by providing a function **BodhiDisplay(object)** which traverses through the object recursively and prints all the attributes of the object including the meta-data. So Users can use this function if necessary.

Having decided that Bodhi provides a function which displays the data and meta-data by traversing the object recursively, some more issues arise which are as follows:

1. An issue is how does the query processor traverses recursively. The main problem is how to get the attribute value at runtime given an object and attribute name. Some possible solutions to this are
   (a) While generating code for the query, generate code for functions which returns attributes given an object. But this has some disadvantages:
      i. So much of effort to generate the code from the query and this will increase the code generation time.
      ii. This code bloating will increase the compile time also.
   (b) Generate some standard functions, for each class, which returns the attributes given an object of that class type.
2. One more issue is how to pass the data from the query processor to the Visualization. There are two possible ways of doing this:
   (a) Materialize all the results in a particular format in a file and pass them to the GUI-server. GUI-server will understand the format and convert them into the format needed by the GUI-client for displaying them accordingly. But one disadvantage is that data has to be materialized in a file and it takes more disk accesses.
   (b) Another solution is pass the attributes to the GUI-server as and by when traversing the object recursively.
3. One more issue arises which is how to display the meta-data and data and in which format?. A question arises whether there is a unique way of displaying the meta-data for all applications?. The answer is unfortunately no. Each application may need a different type of displaying the data and the meta-data. So, the query processor has to be independent of the displaying format. But if results are materialized while passing the results from query processor to the GUI-server then this problem does not arise because there can be a protocol between the GUI-server and query processor and GUI-server can then convert this into the format necessary to the GUI-client and the flexibility of having more than one displaying format is possible by adding one more converter in the GUI-server. If results were not materialized but were pipelined from the query processor to the GUI-server then one possible solution is to define an interface using which the object's values are passed to the GUI-server and GUI-server sends the data and meta-data according to the needs of the GUI-client. Extensibility to have many output formats can be obtained by having different implementations of this interface.

```
class Plant_ClassController: public ClassController
{
    public:
    virtual GenericType GetAttrib(Ref<Plant> obj, char * attrName)
    {
    if(!strcmp(attrName,"name")) return obj->name;
    if(!strcmp(attrName,"specs")) return obj->specs;
    ...
    ...
    ...

    }
};
```

Table 5.8: **ClassController Class having functions to help in recursive traversal of objects**

```
class Biodiversity_ModuleController :public ModuleController
{
    public:
    virtual ClassController* GetControl( char * className)
    {
    if(!strcmp(className,"Plant")) return new Plant_ClassController();
    if(!strcmp(className,"State")) return new State_ClassController();
    ...
    ...
    ...

    };
};
```

Table 5.9: **ModuleController Class having functions for returning ClassController objects**

Implementation Details

In Bodhi these are the following solutions(decisions) provided(made) for the above mentioned issues.

- For the problem of traversing the objects recursively we have a function generated in the ClassController class, which given an attribute name and the object returns the attribute value. The sample code of such a function is shown in Table 5.8

  Having decided that ClassController class will have functions for returning the attributes, one more problem arises regarding how to get the ClassController object at runtime. A solution is that a class will be generated for each module which given the class name of any class in that module returns the ClassController object of that class, using which we can get the attribute value.

  Table 5.9 shows the ModuleController class generated for the example shown in Table 5.2.1

- For the problem of materializing the data, Bodhi does not materialize the results. Instead the results were pipelined to the GUI-server and it displays the data and the meta-data.

- For the problem of meta-data displaying Bodhi has defined an interface, called as Formatter class, to which data will be passed as and by when attributes are traversed and GUI-server implements the interface to produce the results in the way the GUI-client needs. This leads to the independence of the query processor to the format of displaying the results. Any support for new format has be done by the GUI-server. For example, if GUI-server decides to send all its output as Latex file then a class is defined which implements the Formatter class and this object is passed to the query processor. Table 5.10 demonstrates how a typical query will be to use a new formatter in displaying the results.

```
//file query.oql

    class LatexFormatter: public Formatter
    {
    //This implements the interface defined by
    //the Formatter class.
    ...
    ...
    ...
    };

    main()
    {
    SetFormatterObject(new LatexFormatter());
    ...
    ...
    %for  each i in (Valid_OQL_Query) do BodhiDisplay(i);
    ...
    }
```

Table 5.10: **An example demonstrating the extensibility of the formatters of results**

Here all the objects which satisfy Valid_OQL_Query are passed to BodhiDisplay function.

The interactions between the query processor and the formatters is shown in the Figure 5.3. It can be observed that the architecture of interactions, for formatting the results, between the query processor and the Visualization is nothing but a Builder pattern [GHJV95b]. There will be one ModuleController class for each module and whenever query processor changes to a module its moduleControllerObject is set to the current module's ModuleController object. There are several formatterObjects possible, one for each type of the formatting. Using the SetFormatterObject supplied by the query processor, as shown in Table 5.10, one can pass the object to the query processor which is to be used for formatting.

A high level algorithmic description of the function BodhiDisplay, which traverses recursively and displays all the attributes, is shown in Table 5.11. This also explains how the formatters interacts with the query processor in displaying the results. Actually the code is written in OPTL. The functions ItemStart, ItemValue and ItemFinish, in Formatter class, displays the delimiters of the object and the object itself. The functions Init and Finish displays the delimiters of the whole query if needed.

To demonstrate the principle an XMLFormatter is implemented. The tags used in this implementation are:
$< STRING >$, $< LONG >$ and $< REAL >$ to represent the basic types
and $< ERROR >$ to display an error message
and $< BODHIDISPLAY >$ to denote the start of the results
and $< CLASSTYPE = "classname" NAME = "attributeName" >$ to represent an object
and $< COLLETIONTYPE = "className" NAME = "attributeName" >$ to represent the attributes which are of type collections.

5.5 Query Optimizations in Bodhi

This section discusses the query optimizations that are done in Bodhi. First, optimization of path expressions using path dictionary is discussed and then the indexing on parameterless member functions is discussed.

5.5.1 Path Expression

As explained before in Section 5.1 the presence of path expression in queries poses a serious problem in terms of time. The relationship of between the objects forming a hierarchy can be represented using a multi-level graph called as *aggregation graph*, where each level i represents objects of a

```
BodhiDisplay(GenericType obj, char *attribName = " " )
{
//GenericType  is union of basic
//types and reference to other object.

/*Global Variables**/

/*   formatterObject points  to the current Formatter object */

/*   moduleControllerObject points to the current module's   */
/*   ModuleController object                                 */


/*Local Variables*/

char *typ,*tp;         //To store the type information

GenericType obj1;

 ClassController *conObj;//ClassControllerObject;


/*Algorithm*/

    typ = GetTypeOf(obj);
    if(typ is basic type)
    {
        formatterObject->ItemStart(typ,obj,attribName);
        formatterObject->ItemValue(typ,obj,attribName);
        formatterObject->ItemEnd(typ,obj,attribName);

     }
     else
     {
        conObj = moduleControllerObject->GetControl(typ);
        for each attribute a of type tp in typ
         {
            obj1 = conObj->GetAttrib(obj,a);

            formatterObject->ItemStart(tp,obj1,a);
            formatterObject->ItemValue(tp,obj1,a);
            if(tp is not a basic type) BodhiDisplay(obj1);
            formatterObject->ItemEnd(tp,obj1,a);
          }
       }
}
```

Table 5.11: **Pseudo code for BodhiDisplay Function**

type T_i. Each node n_{ij} denotes an object o_{ij} which is of type T_i. Each edge denotes the attribute relationship between two objects. So a path of type a.b is represented by an edge from object a to object b. A generalized path expression is of the form $o_0.a_1.a_2.\ldots.a_n$, where o_0 is an object of type T_0 and a_i is an attribute of type T_i in the type T_{i-1}, $1 \leq i \leq n$. A path expression $o_0.a_1.a_2.\ldots.a_n$ specifies a collection of objects o_{nj} of type T_n, where T_n is at level n deep in the aggregation hierarchy starting from type T_0 and the objects o_{nj}, $j \geq 0$, are reachable from o_0 in the *aggregation graph.*

Evaluation of Path Expression

This sub-section discusses various methods of evaluating a path expression.

The Naive Method

The naive method of evaluating a path expression $o_{0k}.a_1.a_2.\ldots.a_n$, is to start from object o_{0k} and fetch its attribute a_1 and then its attribute a_2 and so on.

Path Expressions into join

The drawback of the naive method mentioned above is that it needs n lookups for each object o_{0j} if the path is of length n, Still worse if one of the attribute in the path is a collection. (By visualizing the *aggregation graph*, one can come to a valid conclusion that the set of paths in the *aggregation graph* starting at level i and ending at i+k is equivalent to a join of the extents associated with the types $T_j, i \leq j \leq i+k$). So, one of the optimization followed by some of the optimizers, including Lambda-DB, is to convert the path expression into a join [Feg97b, BKG93] and then select from the resulted join table. This is also known as *materialization* of path expressions [CD92].

Path Dictionary

A data structure called *Path Dictionary*, which stores all the objects that are related by the attribute relationships, is proposed in [LL98c]. Using this, evaluation of any path expression $o_{0k}.a_1.a_2.\ldots.a_n$ turns out to be a lookup which returns a collection of objects which are reachable from o_{0k} and are at the depth of n in the aggregation graph. For fast retrieval it stores the OIDs of the objects involved in the attribute relationship in a generalized list structure. (for a detailed description on path dictionary see [LL98c] and see [Tad00a] for a detailed description of how it is implemented in Bodhi).

A path expression of the form $o_0.a_1.a_2......a_n$ can be treated as a series of functions $a_1, a_2....a_n$ applied on object o_0 and output of one function is piped as input to the next function. So, it is valid to convert a path expression (which is a series of functions) into a special operator f which is the composition of all the functions $a_i, 1 \leq i \leq n$, where f takes an object of type T_0 and returns collection of objects of type T_n. This function f is materialized in the form of path dictionary so that we need not evaluate this composite function f for each query, instead objects can be fetched from the path dictionary. Since path dictionary maintains associations among the objects, it is possible to fetch objects belonging to level i given any object at level i+k in the path(where i is a non-negative integer and k is an integer).

Implementation Details

In Bodhi, path dictionary is chosen as the index mechanism for aggregation hierarchy.(for more details see [Tad00a]). This sub-section gives the implementation details of the regarding the optimizations using the path dictionary.

Index Creation

Since there can be numerous attribute relationships between the classes and there is a space overhead associated with each path dictionary, only the mostly used paths are to be stored using the path dictionary. But the choice of mostly accessed paths is application dependent. As there is no fundamental rule regarding which path expressions are to be stored and which are not, in Bodhi, this choice has been given to user and user has to inform explicitly to the query processor to maintain the path dictionary for a given path. For example, to create a path dictionary index on the path *Plant.statesFound.capital.location* the OQL statement is shown below:

CREATE PATH INDEX pid1 ON Plant.statesFound.capital.location;

```
COMPR ( BAG, p,
        ITERATE (p , plants ) ,
        ITERATE(_X0 , PDITERATOR(pid1,0,2,pid1,p)) ,
        AND(EQ(_X0,bangalore_var),TRUE)
        )
```

Table 5.12: **Path Expression in Monoid Comprehension Calculus**

Transformation of a Query with Path Expression

As said before the queries are represented as monoid comprehensions in the intermediate form. We shall discuss,how an OQL query with path expression, transforms in each of the Lambda-DB's phases. Only the phases where there is a special treatment for path expression is explained here.

- **Parsing and Type Checking Phase:** In this phase OQL query is translated into monoid comprehensions. During the translation, the presence of path expressions of the form $o_0.a_1.a_2.\ldots.a_n$, is detected and if a path dictionary exists for that path expression then the whole query is rewritten by substituting every occurrence of the path expression $o_0.a_1.a_2.\ldots.a_n$, by a new unused variable, say _X0 and an iterate expression of the form

  iterate(_X0,pditerator (indexId,0,n,o_0) is added to the FROM comprehension. (The signature of pditerator is *pditerator (indexId, sourceClassNum, resultClassNum, srcVariable)*. This will act as a data generator (the composite function which was mentioned earlier) which takes the srcVariable (o_0)and returns all the objects that are at level resultClassNum and are reachable from srcVariable at level sourceClassNum in the aggregation hierarchy.

  By the definition of monoid comprehension calculus defined in Section 5.3.2, a monoid comprehension calculus

  can have a generator of the form v $\longleftarrow$e, where v is a variable and e is an expression or an extent name.

  In monoid comprehensions, pditerator is represented as an expression generating objects.

  For example, considering the query

  *SELECT * FROM p IN plants WHERE p.statesFound.capital= bangalore_var;*

  and assuming that the index is created as shown in Section 5.5.1. then the monoid calculus expression is as shown below:

  $BAG\{p|p \leftarrow plants,\ _X0 \leftarrow pditerator(pid1, 0, 2, p),\ EQ(_X0, bangalore_var)\}$

  The internal representation of this query, equivalent to the above monoid comprehension, is shown in Table 5.12. Here the expression denotes that the result collection type is a bag of plants and the projection variable is p and the remaining represents the comprehension where each item is a data generator or a predicate.

- **Translating into algebraic form**

  In the phase of translating the monoid calculus expression into monoid algebra, as mentioned in Section 5.3.8, all data generators are either converted into a join (if the domain is an extent) or an unnest operator (if the domain is an expression). So, here the expression having pditerator expression will be converted into an unnest operator as this is a data generator and it is dependent on another variable. Refer to 5.3.3 for a discussion of unnest operator. The "path" argument to unnest operator, here, is the *pditerator* expression which generates the data. For the above example the query, in the algebraic form, is shown in Table 5.13.

- **Physical Plan Generation**

  In this phase the physical operators implementing the algebraic operators will be substituted and the above example query's physical plan is as shown in Table 5.14

  After the physical plan is generated, C++ code will be generated for the plan. The code generated for the pditerator will be a function called PathIter which will initialize the path dictionary iterator (path dictionary interface has support to define iterators over the results) and iterates over the objects supplied by path dictionary.

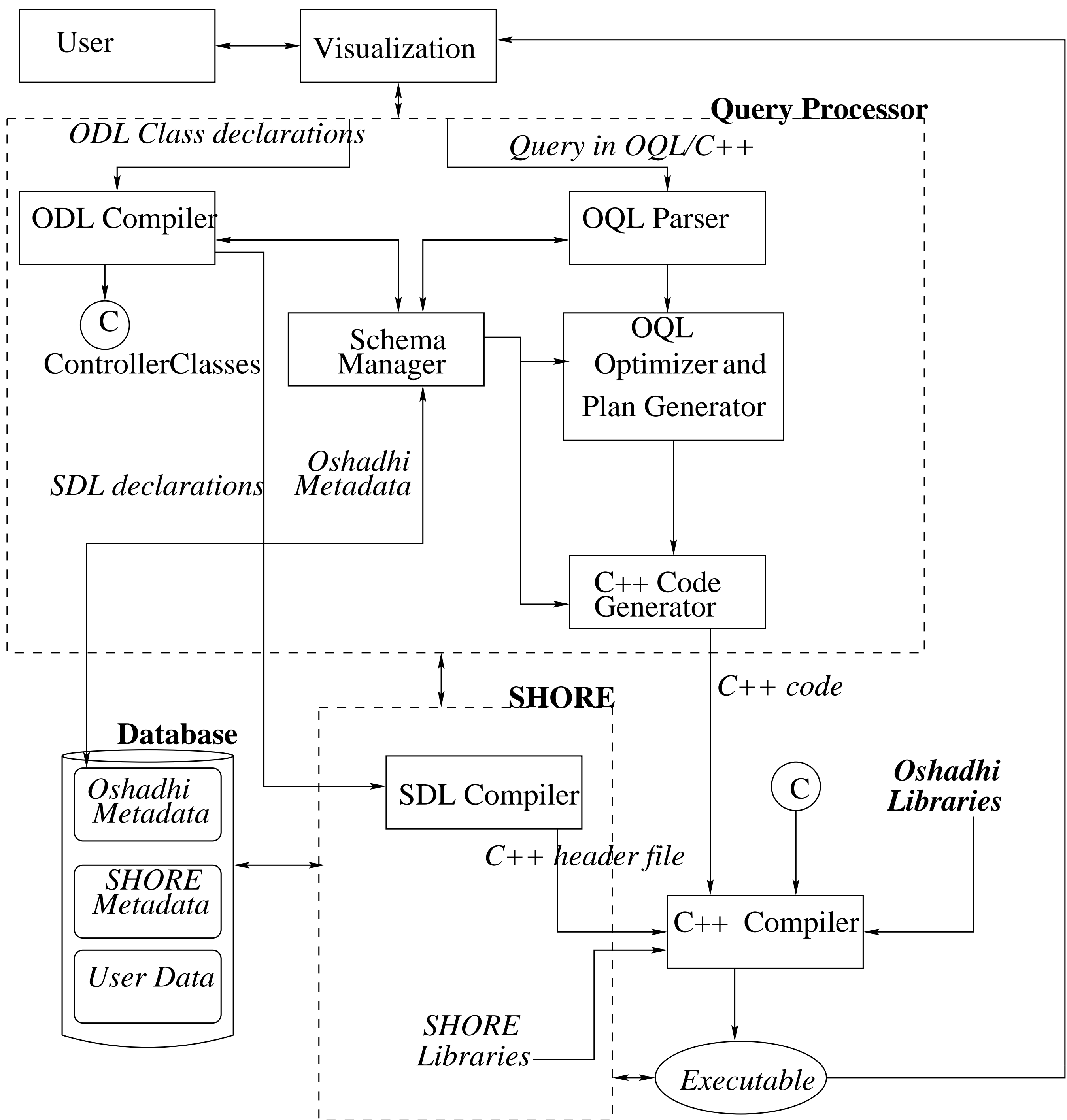


Figure 5.2: **Components of Query Processor and Query Flow in Bodhi**

```
REDUCE ( BAG,
         UNNEST ( BAG,
                  GET( BAG , plants ,_X1 , AND() ),
                  _X2 ,
                  PDITERATOR ( pid1 , 0 , 3 , _X1  ),
                  AND(EQ(_X1,bangalore_var))
                  ),
         _X3,
         AND()
         )
```

Table 5.13: **Path Expression in Monoid Algebra**

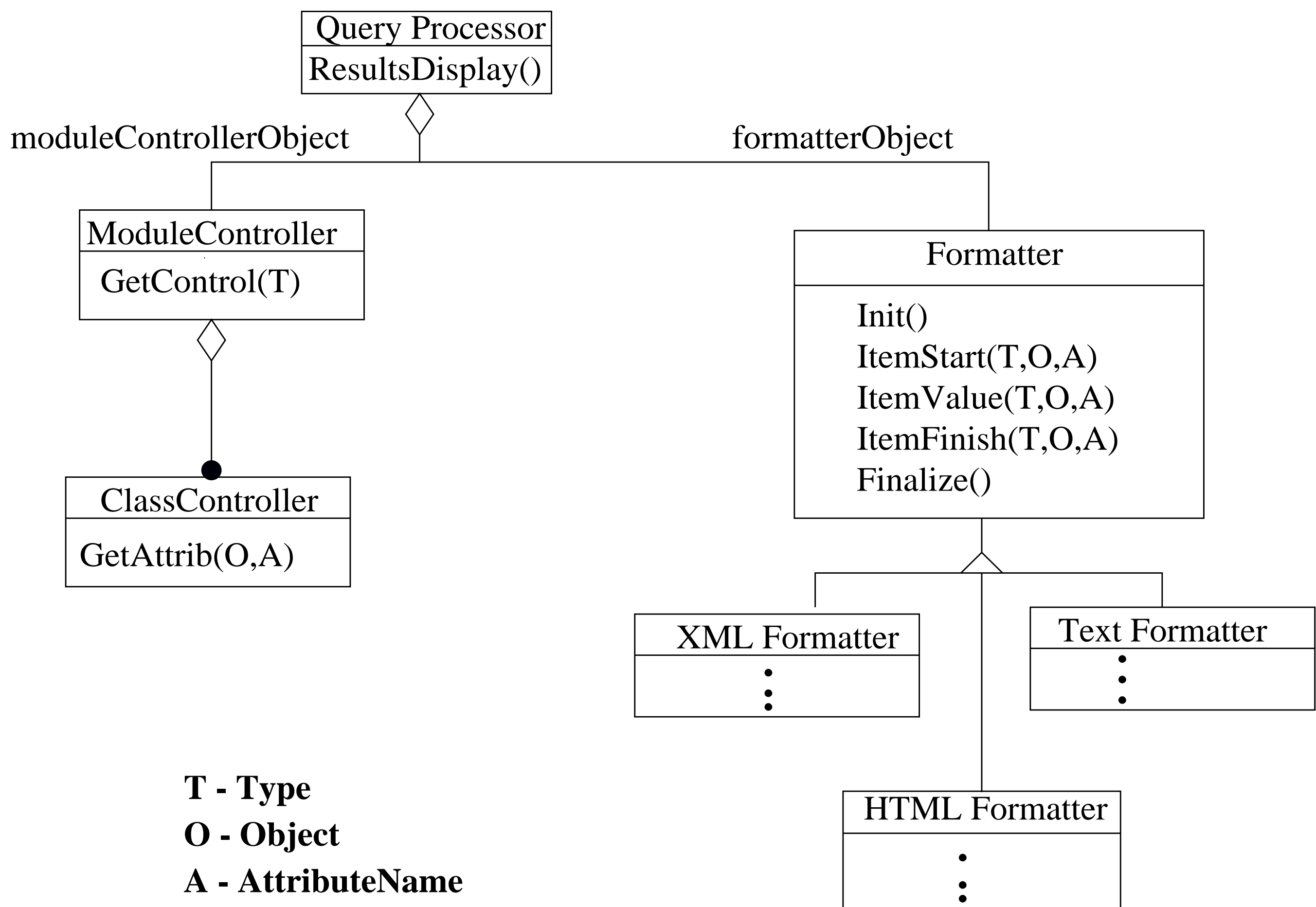


Figure 5.3: **Query Processor and Result Formatters**

Index Updation

Since path dictionary stores the relationships among all the objects which are related by attribute relationships, any object creation or update operation may trigger the need to update the path dictionary index.

If an object of type T is created/updated then all path dictionary indices having this type T as one of its component in the path will be updated by the query processor. For example, if path dictionary index is created as shown in Section 5.5.1 then the query processor will generate code to update the path dictionary index pid1 whenever any of the objects of type Plant, State, City or point is created/updated.

5.5.2 Derived Attributes

As mentioned earlier derived attributes can either be represented as functions or attributes. If they are represented as attributes then usual optimizations will suffice, But, as it is common, if they are represented as functions then indices cannot be build on that.

```
REDUCE ( BAG,
         UNNEST ( BAG,
                  TABLE_SCAN( BAG , plants ,_X1 , AND() ),
                  _X2 ,
                  PDITERATOR ( pid1 , 0 , 3 , _X1  ),
                  AND(EQ(_X1 , bangalore_var))
                  ),
         _X3,
         AND()
         )
```

Table 5.14: **Path Expression in Physical Plan**

```
CLASS Rectangle ( EXTENT boxes)
{
ATTRIBUTE LONG x1,x2;
ATTRIBUTE LONG y1,y2;
LONG Area() DEPENDS ON x1,x2,x3,x4;
};
```

Table 5.15: **An Example in ODL For Derived Attribute**

So, in Bodhi, we have followed the approach that any member function which does not take any parameters can be used to index the object. But, as the code for member functions can be written in C++ language, the attributes on which the function's return value is dependent is not known to the query processor. So, index may become inconsistent if one of these values is changed.

So, in Bodhi, we have followed the approach that the function at the time of declaration specifies the list of attributes on which it depends. Based on this information, whenever an object is created or the attributes of the object are modified, the query processor checks if there is any index which depends on this attributes, if so it computes the function and updates the index maintained on it. This facility to create index on parameterless member functions is not a new indexing mechanism as such, but extends the applicability of the existing access methods with the facility to index on the derived attributes.

Implementation Details

OQL has been extended to support the facility for a function to specify the dependencies (i.e., on which attributes of the same class this function depends) and in the phase of transforming the query from algebraic to physical plans, if the predicate is a function on which index exists then the INDEX_SCAN is used as the physical operator.

An Example in ODL

Suppose an application needs a Rectangle class and it need to represent the co-ordinates and Area and the database designer has chosen to represent Area as a function rather than an attribute. Then the class declaration in ODL declaration will be as shown in Table 5.15.

Index Creation

The syntax,in OQL, to create the index looks like:

CREATE BTREE INDEX bid1 ON Rectangle(Area());

where *bid1* is the index identifier.

An Example Query

Suppose the query is

*SELECT * FROM r IN rectangles WHERE r.Area() > 200;*

For this query, the usual processing will be, all objects from the extent rectangles will be retrieved and then the predicate $Area() > 200$ will be applied. But this forms a sequential scan. Using the index defined above, only objects satisfying the predicate will be retrieved.

5.6 Extensions

This section discusses some of the extensions that can be done to Bodhi from the point of view of query processor and optimizer.

- **Spatial Join:** There exists several spatial join algorithms which exploit the properties of particular spatial access methods [Güt94a]. So, if Bodhi is extended to support spatial joins then the query processor can be extended to support this spatial joins.
- **Index on Class Hierarchy:** Bodhi supports indices to be built on class hierarchies. The index chosen for this purpose was MT-index (for more details regarding the decision to use this mechanism and the details of MT-index see [Tad00a]). Query processor can be extended to support this indexing mechanism where user can optionally built an index on the class

hierarchy and optimizer will take care of it's presence and use it in query processing for the queries on class hierarchies.

- **Schema Evolution:**

  The idea is that type can evolve over time and still the old data should be accessible using the new type. (some systems use versioning and some uses restructuring the database after type changes and some have the flexible data model which can be resilient to changes).

- **Optimization of Several Queries:**

  Until now all the optimizations are done on a particular query. A possibility of optimization across several queries can be looked into.(This can be done by taking a list of queries and optimizing over this window of queries and the results of a query/sub-query can be materialized and used in next query).

- **Optimization of user defined functions:**

  Optimization of user defined functions can be done where each function exports its cost and the optimizer uses it to optimize the query.

5.7 Summary

This chapter concentrated on the query processing and query optimization of Bodhi. Since the query requirements of applications, even within a domain, vary significantly, a general purpose query language was chosen for Bodhi. ODMG's ODL and OQL [Cat93] has been extended to suit the needs of Bodhi(like support for various access methods and support for importing a module from other module).

The query optimizer developed was a query-rewrite rule-based optimizer, built by extending Lambda-DB's optimizer [Feg97b]. Query optimizer tries to optimize the query by taking into consideration the presence of indices like path dictionary [LL98c] on aggregation hierarchies and indices like Hilbert R-trees for indexing on spatial data types. Query processor supports indexing to be done based on parameterless member functions rather than just the attributes and optimizer optimizes the query by using this index if the query has a member function on which index is built. Query processor communicates with Visualization to receive queries in OQL and sends the data and the meta-data for displaying them. The interactions between the query processor and visualization are designed in such a way that Visualization can display the data and meta-data in its own format rather than query processor fixing the format of displaying the results.

Chapter 6

XML Visualization Interface

6.1 Introduction

There is a need to visualize all biodiversity information in sophisticated ways, because query results in biological and ecological domains, tend to be either enormous in size (geospatial queries, sequence homology searches etc) or complex with deep aggregation hierarchies (phenotype classification results). In case of spatial queries, which concern about the geographical distribution of plants and different climatic conditions, the results can vary from simple text to complicated overlapping images and multidimensional spatial objects. In order to make sense of these results, a proper visualization interface is needed to enable the end user query the database and visualize the results of such queries. Thus, visualization techniques play a very important role in Biological and Geographical Information systems.

For Bodhi, we have built a visualization interface that can be used to query the database server and view the results of such queries in sophisticated ways. With the rapid growth of Internet throughout the world, there cannot be a better medium for information exchange than the World Wide Web. Keeping this view in mind, the visualization interface has been designed to be accessible through a web-enabled browser (Netscape, Internet Explorer 5 or above etc) over the Internet.

The visualization module consists of a GUI-client and a GUI-server that interacts with the query processor. The GUI-server is built as a layer of the query processor and the GUI-client is built as a Java applet, using the Swing library package of JDK1.2. The whole application is deployed on the web. During execution, server-GUI classes are downloaded to the client machine, where queries are formed and passed over the network, using the framework of Remote Method Invocation, to the query processing layer. The query processor then executes the query and sends the object results back to the GUI-client. These query results are then displayed in the GUI-client using XML, which conforms to a DTD. The interface provides facilities to browse species characteristics for taxonomic database and geographical characteristics for spatial database. Generic database queries can be formed using the GUI. It also provides facilities to insert and delete tuples from the database.

6.2 Visualization Architecture

The visualization interface is a web enabled front-end consisting of a GUI-client and a GUI-server. There is a fixed set of protocols to communicate between the GUI-client, the GUI-server and the Query Processor. The architecture is depicted in Fig 2. During execution server-GUI classes are downloaded to the client machine using the security mechanism provided by the Java RMI framework. Once the client GUI is generated, queries are formed through user interactions by filing appropriate, customized forms and menus. These queries are then sent to the GUI-server over the network through the same RMI framework. GUI-server application program then interacts with the Query Processor through the Java Native Interface and passes the query into it. The query processor processes the queries using the services of Taxonomy, Spatial and Genome modules and passes the results back to the GUI-server which in turn passes the results to the GUI-client where they are displayed appropriately using XML.

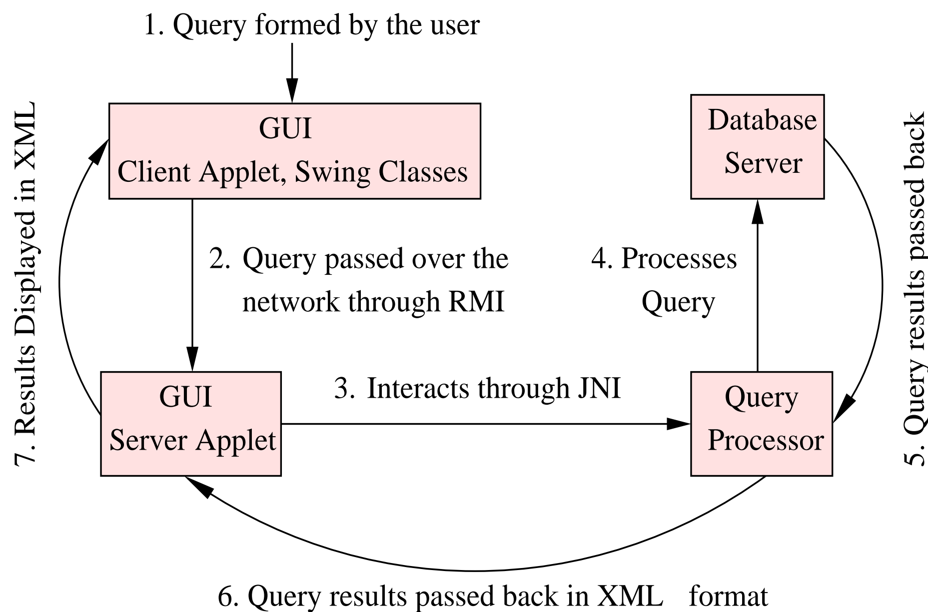


Figure 6.1: Bodhi Visualization Architecture

6.3 System Requirements

6.3.1 Modeling requirements

Modeling the graphical user interface to support the above kind of ecological data needs several issues to be looked at:

1. The system should support the inherent aggregation hierarchy of the taxonomic data, the maps and the environmental conditions associated with the geographic data.

2. Typical queries on taxonomic data require specifying the identifying properties of a taxon and getting information about that taxon matched on the specified properties. The GUI should provide provisions for entering the characteristics of the taxons and form the query.

3. Spatial queries usually require displaying of maps and extracting information about geographic regions which have certain kinds of environmental characteristics or in which certain kinds of species are distributed.

4. Sometimes there are needs to modify the schema in the bio-diversity application. For instance, new types of characters may be added to aid the process of identification or new geographical organizations may evolve with time. The model for the GUI should be flexible enough to support these features.

5. In today's world, the world wide web is the best medium for information exchange. Keeping this view in mind, it would be most suitable for the visualization interface to be a web-enabled front-end, where the end user fills up the customized forms to form their specific queries and submit it to the GUI server over the network in an efficient and secure way and get back the results.

6.3.2 Typical queries

Here we list some of the typical queries in the bio-diversity application:

- View the information about an order, a family, a genera or a species having a certain set of characteristics.

- Display the map of a geographic region which grows certain vegetation types and possesses certain soil properties.

- Display the distribution of a species in a particular geographic location.

- Identify a species based on certain flower / leaf / seed / fruit characteristics.

6.4 System Design and Modeling

In this section, we discuss the issues involved in the system design for visualization architecture, the other choices we had for implementing the system, the ones which we chose and reasons for our choices.

6.4.1 Library Package

Over the years various interface libraries have come up that provide configurable interface components, which allow the developers to implement sophisticated graphical user interfaces.

The cross-platform GUI library, wxWindows, allows C++ applications to compile and run on several different types of computers. This toolkit consists of one library per supported GUI, such as Motif [Bra92] or Windows. Besides providing a common API (Application Programming Interface) for GUI functionality, it provides functionality for accessing some commonly-used operating system facilities, such as copying or deleting files. wxWindows is a 'framework' in the sense that it provides a lot of built-in functionalities which the application can use or replace as required, thus saving a great deal of coding effort.

Swing Library package

Inspite of the advantages provided by the above mentioned software packages, we have chosen the Swing package [E+98] from the Java Foundation Classes (JFC) [E+98] for the following reasons:

- Swing development has its roots in the Model-View-Controller (MVC) architecture. The MVC-based architecture allows Swing components to be replaced with different kinds of data models and views. The Swing library supports a cross-platform look-and-feel, also called the Java look-and-feel, that remains the same across all platforms where ever the program runs. It also allows the developer to create his/her own look and feel.
- Swing supports lightweight component development. For a component to qualify as lightweight, it cannot depend on any native system classes, also called "peer" classes. In Swing, most of the components have their own view supported by Java look-and-feel classes instead of depending on any peer classes. Swing supports a good number of sophisticated user interface lightweight components such as buttons, check boxes, combo boxes, tables, text fields and so on as depicted in Fig 3.

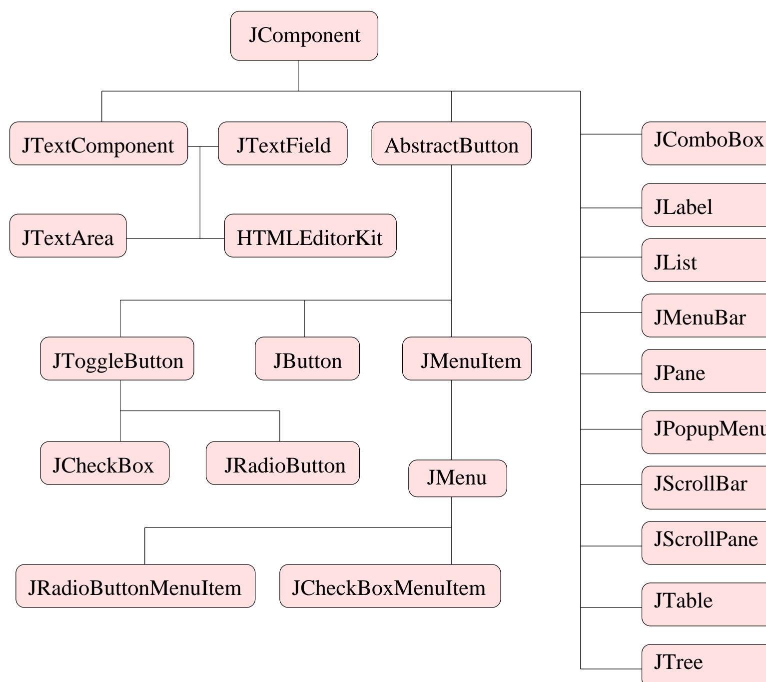


Figure 6.2: Swing's Component Hierarchy

- Swing components function based on the delegation event model (introduced in JDK1.1). The delegation event model supports a clean separation between the core program and its user interface. The delegation pattern allows robust event handling that is less error prone due to strong compile checking. In addition, the delegation event model improves performance because the toolkit can filter out undesirable events (like high frequency mouse events) that are not targeted to execute any function.
- As the underlying platform for the Swing package is Java it allows an application to be platform independent and to be easily deployed on the web browser.

6.4.2 Network Interconnection

Having decided which software library to use for the GUI development we had a couple of choices for deploying the network interconnection between the user interface and the query processor.

Java sockets

The first choice for implementing the basic network communication was Java sockets [N+99], which are flexible and sufficient for general purpose communication. Here, the server address needs to be hard coded in the source code and whenever the client (GUI) sends a request to the server it sends to the fixed server IP and port number. The server always listens to that port number for incoming requests and when it gets one, it forks one process to which it gives the request to handle and it continues to listen to the same port for more incoming calls. However, sockets require the client and the server to engage in application level protocols to encode and decode messages for exchange. Designing such protocols is cumbersome and can be error-prone.

Remote Procedure Call

An alternative to sockets is Remote Procedure Call (RPC) [Sta97], which abstracts the communication interface to the level of a function call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR.

Remote Method Invocation

However, RPC doesn't translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. Remote Method Invocation (RMI) [Dow98], provided by Java platform matches the semantics of object invocation, where a local surrogate (stub) object manages the invocation of a procedure on a remote object. It can dynamically resolve method invocations across Virtual Machine (VM) [L+99] boundaries, thus providing a fully object-oriented (OO) distributed environment. Besides, RMI supports various reference semantics for remote objects, such as live references, persistent references and lazy activation. It also provides a robust security mechanism for downloading classes and a distributed garbage collection algorithm for active objects. Thus, the network interconnection between the GUI client and the Database server has been deployed using Remote Method Invocation.

6.4.3 The Native Interface

The GUI server application program in our visualization architecture needs to call C++ methods from the query processor. For this, we have chosen the Java Native Interface (JNI) [Lia99]. It allows Java code that runs inside a Java Virtual Machine (JVM) to interoperate with applications and libraries written in other programming languages, such as C, C++ and Assembly. JNI also lets the native method to create, update and access Java objects and share them. The native methods, using the JNI can call the existing Java method, pass it the required parameters and get the results back when the method completes.

6.4.4 Displaying Query Results

XML

Once the query has been executed in the database server, the results have to be transported back to the visualization interface for rendering on the screen. There are several data exchange formats,

such as: DELTA (Description Language for Taxonomy), MEDATLAS (for exchanging oceanographic data), CSDGM (Content Standards for Digital Geospatial Metadata), DIGEST (Digital Geographic Information Exchange Standards) etc, which are used by different commercial applications for exchanging taxonomic and geospatial information. In Bodhi, we have used *XML* (eXtensible Markup Language) for meta-data and query-results exchange between the GUI and the database server. Reasons for choosing XML as the data exchange format are as following:

XML is structured : Using XML, structural relationships that exist inside the data can be embedded into it. A document type definition (DTD) is used to create these relationships and the XML document conforms to this DTD. The validity and integrity of the document is checked by an XML parser.

XML is platform independent, textual information : Information in an XML document is stored in plain text. There are several advantages of keeping things in plain text. First, it is easy to write parsers and all other XMl enabled technology on different platforms. Second, it makes everything very interoperable by staying with the lowest common denominator approach.

XML is language independent : By being language independent, XML bypasses the requirement to have a standard binary encoding or storage format. Language independence also fosters immense interoperability amongst heterogeneous systems and it is also good for future compatibility.

XML is web enabled and totally extensible : XML is derived from SGML (Standard Generalized Markup Language). So in essence, the current infrastructure available today to deal with HTML content can be re-used to work with XML. Even if clients donot support XML natively, there are Java Servlets which can convert XML with style-sheets, to generate plain HTML that can be displayed in all web browsers. Using XML to pass parameters and return values on servers, makes it very easy to allow these servers to be web-enabled. Finally, by not predefining any tags in the XML Recommendations, the W3C (World Wide Web Consortium) allowed developers full control over customizing the data. This makes XML very attractive to encode the data that already exists in legacy databases

6.5 Implementation Details

6.5.1 Data Transaction between the GUI and the Database server

Database Schema and Meta-Data Exchange : There are several occasions when data transaction between the client-GUI and the query processor is needed.

The first transaction takes place when initial connection is made to the database server and the GUI classes are loaded to the client machine requesting for database schema and any other meta-data that might be useful in processing the queries at a later stage. The query processor takes up this request and searches the database for such information and ships it over to the client GUI. The format in which this meta-data is exported to the visualization interface follows the BNF (Backus Naur Form) grammar which is shown in Fig 4. This information transaction is very much essential, because the database schema might be updated at any point of time and the GUI must take care of the changes. The schema might change for various reasons such as:

- new character types may be added to the taxonomic database to aid the process of species identification and their distribution in different geographical locations,
- or new geographical organizations may evolve with time

Once this meta-data are exported from the query processor, the user interface is generated according to that schema. For example,

- the data types that are present in the database such as: taxonomy, spatial, genome, bibliographic;
- the kinds of operations that are supported by the query processor; such as: adding a taxon to the taxonomy database, viewing special characteristics of a species or searching the database that meets certain specified characteristics on a taxon and a geographical location;
- the characteristics of each individual species;
- the properties of the geographical locations, etc

are to be taken into consideration when generating the user interface.

```
<Export-Format> := <className> ( <Export-Dcl> { "," <Export-Dcl> } )

<Export-Dcl>     ::= <attrib-Dcl> | <relationship-Dcl> | <method-Dcl>

<attrib-Dcl>     ::= attribute ( <attrName>, <attrType> )

<method-Dcl>     ::= method ( <methName>, <returnType>,
                         params ( <params-Dcl> { , <params_Dcl> } ) )

<params-Dcl>     ::= ( in | out | inout ) ( <attrName>, <attrType> )

<attrType>       ::= <Identifier>

<methodType>     ::= <Identifier>

<className>      ::= <Identifier>

<methName>       ::= <Identifier>

<Identifier>     ::= [a-zA-Z][a-zA-Z0-9_]*
```

Figure 6.3: Meta-Data Export Format in BNF

OQL Query Generation

Once the user interface is generated from the database schema the end user can navigate through the GUI and form the query by filling appropriate customized forms and menus. This query is then translated into an Object Query Language called OQL [Kon00c].

OQL is a declarative query language standard proposed by ODMG (Object Data Management Group) [Cat93] and has been widely accepted as a query language for OODBMS. OQL, by itself is not computationally complete; however queries can invoke methods written in the host language and any host language method can include OQL queries. OQL supports only B+ tree [Che00] index structures which is not enough for Bodhi. So it has been extended to support several index structures.

Example queries in OQL form

Here, we give a flavor of the query language designed for Bodhi by showing some example queries.

1. **Object Creation:**

   To create a persistent object, say of type City the OQL query takes the following form:

   po := City(location:p, belongsToStates:s);

   where s and p are state and point objects created similarly. po is the newly created City object.

2. **Query on a Class hierarchy:**

   To search for a species which meets certain characteristics:

   s := Species(name : A, Properties(
       [Stalk] : SUB_SESSILE,
       [Arrangement] : DISTICHOUS,
       [OvaryPosition] : SUPIRIOR));
   FOR EACH G in Genera
       WHERE G.name = "B"
       G.Search(S);

   This will first create a Species object s, which has name A and properties as specified, and will search in Genera having name B for information about that Species.

3. **Query with a Join:**

   To find all lakes which flow through the city Bangalore the query will be:

```
FOR EACH i IN (
        SELECT * FROM c IN cities, w IN waterbodies
        WHERE c.name = "Bangalore"
        AND c.location.overlap(w.location)
        AND w.type = LAKE )
DO cout << i →w→name << endl;
```

Once these OQL queries are generated by the client-GUI, they are passed over the network through Remote Method Invocation to the server GUI and then, to the query processor through Java Native Interface. The query processor first parses the query for its correctness and then optimizes it. Then the physical plans are generated and the C++ code is produced for the query in terms of these physical plans. Finally, the query is executed and the results are shipped back to the visualization interface.

6.5.2 Query Results Exchange

There are several issues which come up when the query results are shipped back to the visualization interface. First of all, the mechanism by which the results are going to be passed back, and then the way in which the visualization module is going to interpret these results.

The first problem is tackled by using Remote Method Invocation, where the results are packaged into an object and then transmitted through the network using the concept of Java Object Serialization.

The way in which the visualization interface interprets the query results is very important, because there can be various types of results and each of them might need special treatment for displaying them. So the GUI server needs the type of these results. For example, output of a query can be a set of numbers which might represent a set of points describing a polygon or it might be the salary of an employee. In the former case, the visualization interface has to render the points and display the polygon, whereas in the latter case it has to just print them. So the results must also have associated meta-data describing the structure and context of the output. But the issue is who is responsible for generating this meta-data. There are two solutions to this:

1. The user writes the code for displaying the object's attribute values and the meta-data in the action part of the query in a format acceptable by the GUI-server.
2. Bodhi will provide a function which does all this chores of displaying the data and meta-data and the user can just call this object that is to be displayed as parameter.

We have followed the second approach by providing a function *BodhiDisplay(object)*, which traverses through the object recursively and outputs all the attributes of the object including the meta-data. A DTD (Document Type Definition) has been designed to structure the query results. These results are then packaged into XML format which conforms to the DTD and validated by an XML parser and then shipped back to the GUI-client. In the following, we provide some part of the DTD that we have written based on the different data models implemented in Bodhi.

Having decided how to generate the meta-data and how to pass it over to the visualization module the next issue is of displaying the query results according to its meta-data. This meta-data which specifies the structure and context type of the object is nothing but XML tags [B^+98], which have meanings as described in the DTD. So an XML supported browser (such as, IE5) can be used to view the query results. A Formatter class in the query processor does the job of converting the query results into an XML format. Some part of the DTD, which is based on the Bodhi Object Model is depicted in Table 1.

At the end of this thesis we have shown some of the screen capture imagaes of the GUI.

6.6 Related Work

There have been several efforts to build visualization interfaces for biological and GIS applications. The educational software, *Paradise* [D^+94a], from the University of Wisconsin, has a GUI mainly for GIS applications. The Paradise front-end has been implemented using Tcl/Tk and it includes key features such as: display of objects with spatial attributes on a 2-D map, layered display of overlapping objects, updating objects from the GUI etc. But the GUI cannot be deployed on the web because of the underlying platform. The *Idrisi* software package for GIS and image processing applications provides raster geographic analysis and a set of extensive tools for image processing.

```
<!-- Top level declarations -->
<!ELEMENT BODHIDISPLAY    (TAXONOMY | SPATIAL | (TAXONOMY, SPATIAL))>
<!ELEMENT TAXONOMY          (ORDER*, FAMILY*, GENERA*, SPECIES*)>
<!ELEMENT ORDER             (NAME, FAMILY*, GENERA*, SPECIES*)>
<!ELEMENT FAMILY            (NAME, GENERA*, SPECIES*)>
<!ELEMENT GENERA            (NAME, SPECIES*)>
<!ELEMENT SPECIES           (NAME, PROPERTIES*)>
<!ELEMENT NAME              (#PCDATA)>
<!ELEMENT PROPERTIES        (FlowerChar*, LeafChar*, BranchChar*, FoliageChar*,
                             CanopyChar*, StemChar*, HabitChar*)>

<!-- Toplevel Flower Properties -->
<!ELEMENT FlowerChar        (Arrangement?, Stalk?, Placentation?, InfloChar?,
                             SepalChar?, PetalChar?, Claw?, Aestivation?,
                             AntherChar?, OvaryChar?, StyleChar?, StigmaChar?)>

<!-- Enum values -->
<!ELEMENT Arrangement       (keyword)+>
<!ELEMENT Stalk             (keyword)+>
<!ELEMENT Placentation      (keyword)+>

<!-- InfloChar -->
<!ELEMENT InfloChar         (InfLocation?, FertilePart?, AxisNature?,
                             AxisThickness?, AxisOutline?)>
<!ELEMENT InfLocation       (keyword)+>
<!ELEMENT FertilePart       (keyword)+>
<!ELEMENT AxisNature        (keyword)+>
<!ELEMENT AxisThickness     (keyword)+>
<!ELEMENT AxisOutline       (keyword)+>

<!-- SepalChar -->
<!ELEMENT SepalChar         (NerveChar?, SymmertyChar?, Form?, FusionChar?, Color)>
<!ELEMENT NerveChar         (keyword)+>
<!ELEMENT SymmertyChar      (keyword)+>
<!ELEMENT Form              (keyword)+>
<!ELEMENT FusionChar        (keyword)+>
<!ELEMENT Color             (keyword)+>
...
```

Table 6.1: Part of the Document Type Definition (DTD) for the Taxonomic Object Model

There are other commercial GIS applications such as *ArcView, MapInfo, ERDAS, ERMAPPER* which are mainly used for map display but they are not capable on being deployed on the web through Internet supported browsers.

6.7 Summary

In this chapter we have talked about the visualization architecture and implementation details for a general purpose ecological application. The whole application can be accessed through a web enabled browser and database queries can be formed and executed over the network. The mechanism for meta-data and query results exchange between the database server and the end user is XML, which conforms to a document type definition (DTD). It also supports display of maps and species-specific characteristic for geospatial and taxonomic queries.

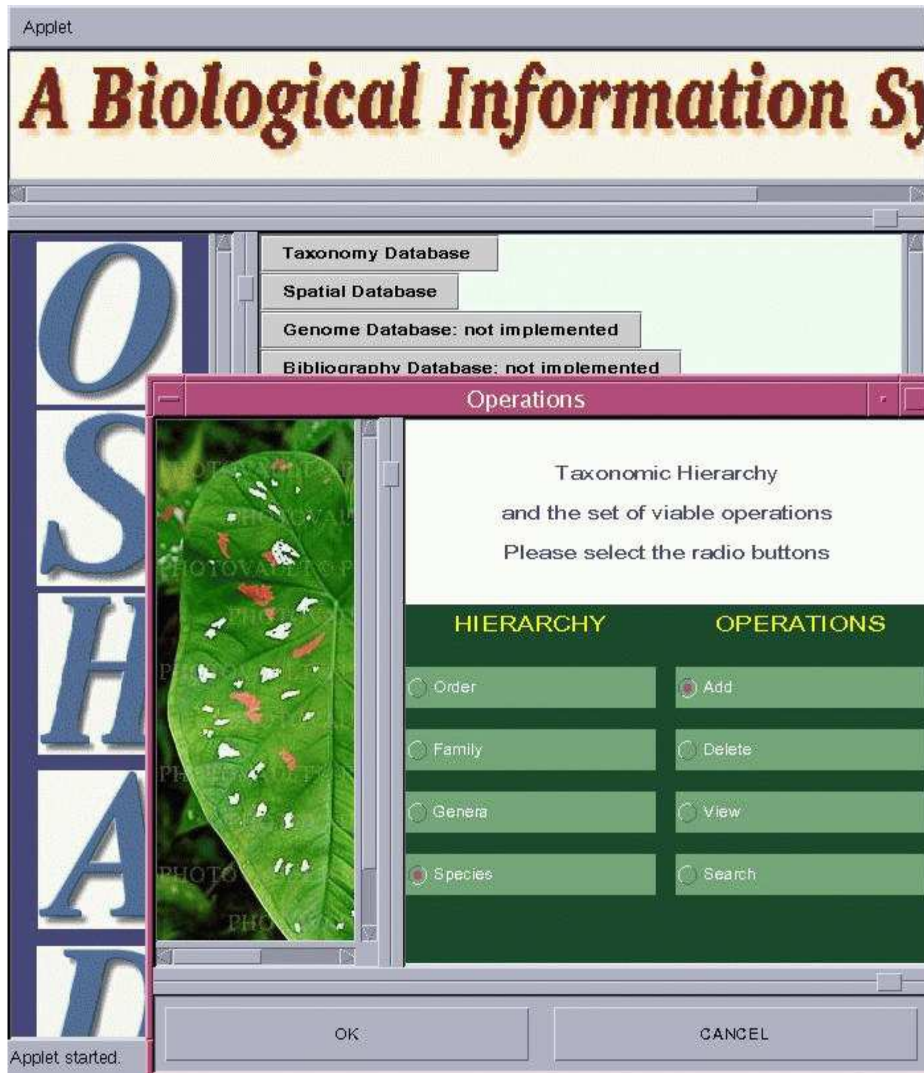


Figure 6.4: GUI screen shot for Taxonomic Database Operations

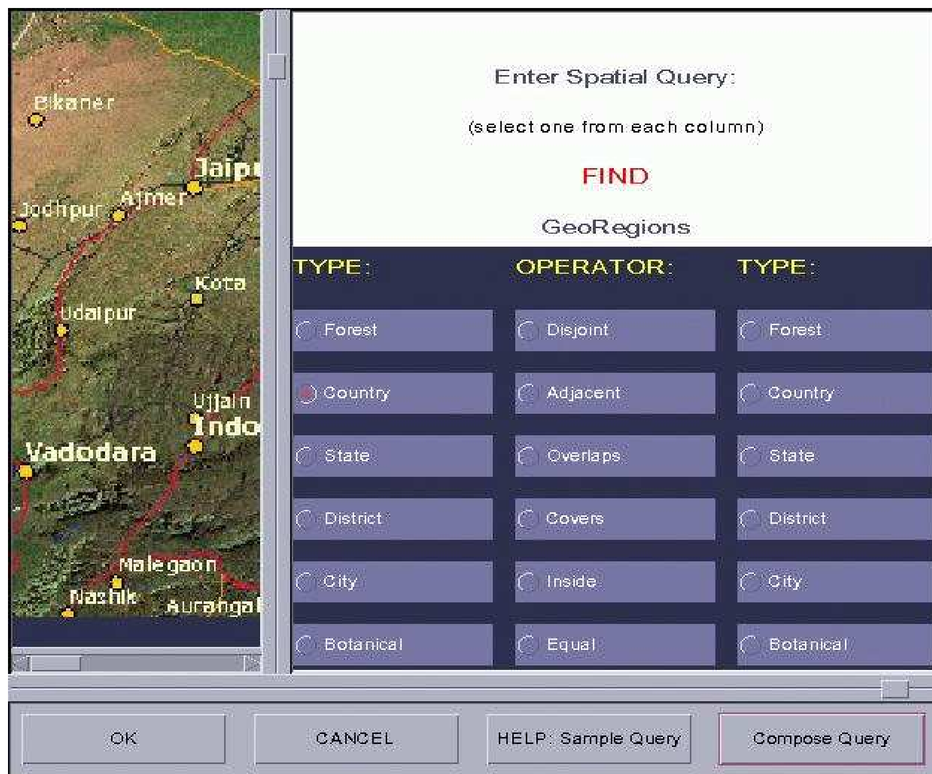


Figure 6.5: GUI screen shot for Spatial Database Queries: Spatial characteristics are specified using this form

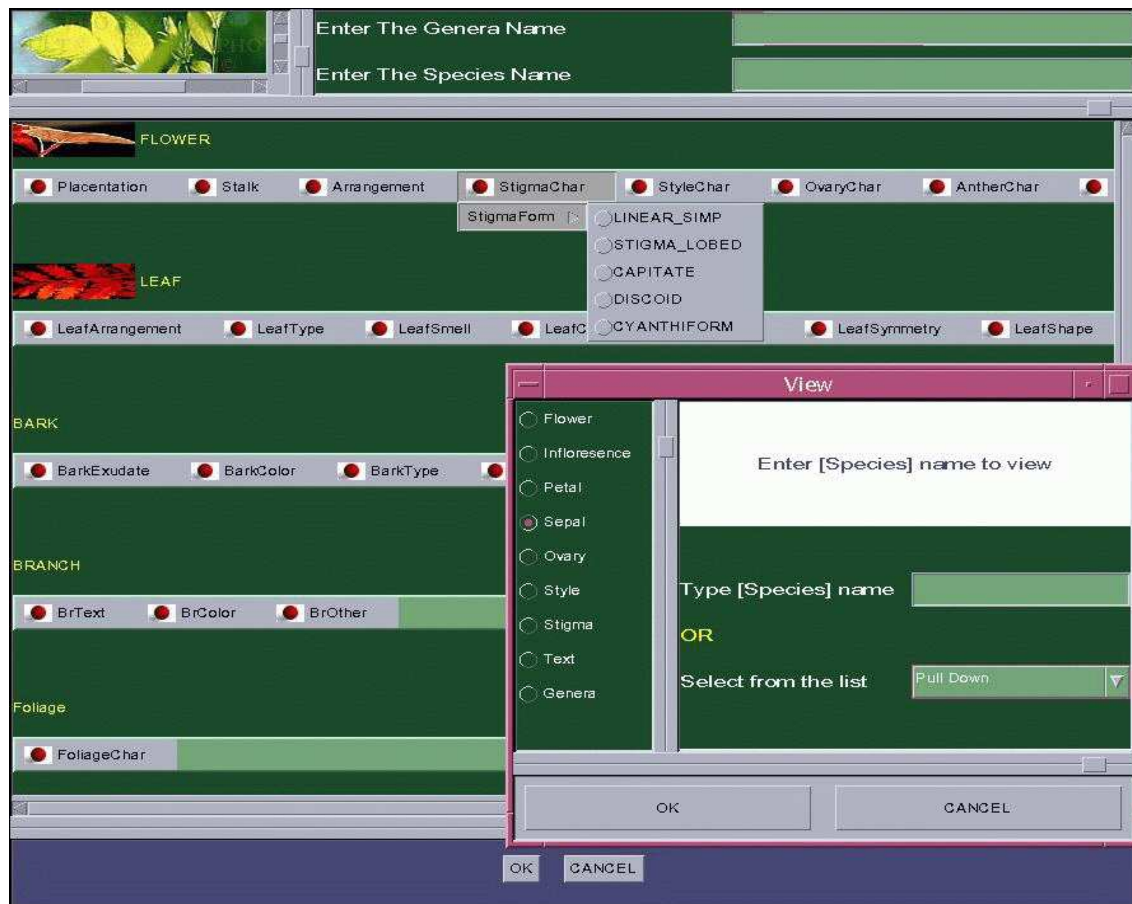


Figure 6.6: GUI screen shot for Taxonomic Database Queries: Taxon characteristics are entered using this form

Chapter 7

System Integration, Testing and Evaluation

In the previous chapters, each module was separately built and tested. Spatial data represents geographical location of a species. So, queries in various biological community is associated with spatial data access. In addition spatial queries are complex and compute-intensive. This makes spatial module one of the important modules that needs to be evaluated. In this chapter spatial module is integrated in the system along with taxonomy module. As shown in Fig 7.2, schema definition is done through query processor and the storage is taken care by *Shore*, at lower level than query processor. Any methods that is called for only reading underlying object needs to be defined as *constant* in SDL (*Shore* Data Language). But during schema definition, the operations of the class of objects cannot be defined as *constant*s. This incompatibility between *Shore* and the query processor leads to entry of operation in log record, even though the method is called for reading only. This unexpected log entry leads to log overflow, crippling the operation of *Shore*. In addition several other problems are found in *Shore* itself. One interesting problem noticed is index once built on any extent cannot be deleted until the extent itself is destroyed. This restricts the flexibility of user operations. Unlike SQL, **Drop Index** cannot be implemented here. In this chapter several such vulnerable errors are taken care of. Further, several issues that are faced during addition of spatial module, are discussed in section 7.3. The query processor is built using *lambda-DB*, which converts all the schema definition and queries from user level to SDL, understood by *Shore* as shown in Fig 7.2. The major hurdle in this project is incompatibility of code generated by *lambda-DB* and the code that should ideally be generated. So, several extensions to *lambda-DB* is done. Some of them are mentioned in section 7.3.

As mentioned earlier in any biological information system, a large extent of queries are associated with spatial operations. Some of them are picturized in Fig 7.1. Some example spatial queries to illustrate the pictures are

- Retrieve all the forests that are in Karnataka
- Select all the states that contains himalayan vegetation and forest area more than 100 sq. km.
- Select all the cotton growing areas that are within a distance of 300 km. from Mumbai.

Some of these types of queries are described in section 7.4.2. As seen in Fig 7.1, spatial queries include several overlaps, adjacency, containments and disjointness. For example, in case of second query above, all the polygon objects that overlaps with another set of polygon objects needs to be retrieved. Mostly all the queries in spatial domain like overlaps, containments include joins. All such joins in spatial queries mostly resemble the range selection and joins in non-spatial domain which are complex and compute-intensive and hence costly. Again, unlike non-spatial database system, where number of tuples in one page is quite large, spatial data type being inherently complex and hierarchical, there can be very few objects in the type in a page. This means disk access increases, if the number of objects are same as number of objects in non-spatial system. All these constraints makes spatial data management more complex. This complexity and dominance of spatial queries in biodiversity application makes it essential to evaluate the spatial module in the system.

Several functions in spatial and taxonomy modules in *Bodhi* are tested after integration. Since visual tools are still not developed, the correctness is tested by generating small tractable data. The unavailability of taxonomic data and specific benchmarks restricts the evaluation of whole system including the two integrated module. However, presence of several geo-spatial benchmarks and importance of spatial module leads to evaluation of spatial module in *Bodhi*. Benchmarking can be broadly categorized into two parts:

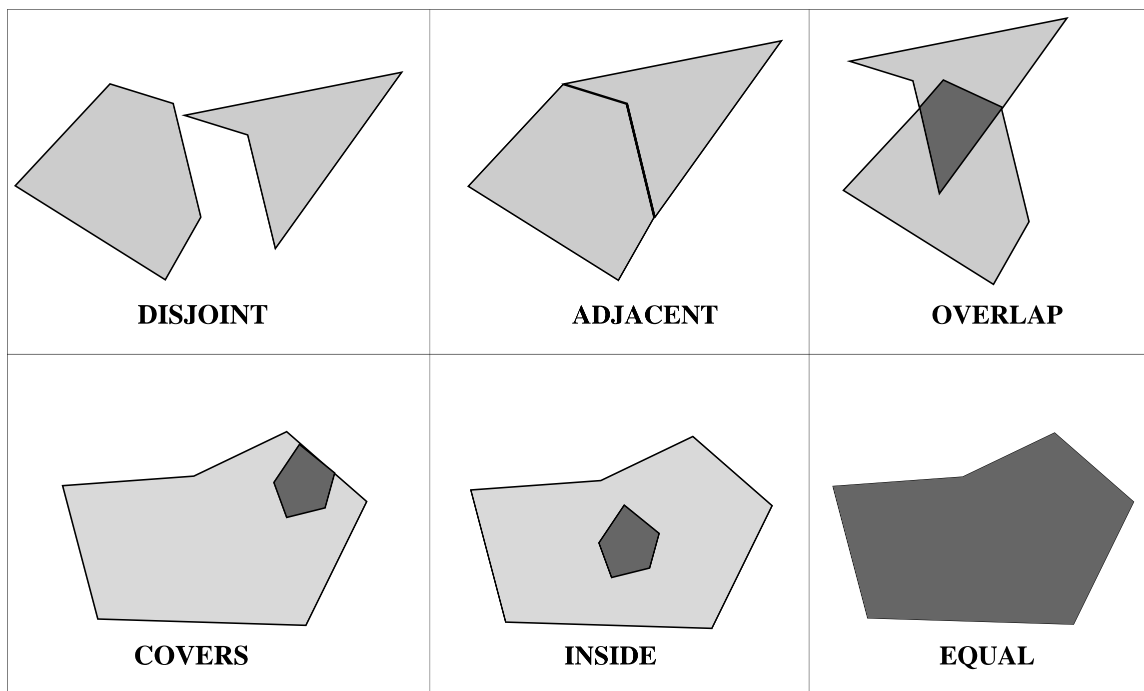


Figure 7.1: Spatial relationship figure

- Functional Benchmarking, where the queries cover the ranges needed for the application. System throughput, and other system performance issues are generally not measured using this benchmark.
- Performance Benchmarking, where performance evaluation of system is done by stressing the system with data. The stressing of system is done by scaling up the data.

The choice for functional benchmarking for evaluating spatial module rather than performance benchmarking stems from the fact that the spatial queries varies to a large extent, from simple non-spatial selection to complex spatial joins. This leads to opt for *Sequoia* 2000 benchmark[MSM93]. *Sequoia* 2000 is a benchmark that captures the data base requirements of earth scientists working in SEQUOIA 2000 project. The benchmark can also be used by Geographic Information Systems (GIS) and several other engineering and scientific DBMS users including biologists. This benchmark contains real queries that represents the tasks of biologists. Moreover *Sequoia* does not scale up data to stress system. Hence, it only looks at various functional constraints of database rather than system performance. This leads to choice of *Sequoia* for evaluating spatial module. Other than queries related to raster data, *Bodhi* supports *all* queries mentioned in *Sequoia* 2000 benchmark. Raster data is still not supported in *Bodhi*. In near future, this will be implemented in spatial module. Some evaluation results are mentioned in section 7.4.3.

7.1 Motivation

Although previously each module is separately tested, various complexities in different modules can lead to incompatibility. On course of system integration all these incompatibilities are identified and removed. Moreover, the spatial module is never tested with large volume of data where the system performance is bottleneck. In this chapter, large data volume is put in database successfully.

7.2 Related Works

There are several geo-spatial systems available from both commercial sectors and research institutes. *Paradise*[D$^+$94a] is a geo-spatial system built on top of *Shore*. It uses object-relational database system. Its main goal was to test scalability of various geo-spatial query processing techniques. Although the motivation of *Bodhi* is different than *Paradise*, as spatial module is essential part of bio diversity application, it needs to support various features of spatial system. *Paradise* also uses *Sequoia* benchmark for evaluating their system. They scale up their data to achieve the goal project as mentioned before. *Paradise* extend SQL to support spatial data primitives and set of

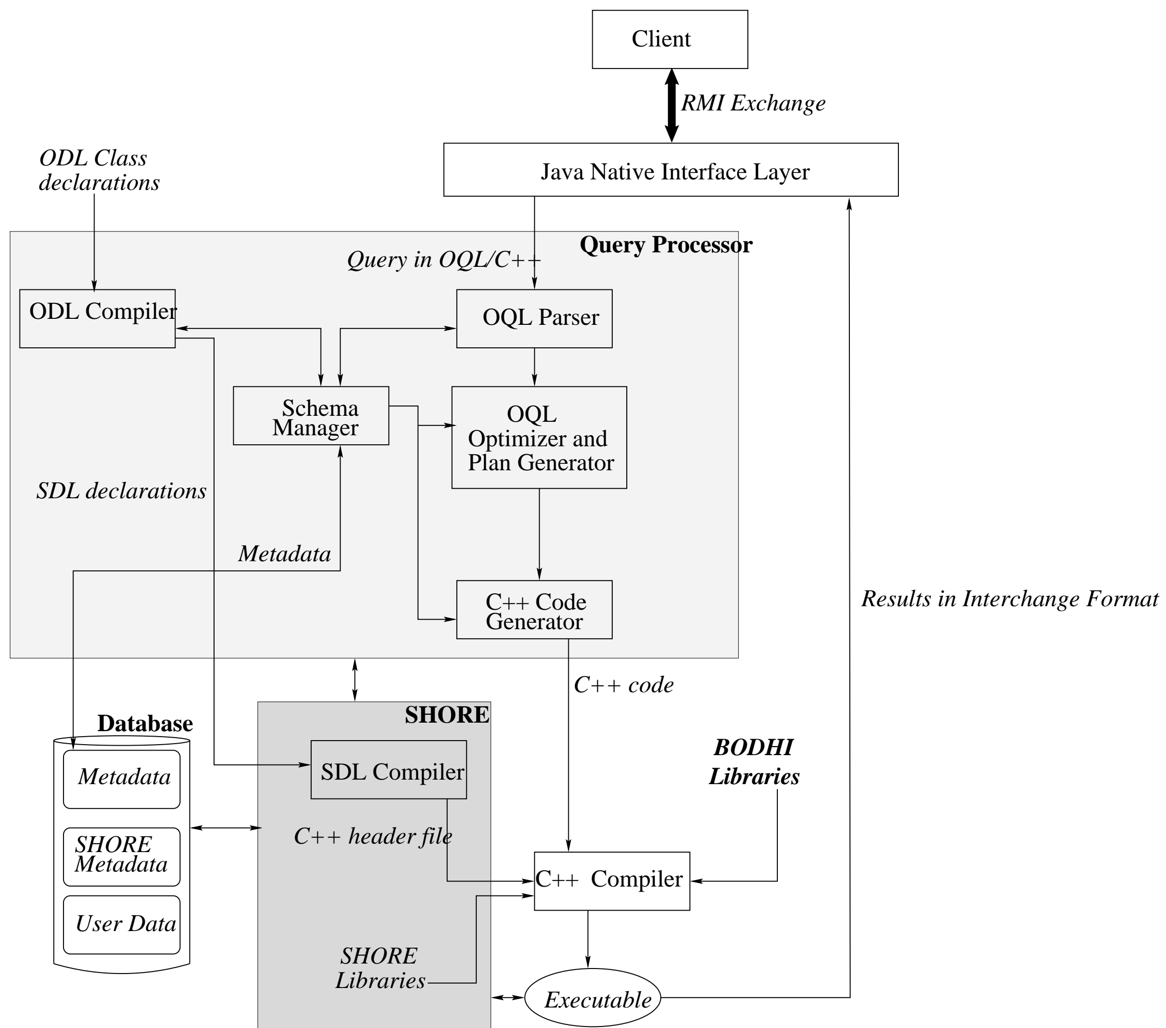


Figure 7.2: Query flow in *Bodhi*

spatial operators. Whereas *Bodhi* uses ODL (object Query Language) for schema definition and OQL (Object Query Language) for query. All the data types are implemented as abstract data type using C++ in both the systems. Since object-oriented system is used in *Bodhi*, it is possible to handle complex queries like recursive queries, which were not tested by *Paradise*.

7.3 Issues in Integration of Spatial Module

The project started with the view of lack of benchmark in this application domain. But to create a benchmark or even to run an existing benchmarks, the system should be integrated. Running benchmarks separately on individual module can only give fair idea of the component, but the whole system performance cannot be evaluated. This makes it essential to integrate the spatial module in *Bodhi*. On the course of system integration several problems regarding interfacing with query processor are faced. Some of these are solved by modifying some amount of codes from developed module. Some of the vital problems and the ways they are been solved are as follows:

7.3.1 Extension of Schema Definition Language

Previously the schema for spatial module is developed using SDL. But as mentioned earlier it needs to be done using ODL, this results in transforming the SDL schema definition into ODL. But as mentioned earlier due to incompatibility of λ-DB with *Shore*, ODL has been extended to handle constant methods in schema definition. These are used for only reading but no updation of objects. Inclusion of new tokens in language needs to be reflected in schema manager code, as the meta information collected by schema manager during parsing is used for type checking.

Any object when needs to be updated needs **update()** function call in SDL. On calling **update()** function, *Shore* is informed about intention to modify the object, and hence a log record is written for the operation on object. When λ-DB generates C++ code from OQL query, it automatically

adds **update()** function even when a constant function is called on intention to read only. Due to this, for each objects when retrieved a log record is written. Since, *Shore* allows log size only up to 10 MB and there can be millions of objects that can be retrieved by single query, there is high probability that the system goes out of log space and cripples. But in SDL operations on constant functions does not lead to a log entry. Since ODL is extended to handle constant functions, and λ-DB is modified to generate SDL code without **update()** function call, several functions in the modules that uses **update()** function are modified to call a constant function. The *lambda*-DB code is changed when optimized algebra is translated to corresponding C++ code. The declaration associated with the method is obtained from catalog and if the declaration is constant function, during translation *update* is not written in translated code.

7.3.2 Extension of Query Language

OQL needs to be extended to support queries in which complex objects are passed as parameter to methods. Simple objects are integer, real or string types, which are supported by λ-DB. But complex objects like sequence of another objects or structures cannot be passed as parameter. An example of such query is as follows

Retrieve all the forests that are in State A :
*%for each v in select * from s in States*
where s.name = A
*do select * from f in Forests where f.Inside(v)*

Almost all the benchmark queries except data loading, needs to call functions, with object as its parameter as one above. Details of such queries are found in section 7.4.2. This needs change of code on typechecking and parsing, where it fails. Plan generation however need not to be changed, because the information about the objects and schema is maintained already in database catalog.

Several *signatures* needs to be added to the primitive data types to support benchmark queries. Functions are added to support query 7 and 8 (refer section 7.4.2) of *Sequoia* which are actually query 4 and 5 in this report. Enhancement of query processor is done to support use of spatial index. **Closest** aggregate operator is added to support Paradise queries. But it is still not aggregated with the lambda-DB.

Polygon, polylines are *sequence* of points. But OQL supported by λ-DB, do not support creation of sequence of objects. Till now polygons and polylines are described as *set* of points. OQL is modified to handle creation and queries over *sequence* of objects. The problem is solved by adding the code for transformation into C++.

Spatial indexes although were developed, were not integrated with query processor to make use of it in spatial data access when query is done using OQL. Currently λ-DB is extended to handle join using spatial indices. Also several spatial operations like overlap, disjoint, intersection that joins two types of spatial objects are integrated with *lambda*-DB. Building up rtree index on large data causes log entry which increases in size beyond the limit set by *Shore* and the system cripples. This problem is also valid for btree index creation and hence this needs some change in code in λ-DB where index is created. The specific function is *create_pool_index()*. The code needs to be modified to deal with intermediate commit. *Stream* class and its operation needs to be modified.

7.3.3 Problems in underlying *Shore*

The underlying *Shore* storage manager uses normal unix file system to store data in database. It keeps restriction in size of file system although underlying operating system may allow it. This problem is noticed during loading large data. The size of file system restricts the amount of data file. Only solution to this is to use multiple volumes but λ-DB does not allow storage of data in multiple volumes. Multiple volumes can be added to *Shore* by changing configuration file, read during starting of server. However there can be only single root volume. This is only viewed by λ-DB and hence it is not possible to go beyond one volume currently.

There is incorrect typecasting of *unsigned* variable to *signed* variable. This also attributes to the limitation of number of pages available and in sense reduction of volume size. This problem cannot be solved by just changing the variable delaration, but needs more work on how to handle conversion of *OR*-ing of 4 bytes of two variables where one is *unsigned* and other is *signed*.

Inconsistent handling of *string*s. For genericity purpose λ-DB converts all keys for index that needs to be accessed into *sdl-string*s and sends them to *Shore*. The *sdl-strig* may contain intermediate null value which means termination of it in C++. On comparing with stored index keys, it compares using C++ string comparision operator. In effect the index lookup always failed.

Object creation is not only costly in time but also costly due to space overhead. If created sequence or set of objects *Shore* size bursts up. To meet with this problem objects are created with much lighter weight *structure*s. This also reduced the time for data loading.

Above mentioned problems are never unveiled if the system is not integrated and evaluated. All the functions in the spatial module are tested using small tractable data after integrating it to *Bodhi*. When an object is constructed using OQL through λ-DB, the schema manager in λ-DB takes care of updating system catalog for various purposes like building up index, cost of accessing the object. This counts for some time in constructing object.

7.4 Benchmark Description and System Evaluation

There is no system specific benchmarks which can give proper estimation of *Bodhi*. Moreover the problems are as follows:

- The unavailability of large real dataset de-escalate much of the evaluation process. Synthetic data generation might not produce proper distribution needed for evaluating the system. In this evaluation method some distribution of points are assumed and data is generated.
- The proper selectivity is not defined in case of geo-spatial system in any of the geo-spatial benchmarks. Though benchmarks talk about scaling up of data, there is hardly any restriction how selectivity should change according to scale-up.
- *Sequoia* benchmark queries are relatively simple. Queries are not deeply nested which are notably costly.

But as mentioned earlier, the sole purpose of this evaluation is not to check performance, which can be compromised in this application, but the functionality. Though *Sequoia* 2000 queries are simple, they exploits almost all types of operations possible on spatial data. This leads to the choice of *Sequoia* as benchmark. The reasons why both taxonomic and spatial modules are not tested together are mentioned below:

- There is no specific benchmark, as mentioned earlier by which we could have evaluated the whole system using both taxonomic and spatial modules.
- Unavailability of real data set, where taxonomic data is associated with spatial data.

In following sections the benchmark data and the description of benchmark queries are given.

7.4.1 Benchmark Data

Data for evaluation is generated synthetically by following some distribution. The data types, described before, are defined below:

Point double longitude;
double latitude;
string name; // this field can be NULL

Polygon Sequence⟨Point⟩ points;
//array of points forming boundary
string name;

Network Set⟨Line⟩ lines;
//represents drainage in *Sequoia*
Set⟨Point⟩ points;

Layer Set⟨Polygon⟩ regions; // a set of related polygons Set⟨Point⟩ points;

7.4.2 Query Descriptions

The queries for evaluating the system is *Sequoia* benchmark queries. As described in [MSM93], it covers the range of queries necessary for this field. Followings are the queries of *Sequoia* that are supported by *Bodhi*

Query 1: Create and load the data base and build necessary indices.

Query 2: Select one city based on its name. This query tests whether the system has non-spatial indexes like B^+ tree
%select * from C in Cities where $C.name = Bangalore$;

Query 3: Retrieve all polygons in a new DBMS object that intersect with a specific rectangle. The retrieved polygons are stored as new class of objects called *Layer*. This query needs spatial selection. This query will be faster if there is spatial indexes available.

%Layer FOO2;
% for each v in select * from P in Polygons
where $P.intersect(Rectangle)$ = true;
% retrieve v into FOO2;

Query 4: Select all polygons that lie within a particular distance from a point and that are more than specific area. This query requires combination of spatial and non-spatial selection

%select * from P in Polygons
where $P.Area() < CONST1$ and
$P.isWitninCircle(x, y, radius)$

Query 5: Find all polygons which are nearby a city. A city is nearby if it is within a specified square centered at the city's location. This query performs a complex spatial join of the Points and Polygons data set.

%select * from P in Polygons, p in P.points
where $p.name = Mumbai$ and
$P.isIntersectingBox(x, y, CONST)$;

Query 6: This query is same as *Sequoia* query 10. In this query names of all points within polygons of a specific type are retrieved and put in a new DBMS object. To handle this, like Layer, a new class is defined.

%for each v in select x:p.name
from P in Polygons, p in P.points
where $P.Area() > CONST2$;
%append v.x in new object

Query 7: Retrieve all the river segments from any network those are within some specified distance of a specified geographic point. This is the last query in the benchmark which involves recursion.

%select l from N in Networks, l in N.lines
where $l.isDistanceFrom(Point) < CONST3$

7.4.3 System Evaluation Result

Query	Without index	With R^*-Tree	With HR-Tree
Query 1	13.7 mins	N.A.	N.A.
Query 2	24 sec	N.A.	N.A.
Query 3(selectivity 0%)	26 sec	0 sec	0 sec
Query 3(selectivity 10%)	27 sec	10 sec	12 sec
Query 4	32 sec	16 sec	18 sec
Query 5	158 sec	91 sec	93 sec
Query 6	40 sec.	22 sec	23 sec
Query 7	10 sec.	X	X

Table 7.1: Comparative time taken for benchmark queries with data size about 2 MB
N.A. : Not Applicable
X : time not available as index is not built

B^+-Tree on the names of points are built, using which the access time for Query 2 is reduced to 12.6 seconds where the selectivity of output is10%. The selectivity when reduced to around 1%, the access time reduced to 3 seconds using index. Noticably this reduction is proportional to selectivity as should be.

Query	Selectivity	Without index	With R^*-Tree	With HR-Tree
Query 1	N.A.	51 hrs	N.A.	N.A.
Query 2	20%	1.44 hrs	N.A.	N.A.
Query 3	20%	1.31 hrs	12.6 mins	14 mins
Query 4	14%	1.62 hrs	15.2 mins	18 mins
Query 5	15%	2.51 hrs+1.19 hrs	2.47 hrs+10.4 mins	2.46 hrs+11.8 mins
Query 6	3%	3.56 hrs	22.1 mins	23.4 mins
Query 7	14%	2.39 hrs	23.2 mins	25.9 mins

Table 7.2: Time taken for benchmark queries with data size about 170 MB

From above tables, it is clear that data loading is extremely costly. This is possibly due to the following reasons:

1. The data loading was done across NFS, which inherently slows down disk access rate.
2. In *Shore*, pointer swizzling and unswizzling is the costly operation. Pointer swizzling of stored data to an object increases 3 folds the database access time. This also causes overhead in queries. The problem can be solved if data is loaded from *shore* storage manager level instead of value-added server level.
3. Creation of objects has also overhead.
4. Bulk loading is not

As seen from table 1, the effect of both B^+-Tree and R^*-Tree is not notable except in Query 5, where both of the indices are used. The reason for this possibly extra overhead of index lookup even when the data is in cache. B^+-Tree index over large volume cannot be built, as the maximum log size increases over that restricted by *Shore* and the system crippled. As mentioned above, object creation is not only costly in time but also space. Each object is associated with 16 bytes of logical id and 16 bytes of physical id, which causes overhead on light-weight objects like Points. Moreover, high dimensionality of R^*-Tree causes extra space overhead. However, this high dimensionality is to support MT-index.

In table 2 the addition in Query 5 is to reflect the fact, that first a specific point needs to be found, and then all the polygons meeting some condition depending on the point. The first time is the point retrieval time which has not changed as there is no B^+-Tree index over point names. But the second time representating the polygon access time has noticable change.

7.5 Summary

This chapter dealt with inclusion of spatial modules into the system and evaluating it. In integrating the module, several interface problems are addressed. The importance of adding spatial module to the system is to evaluate the system. Since spatial operations are complex and costly, their inclusion in system may possibly slow down the system operation. The evaluation of the spatial module is done. The specification of system on which *Paradise* and *Bodhi* performed are given below. For further details one can refer [D$^+$94a].

- *Paradise* was tested using a cluster of 17 PCs configured with dual 133 MHz Pentium processors. Solaris 2.5 was used as operating system. Whereas *Bodhi* is evaluated using single Sparc/Solaris 2.5 workstation.
- Since the effort of *Paradise* is to parallelize the geo-spatial system, they need to look for better performance. Unlike them *Bodhi* is built to cater the need for biologists and ecologists. As mentioned earlier in this paper, transaction load in this application is not heavy. So, concentration on performance evaluation is though would have been advantageous, not addressed.

Bibliography

[AGM+90] S. Altschul, W. Gish, W. Miller, E.W. Myers, and D. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, (215), 1990.

[ANZ] Anzmeta dtd version 1.1. http://www.erin.gov.au/database/metadata/anzmeta/anzmeta-1.1.html.

[B+90] Norbert Beckmann et al. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331, 1990.

[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.

[B+98] Tim Bray et al. *Extensible Markup Language Specifications (XML) 1.0.* W3 Consortium, W3C Recommendation, 1998.

[Ber94] Elisa Bertino. Index Configuration in Object-Oriented Databases. *VLDB Journal*, 3(3):355–399, 1994.

[Bio] Biopolymer markup language - bioml. http://www.bioml.com/.

[BKG93] José A. Blakeley, William J. Mc. Kenna, and Goetz Graefe. Experiences building the open oodb query optimizer. In *Proceedings of the ACM SIGMOD Conference*, page 237, May 1993.

[BKS93] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1993.

[BO99] Elisa Bertino and Beng Chin Ooi. The Indispensability of Dispensable Indexes. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):17–27, January 1999.

[Bra92] Marshall Brain. *Motif Programming : The Essentials...and More.* X and Motif Series. Digital Press, 1992.

[BS94] T. Boston and D. Stockwell. Interactive species distribution reporting, mapping and modeling using the world wide web. In *Proc. of Second Intl. WWW Conf.*, 1994.

[C+86] Michael J. Carey et al. The EXODUS Extensible DBMS Project: An Overview. Technical report, University of Wisconsin, 1986.

[C+94] Michael J. Carey et al. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 383–394, 1994.

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

[Cat94] R.G.G. Cattel, editor. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann Publishers, 1994.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proceedings of the ACM SIGMOD Conference*, pages 383–392, June 1992.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[Che00] Madhava Rao Cheethirala. The Spatial Perspective of OSHADHI. Master's thesis, Dept of CSA, Indian Institute of Science, January 2000.

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

[D+94a] David J. DeWitt et al. Client-server paradise. *Proceedings of the 20th VLDB Conference, Santiago, Chile*, 1994.

[D+94b] Asuman Dogac et al., editors. *Advances in Object-Oriented Database Systems*, volume 130 of *F:Computer and Systems Sciences.* NATO ASI Series, 1994.

[DÖBS94] Asuman Dogac, M. Tamer Özsu, Alexandrous Biliris, and Timos Sellis, editors. *Advances in Object-Oriented Database Systems*, volume 130 of *F:Computer and Systems Sciences.* NATO ASI Series, 1994.

[Dow98] Troy Bryan Downing. *Java RMI : Remote Method Invocation.* DG Books Worldwide, 1998.

[DSO78] M. Dayhoff, R.M. Schwartz, and B.C. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5, 1978.

[DTM] R. Durbin and J. Thierry-Mieg. A c.elegans database documentation. ftp://ncbi.nlm.nih.gov/.

[E+98] Robert Eckstein et al. *Java Swing.* O 'Reilly & Associates, 1998.

[EK89] E.Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[Feg94] Leonidas Fegaras. A uniform calculus for collection types. Technical Report 94-030, Oregan Graduate Institute, 1994. Available by anonymous ftp from cse.ogi.edu:/pub/crml/tapos.ps.Z.

[Feg95] Leonidas Fegaras. An algebraic framework for physical oodb design. In *5th International Workshop on Database Programming Languages, Gubbio, Italy*, 1995.

[Feg97a] L. Fegaras. An experimental optimizer for oql. Technical Report TR-CSE-97-007, University of Texas at Arlington, 1997.

[Feg97b] Leonidas Fegaras. An experimental optimizer for oql. Technical Report TR-CSE-97-007, University of Texas at Arlington, 1997. Available at http://www-cse.uta.edu/ fegaras/oqlopt.ps.gz.

[Feg98a] Leonidas Fegaras. A new heuristic for optimizing large queries. In *DEXA*, 1998. Available at http://lambda.uta.edu/order.ps.gz.

[Feg98b] Leonidas Fegaras. Query unnesting in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference*, June 1998.

[FM95] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In *Proceedings of the ACM SIGMOD Conference*, 1995.

[FR89a] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. of the ACM SIGACT/ SIGMOD Symposium on Principles of Database Systems*, 1989.

[FR89b] 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.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), 1998.

[GHJV95a] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters: Elements of Reusable Object-oriented Software.* Addison-Wesley Publishing Company, 1995.

[GHJV95b] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley Longman Inc., 1995.

[Gho00] Amitabh Ghosh. Visual interface for oshadhi. Forthcoming master's thesis, Dept of CSA, Indian Institute of Science, June 2000.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: concepts and techniques.* Morgan kaufmann, San Mateo, CA, 1993.

[GRS] N. Goodman, S. Rozen, and L. Stein. A glimpse at the dbms challenges posed by the human genome project. ftp://genome.wi.mit.edu/pub/papers/Y1994/challenges.ps.Z.

[GRS94] N. Goodman, S. Rozen, and L. Stein. Building a laboratory information system around a c++-based object oriented dbms. In *Proc. of 20th Intl. Conf. on Very Large Databases*, 1994.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences : Computer Science and Computational Biology.* Cambridge University Press, 1997.

[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.

[Güt94a] Ralf Hartmut Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.

[Güt94b] R.H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4), 1994.

[Hwa94] Deborah Jing-Hwa Hwang. *Function-Based Indexing for Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, Cambridge,Massachusetts, feb 1994.

[INB] Inbioparque. http://www.inbio.ac.cr/en/default.html.

[Jag90a] H.V. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1990.

[Jag90b] H.V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD Conference*, pages 332–342, 1990.

[KF94a] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proc. of 20th Intl. Conf. on Very Large Databases*, 1994.

[KF94b] 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.

[KKD89] W. Kim, K. C. Kim, and A. Dale. Indexing Techniques for Object Oriented Databases. *Object Oriented Concepts, Databases, and Applications*, pages 371–394, 1989.

[Kon00a] S. Konidala. Query processing and optimization in oshadhi. Master's thesis, Department of Computer Science & Automation, Indian Institute of Science, January 2000.

[Kon00b] Satheesh Konidala. Query Processing and Optimization in Oshadhi. Master's thesis, Dept of CSA, Indian Institute of Science, January 2000.

[Kon00c] Satheesh Konidala. Query processing and optimization in oshadhi. Master's thesis, Dept. of COmputer Science and Automation, Indian Institute of Science, Jan 2000.

[L+99] Tim Lindholm et al. *The Java Virtual Machine Specification.* Java Series. Addison-Wesley Pub Co., 1999.

[Lia99] Sheng Liang. *The Java Native Interface : Programmer's Guide and Specification.* Java Series. Addison-Wesley Pub Co., 1999.

[LL98a] W. Lee and D.L. Lee. Path dictionary: A new access method for query processing in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(3), May 1998.

[LL98b] Wang-Chien Lee and Dik Lun Lee. Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):371–388, May 1998.

[LL98c] Wang-Chien Lee and Dik Lun Lee. Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):371–388, May 1998.

[LOL92] Chee Chin Low, Beng Chin Ooi, and Hongjun Lu. H-trees: A Dynamic Associative Search Index for OODB. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 134–143, 1992.

[LP85] D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, (227), 1985.

[LR94] M.L. Lo and C.V. Ravishankar. Spatial joins using seeded trees. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[LS90] David B. Lomet and Betty Salzberg. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[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.

[MM98] D.R. Maddison and W. P. Maddison. The tree of life: A multi-authored, distributed internet project containing information about phylogeny and biodiversity. http://phylogeny.arizona.edu/tree/phylogeny.html, 1998.

[MP97a] T.A. Mück and M.L. Polaschek. A configurable type hierarchy index for oodb. *VLDB Journal*, 6(4), 1997.

[MP97b] Thomas A. Mück and Martin L. Polaschek. A Configurable Type Hierarchy Index for OODB. *VLDB Journal*, 6(4):312–332, 1997.

[MS86] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 171–182, 1986.

[MSM93] K. Gardels M. Stonebraker, J. Frew and J. Meredith. The sequoia 2000 storage benchmark. In *Proooceedings of the ACM SIGMOD Conference*, Washington D.C, May 1993.

[N+99] Patrick Naughton et al. *Java2 : The Complete Reference*. Osborne McGraw-Hill, 1999.

[OGI] Geography markup language (gml) v1.0. http://www.opengis.org/techno/specs/00-029.pdf.

[Pan91] R.J. Pankhurst. *Practical Taxonomic Computing*. Cambridge University Press, 1991.

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

[RK95] Sridhar Ramaswamy and Paris C. Kanellakis. OODB Indexing by Class-Division. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 139–150, 1995.

[Saa99] H. Saarenmaa. The global biodiversity information facility: Architectural and implementation issues. Technical Report TR-34, European Environment Agency, 1999.

[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.

[SM97] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[SPE] Species2000. http://www.species2000.org/.

[SS94] B. Sreenath and S. Seshadri. The hcC-tree: An Efficient Index Structure for Object Oriented Databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 203–213, 1994.

[Sta97] Open Group Technical Standard, editor. *Remote Procedure Call*. The Open Group, 1997.

[STM99] L.D. Stein and J. Thierry-Mieg. Acedb: A genome database management system. *Computing in Science and Engineering*, 1(3), 1999.

[SW81] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, (147), 1981.

[Tad00a] Rajarao Tadimeti. Database Habitats for Biological Species: Handling Taxonomy Data in OSHADHI. Master's thesis, Dept of CSA, Indian Institute of Science, January 2000.

[Tad00b] Rajarao Tadimeti. Database habitats for biological species: Handling taxonomy data in oshadhi. Master's thesis, Dept. of Computer Science and Automation, Indian Institute of Science, Jan 2000.

[Val87] Patrick Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[Vas74a] P.C Vasishta. *Taxonomy of Angiosperms*. S. Chand and Co., Delhi, 1974.

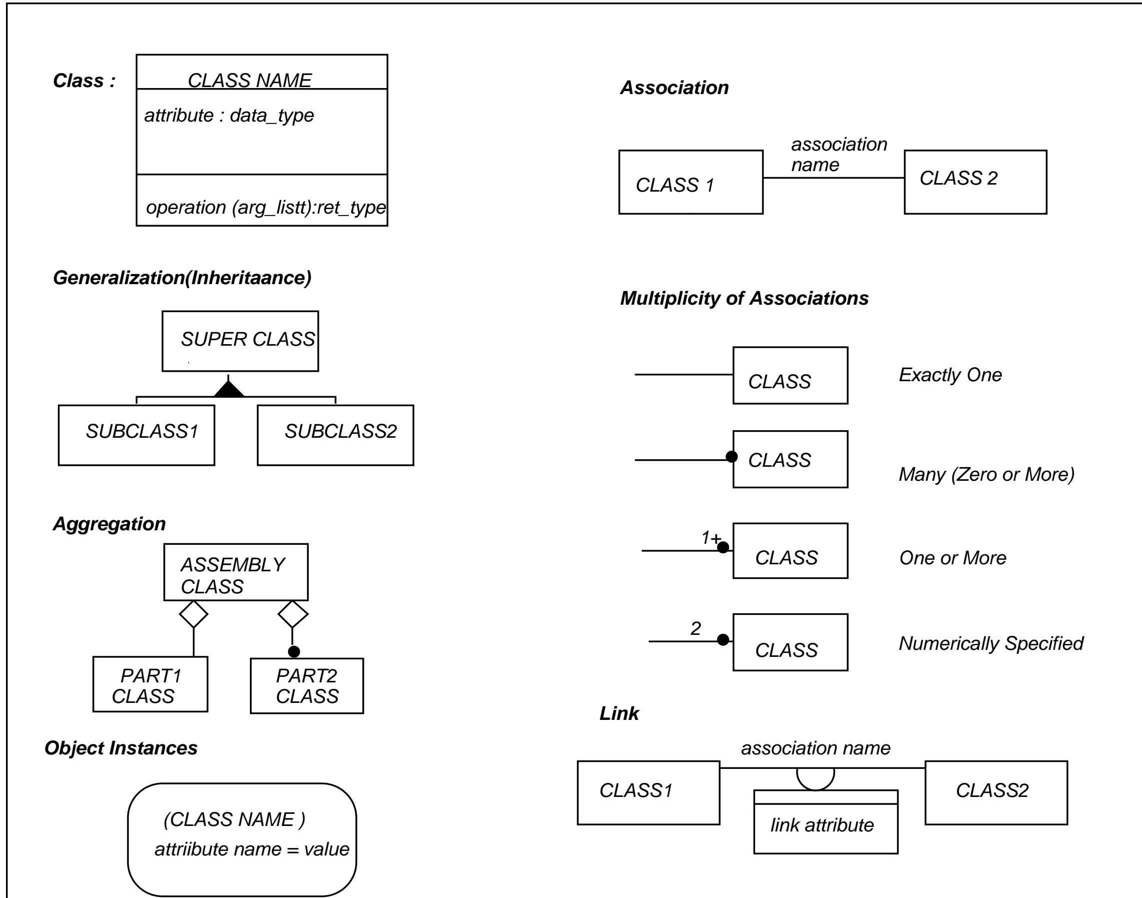
[Vas74b] P.C. Vasistha. *Taxonomy of Angiosperms*. S. Chand and Co., Delhi, 1974.

[VH95] R. Vidya and J. Haritsa. Oshadhi : A biodiversity information system. In *Proc. of Seventh International Conference on Management of Data*, 1995.

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

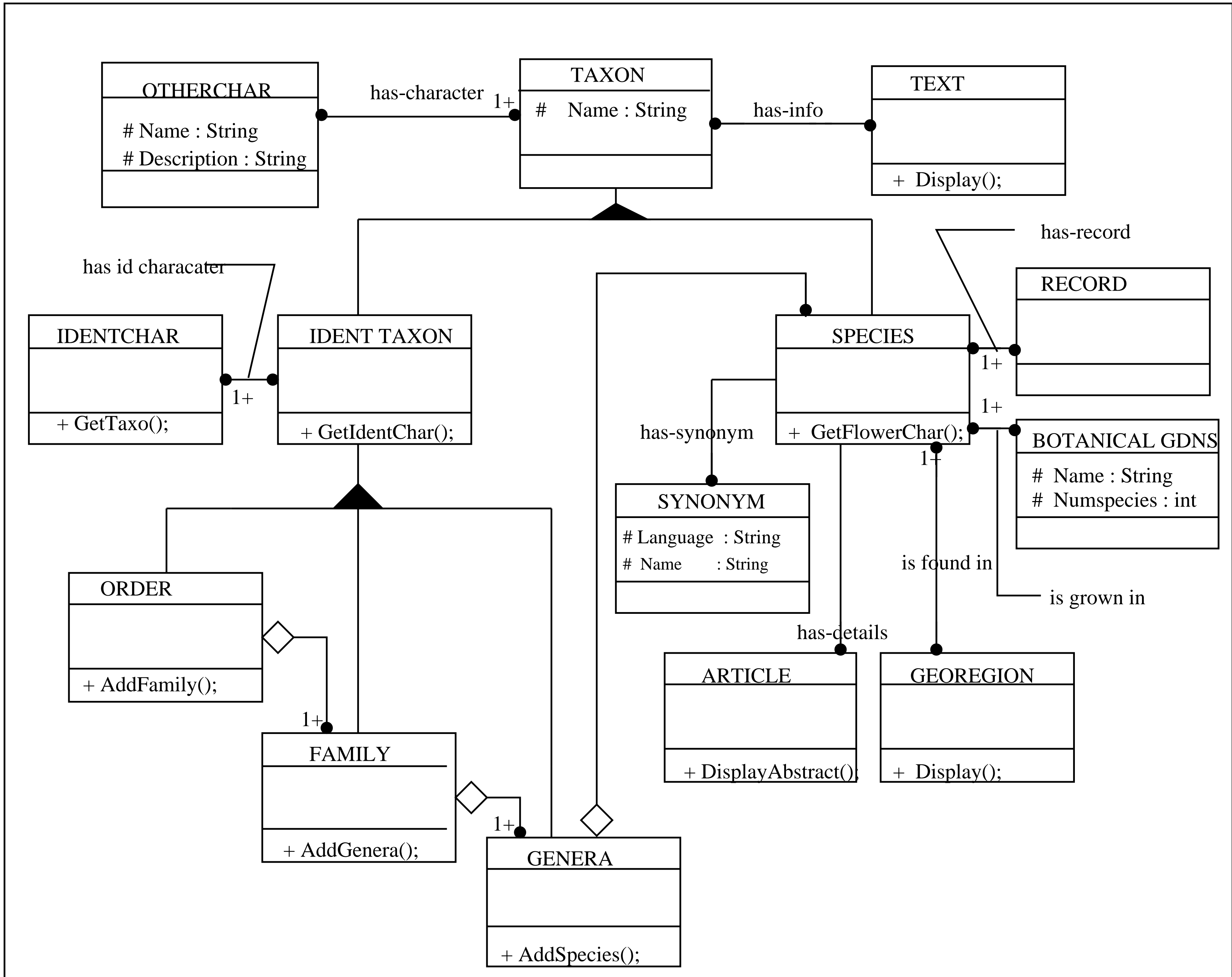
Appendix A

The Rumbaugh Notation

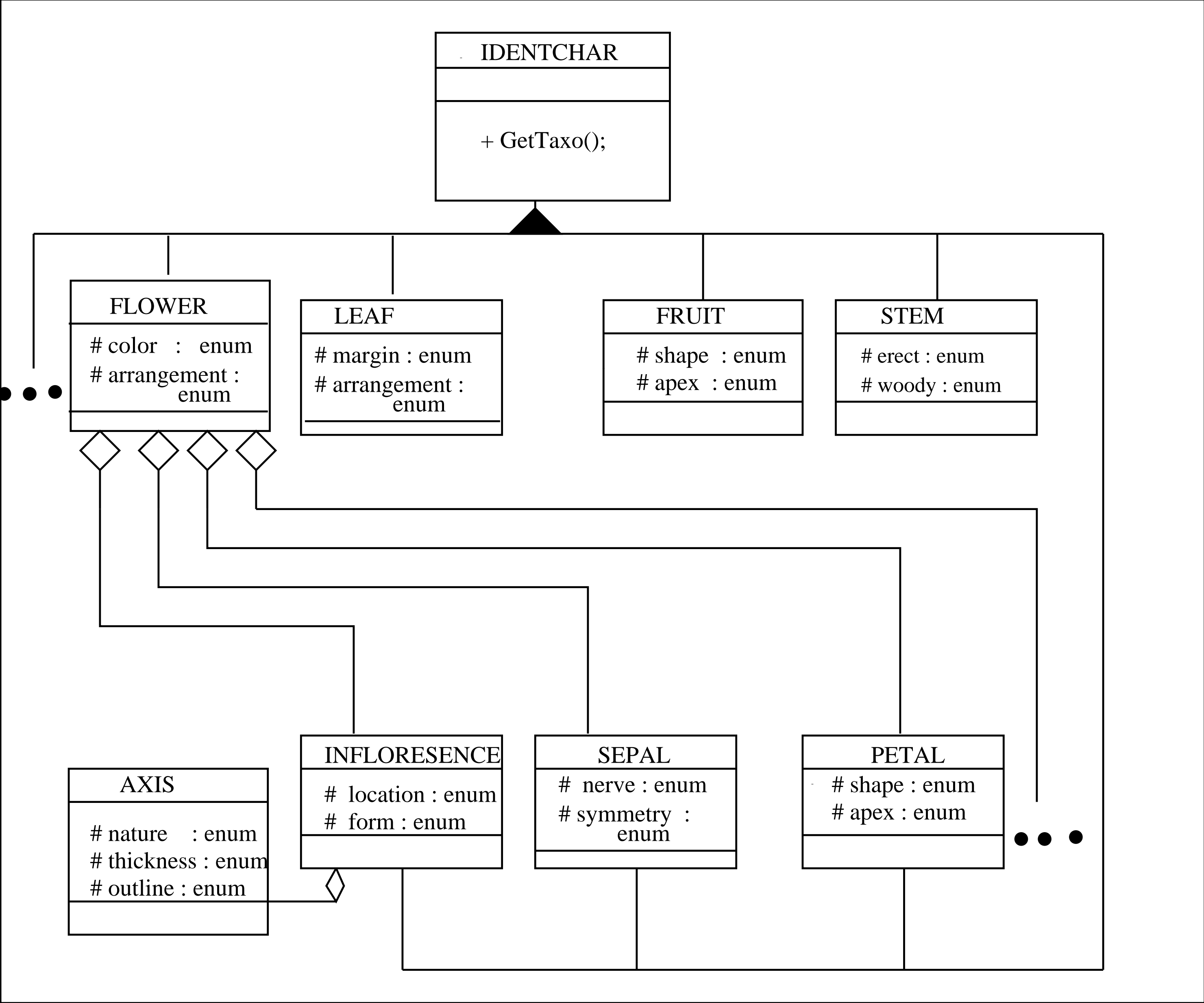


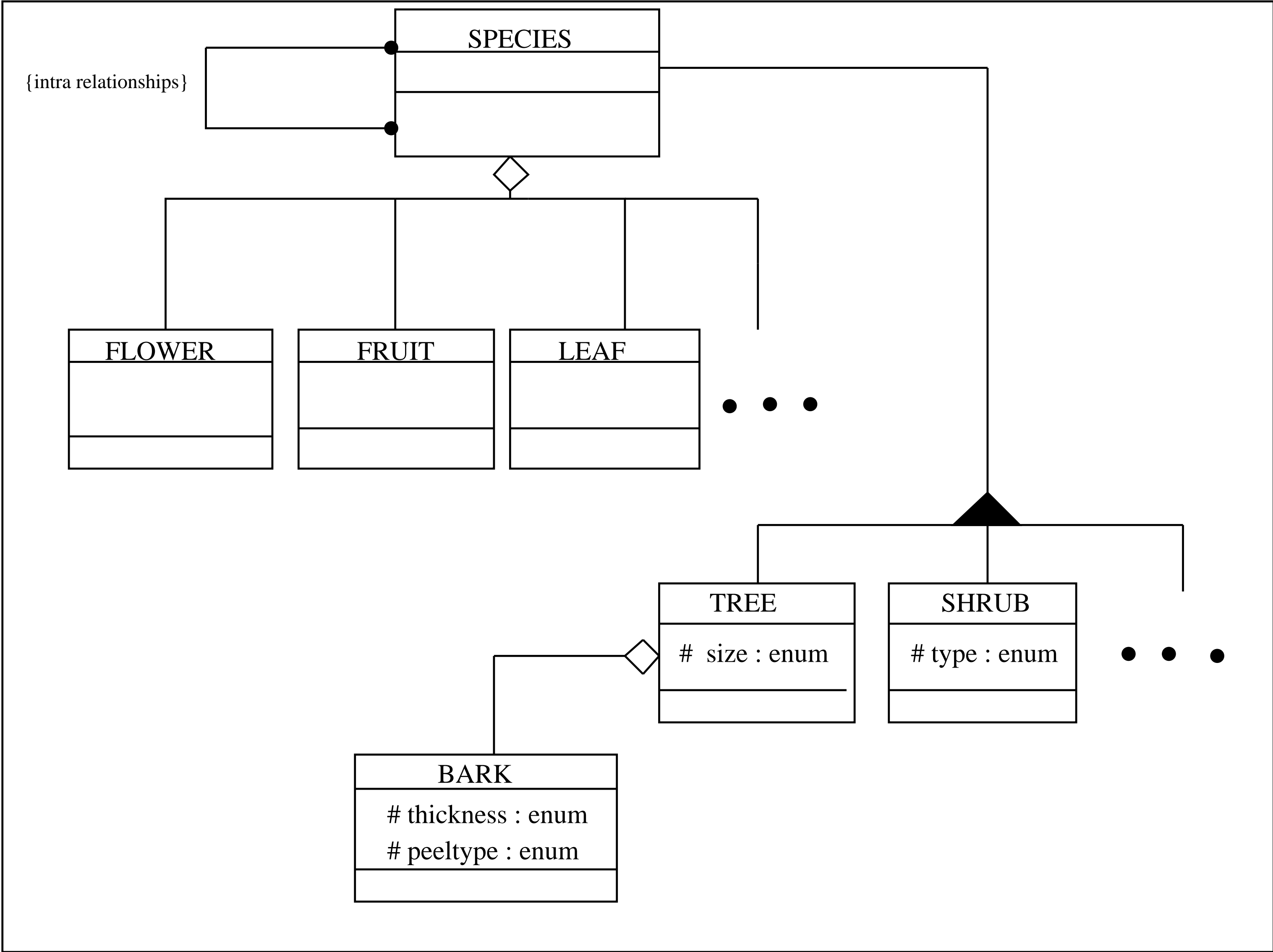
Appendix B

Taxonomy Data Model



OTHERCHAR
Name : String
Description : String
has-character
1+
TAXON
Name : String
has-info
TEXT
+ Display();
has-record
RECORD
has id characater
IDENTCHAR
+ GetTaxo();
1+
IDENT TAXON
+ GetIdentChar();
SPECIES
+ GetFlowerChar();
1+
1+
1+
has-synonym
BOTANICAL GDNS
Name : String
Numspecies : int
SYNONYM
Language : String
Name : String
is found in
is grown in
ORDER
+ AddFamily();
has-details
ARTICLE
+ DisplayAbstract();
GEOREGION
+ Display();
1+
FAMILY
+ AddGenera();
1+
GENERA
+ AddSpecies();


IDENTCHAR
+ GetTaxo();
FLOWER
color : enum
arrangement : enum
LEAF
margin : enum
arrangement : enum
FRUIT
shape : enum
apex : enum
STEM
erect : enum
woody : enum
AXIS
nature : enum
thickness : enum
outline : enum
INFLORESENCE
location : enum
form : enum
SEPAL
nerve : enum
symmetry : enum
PETAL
shape : enum
apex : enum


SPECIES
{intra relationships}
FLOWER
FRUIT
LEAF
TREE
size : enum
SHRUB
type : enum
BARK
thickness : enum
peeltype : enum

Appendix C

Taxonomy Data Model in ODL

The following are the class definitions of some of the major classes in the taxonomy data model of OSHADHI. The definitions are given in ODL.

```
class TaxonomyLevel{
  protected:
    attribute string strName;
    relationship set<SpecialChar> stCollSpecial;
    relationship Text  refTaxInfo;

  public:
    void initialize( in string ilrefcName );
    const string getName() ;
    void addSpecialChar( in SpecialChar irefSpChar );
    void deleteSpecialChar( in SpecialChar irefSpChar );
    void modifySpecialChar( in string istrCharName,
                  in SpecialChar irefSpChar );
    void addText( in string istrText );
  };

class IdentLevel : TaxonomyLevel{
  protected:
    relationship set< IdentChar > stCollIdentChar
             inverse IdentChar::stCollIdentLevel;
    relationship set< Habit > stCollHabit;

  public:
    void addIdentChar( in IdentChar irefIdentChar );
    void delIdentChar( in IdentChar irefIdentChar );
    const IdentChar  getIdentCharElement( in long iiElementIndex ) ;
    void addHabit( in HABIT iHabit );
    const Habit  getHabitElement( in long iiElementIndex ) ;
    void deleteHabit( in  Habit  irefHabit );
  };

class Species : TaxonomyLevel{
  public:
    relationship set<Botanical> stBotanical;
    relationship set<Record> stRecord;
    relationship set<Synonym> stSynonym;
    relationship FlowerChar refFlowerChar
                          inverse FlowerChar::stCollSpecies;
    relationship FruitChar refFruitChar
                         inverse FruitChar::stCollSpecies;
    relationship LeafChar refLeafChar
                        inverse LeafChar::stCollSpecies;
    relationship SeedChar refSeedChar
                        inverse SeedChar::stCollSpecies;
    relationship HabitChar refHabitChar
```

```
                        inverse HabitChar::stCollSpecies;
relationship BranchChar refBranchChar
                         inverse BranchChar::stCollSpecies;
relationship FoliageChar refFoliageChar
                          inverse FoliageChar::stCollSpecies;
relationship CanopyChar refCanopyChar
                         inverse CanopyChar::stCollSpecies;
relationship BarkChar refBarkChar
                       inverse BarkChar::stCollSpecies;
relationship StemChar refStemChar
                       inverse StemChar::stCollSpecies;
relationship Genera refParentGenera;

const  Genera getGenera() ;

void addBotanical( in Botanical irefNewBotanical );
void deleteBotanical( in string istrBotanicalName );
void addRecord( in Record irefRecord ) ;
void deleteRecord( in string istrRecordName );
void addSynonym( in Synonym irefSynonym);
void deleteSynonym( in string istrSynonymName);

FlowerChar setFlowerChar( in FlowerChar irefFlowerChar );
const  FlowerChar getFlowerChar( ) ;
void deleteFlowerChar( in FlowerChar irefFlowerChar );

HabitChar setHabitChar( in HabitChar irefHabitChar );
const  HabitChar getHabitChar( ) ;
void deleteHabitChar( in HabitChar irefHabitChar );

LeafChar setLeafChar( in LeafChar irefLeafChar );
const  LeafChar getLeafChar( ) ;
void deleteLeafChar( in LeafChar irefLeafChar );


BranchChar setBranchChar( in BranchChar irefBranchChar );
const  BranchChar getBranchChar( ) ;
void deleteBranchChar( in BranchChar irefBranchChar );


BarkChar setBarkChar( in BarkChar irefBarkChar );
const  BarkChar getBarkChar( ) ;
void deleteBarkChar( in BarkChar irefBarkChar );

FoliageChar  setFoliageChar( in FoliageChar irefFoliageChar );
const  FoliageChar  getFoliageChar( ) ;
void deleteFoliageChar( in FoliageChar irefFoliageChar );

CanopyChar  setCanopyChar( in CanopyChar irefCanopyChar );
const  CanopyChar  getCanopyChar( ) ;
void deleteCanopyChar( in CanopyChar irefCanopyChar );

StemChar  setStemChar( in StemChar irefStemChar );
const   StemChar  getStemChar( ) ;
void deleteStemChar( in StemChar irefStemChar );

SeedChar setSeedChar( in SeedChar irefSeedChar );
const   SeedChar getSeedChar( ) ;
void deleteSeedChar( in SeedChar irefSeedChar );

FruitChar setFruitChar( in FruitChar irefFruitChar );
const   FruitChar getFruitChar( ) ;
void deleteFruitChar( in FruitChar irefFruitChar );
```

```
  };

class Genera : IdentLevel{
  protected:
    relationship set< Species > stCollSpecies;
    relationship Family  refParentFamily
                         inverse Family::stCollGenera;

  public:
    Species  addSpecies (in string istrSpeciesName );
    void deleteSpecies (in string istrSpeciesName );
    const Species  getSpeciesElement( in long iiElemIndex ) ;

    Family setParentFamily( in Family  irefParentFamily );
    const Family getParentFamily( ) ;
  };

class Family : IdentLevel{
  protected:
    relationship set< Genera > stCollGenera
                 inverse Genera::refParentFamily;

    relationship Order  refParentOrder
                        inverse Order::stCollFamily;

  public:
    Genera  addGenera( in string istrGeneraName );
    void deleteGenera( in string istrGeneraName );
    const Genera  getGeneraElement( in long iiElemIndex ) ;
    Order  setParentOrder( in Order  irefParentOrder );
    const Order  getParentOrder() ;
  };

class Order : IdentLevel{
  protected:
    relationship TaxonDatabase refTaxonDatabase
                               inverse TaxonDatabase::stCollOrder;

    relationship set< Family > stCollFamily
                               inverse Family::refParentOrder;

  public:
    Family  addFamily( in string istrFamilyName );
    void deleteFamily( in string istrFamilyName );
    const Family  getFamilyElement( in long iiElemIndex ) ;
  };


 class IdentChar{
  protected:
    relationship set<TaxonomyLevel> stCollTaxLevel;
    relationship set<IdentLevel> stCollIdentLevel
                                 inverse IdentLevel::stCollIdentChar;

  public:

    const TaxonomyLevel getTaxElement(in long iiElemIndex ) ;
    void addTaxLevel(in TaxonomyLevel irefTaxLevel );
    const CHARTYPE getType() ;
  };

class SepalChar : IdentChar{
  protected:
```

```
    relationship FlowerChar refFlower
                            inverse FlowerChar::refSepal;

  public:
    attribute NerveChar enumNerveChar;
    attribute SymmetryChar enumSymmetryChar;
    attribute Form enumForm;
    attribute FusionChar enumFusionChar;
    attribute Color enumColor;
    attribute long length;
    attribute long width;
    attribute long number;
    void initialize();
    const string getHashValue() ;
    const CHARTYPE getType() ;
  };

// Other identifying characteristics are not listed here

class SpecialChar{
  protected:
    attribute string strName;
    attribute string strDescription;

  public:
    void initialize ( in string ilrefcName );
    void setDescription (in string ilrefcDesc );
    const string getDescription() ;
    const string getName() ;
  };

class Word{
  protected:
    attribute string strWord;

  public:
    void initialize( in string  ilrefcWord );
     const string  getString() ;
  };

class Text{
  protected:
    relationship bag<Word> bgCollWord;

  public:
    void initialize();
    void addWord(in string irefWordString );
    const Word  getWordElement(in long iiElemIndex ) ;
  };

class TaxonDatabase{
  protected:
    attribute TaxonIndex refTaxonIndex ;
    relationship set<Order> stCollOrder
                            inverse Order::refTaxonDatabase;

 public:
    const   Order searchOrder( in string istrOrderName ) ;
    const   Family searchFamily( in string istrFamilyName ) ;
    const   Genera searchGenera( in string istrGeneraName ) ;
    const   Species searchSpecies( in string istrSpeciesName ) ;
    Order addOrder( in string istrOrderName );
    Family addFamily( in string istrOrderName,
```

```
                    in string istrFamilyName);
  Genera addGenera( in string istrFamilyName,
                    in string istrGeneraName );
  Species addSpecies( in string istrGeneraName,
                      in string istrSpeciesName );
  void deleteOrder( in string istrOrderName );
  void deleteFamily( in string istrFamilyName );
  void deleteGenera( in string istrGeneraName );
  void deleteSpecies( in string istrSpeciesName );
  const   Order getOrderElement( in long iiElementIndex) ;
  void initialize( in TaxonIndex irefTaxonIndex);
};
```

Appendix D

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 E

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 4.1. These are implemented as the member functions of the spatial data classes as shown in appendix D. This appendix discusses all those member functions with appropriate example queries.

E.1 Inside Queries

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

E.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'.

E.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'.

E.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'.

E.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'.

E.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.

E.2.1 IsLineInstersectingLine

query: Is line 'line1' intersecting the line 'line2'.
This query finds if the two given lines are intersecting.

E.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.

E.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.

E.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.

E.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.

E.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'.

E.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.

Appendix F

DDL of Oshadhi in BNF

```
<Specification>::=   (<Definition>)+ ;
<Id>::= <Identifier>;
<Definition>::=(<TypeDcl>|<ConstDcl>|<ExceptDcl>|<Interface> |<Module>|<Class>);
<Class>::= <ClassHeader> ’{’<InterfaceBody>’}’

<ClassHeader>::= CLASS <Id> [(EXTENDS <ScopedName>) | (:<InheritanceSpec>) ]
                    [<TypePropertyList>]
     | CLASS <Id> : EXTENDS <ScopedName> : <InheritanceSpec>
                    [<TypePropertyList>];

<Module>::= MODULE <Id> ’{’ <Specification> ’}’;

<Interface>::=<InterfaceDcl> | <ForwardDcl>;
<InterfaceDcl>::= INTERFACE <Id> [: <InheritanceSpec> ] ’{’ [<InterfaceBody>]’}’


<ForwardDcl>::= INTERFACE <Id>;

<TypePropertyList>::=  ’(’ [ EXTENT <Id> ]  [ [ KEY | KEYS ] <Key>{,<Key>} ’)’

<Key>::= <Id> | ’(’ <Id> {,<Id> } ’)’;      //attributes list as keys

<AttributeNameList>::= <Id>{,<Id>};
<InterfaceBody>::= (TypeDcl |ConstDcl |ExceptDcl |AttrDcl |RelDcl
|OpDcl);[<InterfaceBody>];
<InheritanceSpec>::=<ScopedName> {,<ScopedName>};
<ScopedName>::=   <Id>| :: <Id> | <ScopedName> :: <Id>;
<ConstDcl>::=CONST <ConstType>  <Id> EQUAL <ConstExp>;
<ConstType>::= <IntegerType> | CHAR | BOOLEAN | FLOAT | DOUBLE
                            |<StringType>|<ScopedName>;

<ConstExp>::=<OrExpr>;
<OrExpr>::=<ShiftExpr>{ (’|’ | ^ | &  ) <ShiftExpr> };
<ShiftExpr>::=<AddExpr>{ ( ’>>’ | ’<<’ ) <AddExpr>};
<AddExpr>::= <MultExpr>{(’+’ |’-’ ) <MultExpr> };
<MultExpr>::= <UnaryExpr> { ( TIMES | SLASH | PERCENT ) <UnaryExpr>};
<UnaryExpr>::= [’-’ | ’+’ | ~ ] <PrimaryExpr>;
<PrimaryExpr>::=<ScopedName>|<Literal> |  ’(’ <ConstExp> ’)’;
<Literal>::=<IntegerLiteral>|<StringLiteral>|<CharacterLiteral>|
                           <FloatingPtLiteral>  |<BooleanLiteral>;
<BooleanLiteral>::=   TRUE |  FALSE;
<TypeDcl>::=TYPEDEF <TypeDeclarator>| <StructType>| <UnionType>|<EnumType>
<TypeDeclarator>::= <TypeSpec> <Declarators>
<TypeSpec>::=   <SimpleTypeSpec> | <ConstrTypeSpec>;
<SimpleTypeSpec>::=  <BaseTypeSpec> | <TemplateTypeSpec> | <ScopedName>;
<BaseTypeSpec>::= <IntegerType> |FLOAT |DOUBLE | CHAR|BOOLEAN|OCTET |ANY
              | DATE | TIME | INTERVAL  | TIMESTAMP;
```

```
<TemplateTypeSpec>::= <ArrayType> | <StringType>| <CollType>;
<CollType>::= ( SET |LIST | BAG ) '<' <SimpleTypeSpec> '>'
          | DICTIONARY '<' SimpleTypeSpec COMMA SimpleTypeSpec '>'

<ConstrTypeSpec>::=  <StructType> |<UnionType>| <EnumType>
<Declarators>::=   <Declarator> {, <Declarators> }
<Declarator>::= <Id> | <ArrayDeclarator>
<IntegerType>::=  [  UNSIGNED  ]   ( LONG | SHORT )
<StructType>::= STRUCT <Id> '{' (Member)+ '}'
<Member>::=<TypeSpec> <Declarators>;
<UnionType>::=UNION <Id> SWITCH '(' <SwitchTypeSpec> ')' '{' <SwitchBody>'}'
<SwitchTypeSpec>::= <IntegerType>|CHAR | BOOLEAN| <EnumType> | <ScopedName>
<SwitchBody> ::= <Case> [<SwitchBody>]
<Case> ::= <CaseLabelList>  <ElementSpec>;
<CaseLabelList>::= <CaseLabel>,{CaseLabel}
<CaseLabel>::= CASE <ConstExp> : | DEFAULT : ;
<ElementSpec> ::= <TypeSpec> <Declarator>;
<EnumType> ::= ENUM <Id> '{' <Id> {,<Id> } '}';
<ArrayType>::= (SEQUENCE | ARRAY) '<' <SimpleTypeSpec> [ COMMA <ConstExp>] '>'
<StringType>::= ODL_STRING  [  '<' <ConstExp> '>']
<ArrayDeclarator>::=<Id> '[' <ConstExp> ']'  {, '[' <ConstExp> ']'  };


<AttrDcl>::=  [ READONLY ] ATTRIBUTE <DomainType> [ '[' <ConstExp> ']' ]
                         <AttributeNameList>
DomainType::= <SimpleTypeSpec> | <StructType> | <EnumType>
           | ( SET | LIST | BAG | ARRAY ) '<' <Literal> '>'


<RelDcl>::= RELATIONSHIP <TargetOfPath> <Id> [ INVERSE <Id> :: <Id> ] <OptOrderBy>
<TargetOfPath>::=   <Id> | (SET |LIST |BAG | ARRAY ) '<' <Id> '>'
<OptOrderBy>::= [  '{' ORDER_BY <ScopedName>,{<ScopedName>} '}' ]
<ExceptDcl>::= EXCEPTION <Id> '{' (<Member>)+ '}'
<OpDcl>::= [ONEWAY] <OpTypeSpec> <Id> <ParameterDcls>
                     [ <RaisesExpr> ] [<ContextExpr>]
<OpTypeSpec>::= <SimpleTypeSpec> | VOID
<ParameterDcls>::='(' [ <ParamDcl> {,<ParamDcl>} ] ')'
<ParamDcl>::= (IN | OUT | INOUT ) <SimpleTypeSpec> <Declarator>
<RaisesExpr>::=  RAISES '(' <ScopedName> {,<ScopedName>} ')'
<ContextExpr>::=  CONTEXT '(' <StringLiteral> {,<StringLiteral>} ')'
```

Appendix G

Query Language of Oshadhi in BNF

```
<program> ::= <(C_plus_plus_code>)+
<Id>:identifier
<ctoken_list> ::= (CTOKEN)+;
<C_plus_plus_code> ::= % <oql>;|'{' <program> '}'| CTOKEN |;
<foreach_body>::= <ctoken_list>|% <oql>| '{' <program> '}';
<oql> :<define_query>| MODULE <Id> | <Id> <Id> | <Id> := <query>
    | <destination> := <query>
    | <destination> INSERT <query>
    | <destination> DELETE <query>
    | FOR EACH <Id> IN <query> DO <foreach_body>
    | CREATE INDEX <Id> ON <Id> '(' <id_list> ')'
    | DROP INDEX <Id>
    | COLLECT STATISTICS
    | PRINT STATISTICS
    | BEGIN
    | COMMIT
    | ABORT
    | INITIALIZE
    | INITIALIZE '(' <Id> , <Id>  ')'
    | CLEANUP

<destination> ::= <query>.<Id>
<define query> ::= DEFINE <Id>[( [<Id> {,<Id>}])] AS <query>
<query> ::= NIL |  TRUE |  FALSE |
             <integer_literal> |
              <float_literal> |
              <string_literal> |
              <Id> | ( <query> )

//Simple Expressions
<query> ::=  <query> (+|-|*|/|MOD) <query>|
                - <query>|
                ABS( <query> ) |
                <query> || <query>

//Relational  Expressions
   <query> ::= <query> ( = | != | '<' | '<=' |'>' | '>=' ) <query>
<query> ::= <query> LIKE <string_literal>

//Boolean Expression
<query> ::= NOT <query>
        <query> ::= <query> (AND | OR ) <query>

//Constructor
<query> ::= <Id> '(' [<query>] ')'
        <query> ::= <Id> '(' <Id> : <query> {, <Id> : <query>} ')'
        <query> ::= STRUCT '(' <Id> : <query> {, <Id> : <query>} ')'
```

```
        <query> ::= (SET | BAG | LIST ) '(' [<query>{,<query>} ] ')'
        <query> ::= '(' <query>, <query> {,<query>} ')'
        <query> ::= [LIST] '(' query .. query')'
        <query> ::= ARRAY '(' [query {, query} ] ')'

//Accessor
<query> ::= <query> . <Id>
<query> ::= <query> . <Id>( <query> {,<query>} )
        <query> ::= * <query>
<query> ::= <query> '[' <query> ']'
        <query> ::= <query> '['<query>:<query>']'
        <query> ::= first'('<query> ')'
        <query> ::= last'('<query>')'
<query> ::= <Id>'(' [<query> {, <query>}] ')'

//Collection Expression
   <query> ::= FOR ALL <Id> IN <query> : <query>
        <query> ::= EXISTS <Id> IN <query> : <query>
     <query> ::= ( EXISTS | UNIQUE | COUNT |SUM | MIN | AVG ) '(' <query> ')'
     <query> ::= <query> IN <query>
     <query> ::= <query>
     (EQUAL | NOTEQUAL | GREATER | LESS | GREATEREQUAL | LESSEQUAL)
     ( SOME | ALL | ANY ) <query>
     <query> ::= COUNT '(' * ')'
//Select Expression
<query> ::= SELECT [ DISTINCT ] <projection_attributes>
    FROM <variable_declaration> {, variable_declaration }
    [ WHERE <query> ]
                    [ GROUP BY <partition_attributes> ]
    [ HAVING <query> ]
                    [ ORDER BY <sort_criterion> {,<sort_criterion>} ]
   <projection_attributes> ::= <projection> {, <projection>} | *
   <projection> ::= <query>
<projection> ::= <Id> : <query>
<projection> ::= <query> AS <Id>
<variable_declaration> ::= <query> [[AS] <Id>]
<partition_attributes> ::= <projection> {, <projection>}
<sort_criterion> ::= <query> [ASC | DESC ]

//Set Expression
<query> ::= <query> (INTERSECT  |UNION  |EXCEPT   )<query>

        <query> ::= ( LISTTOSET | ELEMENT | DISTINCT | FLATTEN ) '(' <query> ')'
        <query> ::= '(' <Id> ')' <query>
```

Appendix H

Meta-Data Export format in BNF

```
<Export-Format> := <className> "(" <Export-Dcl> { "," <Export-Dcl> } ")"
<Export-Dcl> ::= <attrib-Dcl> | <relationship-Dcl> | <method-Dcl>
<attrib-Dcl> ::= attribute "(" <attrName> , <attrType> ")"
<method-Dcl> ::= method "(" <methName> , <returnType> ,
                     params "(" <params-Dcl> { , <params-Dcl> } ")" ")"
<params-Dcl> ::= ( in | out | inout ) "(" <attrName> , <attrType> ")"
<attrType>   ::= <Identifier>
<methodType> ::= <Identifier>
<className>  ::= <Identifier>
<methName>   ::= <Identifier>
<Identifier> :=  [a-zA-Z][a-zA-Z0-9_]*
```

Appendix I

Remote Method Invocation

Remote Method Invocation (RMI) provides a means of communication between Java applications using normal method calls, and offers the capability for the aplications to run on separate computers - located perhaps as far as on opposite sides of the globe.

One important feature of RMI is that it presents a programmatic interface for networking rather than relying on the sockets and streams approach. The method's major advantage is that it offers you a higher-level, method based interface in which a remote object is treated as though it were local. RMI is also convenient to use and more natural in many ways thatn using sockets. A requirement, however, is that you must run Java on both ends of the network connection-rather than simply both systems implement the same networking protocol.

Here are the steps for building a working RMI application:

1. Develop the remote object, server, and client code
2. Compile the code to generate the class files
3. Run the RMI compiler (rmic) to generate the stub and skeleton
4. Move the .class files to an appropriate location
5. Start the RMI registry
6. Start the server
7. Start the client

In the following paragraphs, the RMI framework, the architecture and the way in which RMI works is explained through an example code.

I.1 Architectural Overview

The RMI system consists of three layers:

- The stub/skeleton layer - client-side stubs (proxies) and server-side skeletons
- The remote reference layer - remote reference behaviour (such as invocation to a single object or to a replicated object)
- The transport layer - connection set up and management and remote object tracking

The application layer sits on top of the RMI system. The relationship between the layers is shown in the following figure.

A remote method invocaion from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote objest as a conduit to the remote object. A client held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote reference layer. Stubs are generated using the rmic compiler.

The remote reference layer is responsible for carrying out the semantics of the invocation. For example, the remote reference layer is responsible for determining whether the server is a single object or is a replicated object requiring communication with multiple locations. Each remote

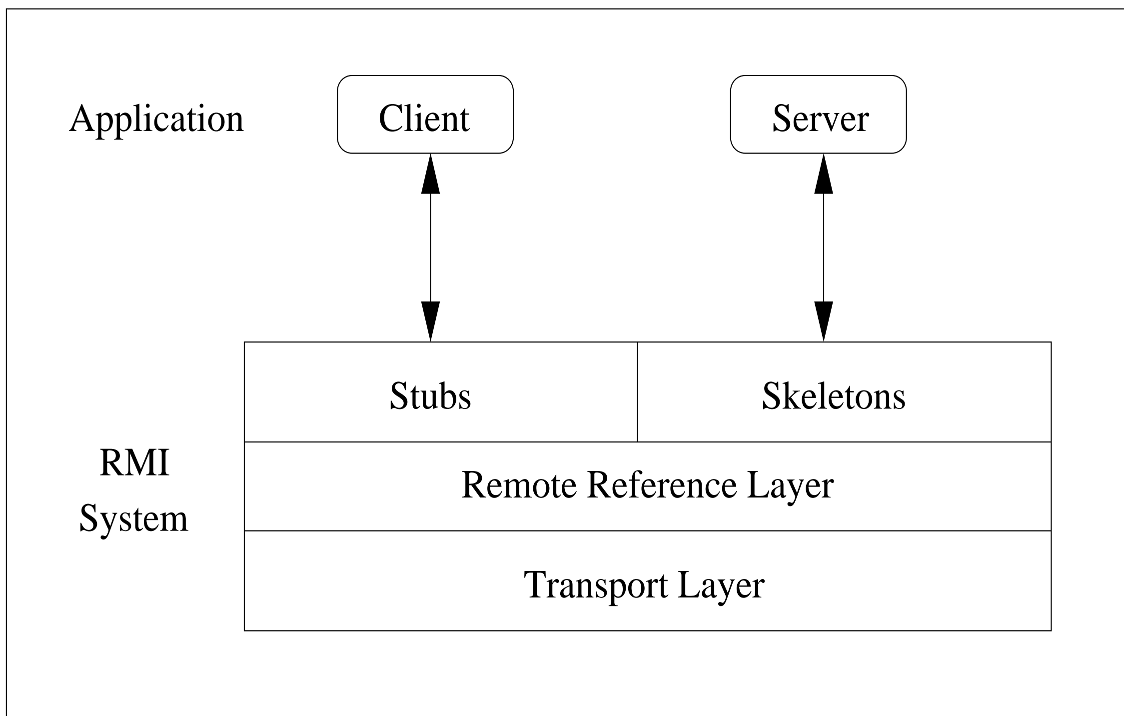


Figure I.1: RMI system architecture

object implementation chooses its own remote references semantics - whether the server is a single object or is a replicated object requiring communications with its replicas.

Also handled by the remote reference layer are the reference semantics for the server. The remote reference layer, for example, abstracts the different ways of referring objects that are implemented in (a) servers that are always running on some machine, and (b) servers that run only when some method invocation is made on them (activation). At the layers above the remote reference layer, these differences are not seen.

The transport layer si responsible for connection setup, connection management, and keeping track of and despatching to remote objects (the targets of remote calls) residing in the transport's address space.

In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behaviour that needs to occur before handling off the request to the server-side skeleton. The skeleton for a remote object makes an up call to the remote object implementation which carries out the actual method call.

The return value of the call is sent back through the skeleton, remote reference layer, and transport on the server side, and then up through the transport, remote reference layer, and stub on the client side.

I.2 How RMI works

With RMI, client and server programs are running, with both using Java. You specify a Java interface to describe each object to be shared remotely, and enumerate the public methods to be invoked on the object. A server will use the interface and create objects that implement it and that are also registered with a name registry service using a URL-based naming scheme.

A client will contact the registry, attempt to retrieve an object by name, and obtain a remote reference to it. You handle method invocation and data passing via skeleton and stub classes that are generated automatically through an RMI compiler from user code, and through communication using object serialization and TCP/IP.

I.2.1 Develop Remote Object Code

RMI supports Java objects communicating via their methods, regardless of where the objects are located. The first step in creating a class to be accessed remotely is to define an interface describing

the methods that are available remotely. Following is a code fragment for the remote interface that we have used in our source code:

```
// Hello.java
...
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Hello extends Remote
{
    String sayHello(String queryString) thrown RemoteException;
    String[] ReadFile(String filename) throws RemoteException;
}
```

java.rmi.Remote is an empty marker interface that indicates that an object is remote, so class objects implementing Hello are marked as being remote references. All the methods in a remote interface must declare that they can throw an exception of type *java.rmi.RemoteException*, which gets thrown whenever a remote method invocation fails.

I.2.2 Develop the Server Code

Once you specify the remote object definition via an interface, the enxt step is to develop the server code. The server implements the objects interface and creates instances of the object to be shared remotely. Here we present a fragment of the source code:

```
// HelloImpl.java
...
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello {
    public String Format[] = new String[500];
    String jni_message;

    public HelloImpl() throws RemoteException {
        super();
    }
    public String sayHello(String queryString) {
        try {
            HelloImpl obj2 = new HelloImpl();
            ....
        }
        catch (Exception e) {
            System.out.println ("Exception..!!");
        }
        ...
    }
    public String[] ReadFile(String filename) {
        try {
            BufferedReader stdin = new BufferedReader(new
                 InputStreamReader (new FileInputStream(filename)));
            String in = "";
            int i = 0;
            try {
                while ((in = stdin.readLine()) != null) {
                    Format[i] = new String(in);
                    System.out.println (Format[i++]);
                }
            }
            catch (Exception e) {}
        }
        catch (FileNotFoundException fe) {
            System.out.println ("File not found!");
```

```
        }
        return Format;
    }
    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            HelloImpl obj = new HelloImpl();
            Naming.rebind("//144.16.79.148/HelloServer", obj);
            System.out.println("HelloServer bound in registry");
        }
        catch (Exception e) {
            System.out.println("HelloImpl err: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

HelloImpl extends *java.rmi.server.UnicastRemoteObject* and implements *Hello*. The first of these classes supplies soem of the basic properties needed for the remote objects, and the second defines the methods to be made available remotely.

I.2.3 Setting up a security manager

The tricky part of this code is what occurs in *main*. The first thing is to establish a security manager. RMI involves loading remote .class files, something like a web browser does with an applet, but this opeartion entails some security risk. If a security manager is not installed, teh default is to load only local files, and RMI by definition cannot work with this restriction. So a security manager has to be set to make explicitly clear that loading remote .class files is acceptable.

An instance of *Hello* then gets created and registered with the name registry service using *Naming.rebind*. This process associates an object reference with a specific name so that the client can look up the object by name through the registry. To actually invoke the remote method, the server and the client use skeleton and stub entities to communicate. These .class files are generated from the server .class files by the RMI compiler described below.

Conceptually, the stub class is:

```
public class HelloImpl_Stub extends java.rmi.server.RemoteStub
                implements Hello, java.rmi.Remote { ... }
```

and the skeleton class is:

```
public class HelloImpl_Skel extends java.lang.Object
                implements java.rmi.server.Skeleton { ... }
```

Usage like: *javap -c HelloImpl_Stub* displays the bytecodes and illustrates what goes on behind the scenes.

The stub is a surrogate for the remote object, and the skeleton is a server-side entity that dispatches calls to the remote object implementation. The stub takes care of collecting (or marshalling) the data and transmitting/receiving it on the client side. The skeleton performs similarly on the server side. Object serialization changes the data into a stream of bytes, which are transmitted via TCP/IP.

I.2.4 Develop the Client Code

```
// QueryView.java
...
public void actionPerformed (ActionEvent evt) {
    ...
    try {
        obj = (Hello)Naming.lookup("//" + M + "/HelloServer");
        message = new String(obj.sayHello(queryString));
```

```
            System.out.println ("Message from JNI & RMI: " + message);
            new DisplayResults(message, M,
                   http://144.16.79.148/~amitabh/classes/cashew.htm");
            System.out.println("\nJNI & RMI over and Query end\n");
        }
        catch (Exception e) {
            System.out.println("HelloApplet Exception: " +
                                            e.getMessage());
        }
    }
```

Once the server is implemented, implementing the client is straightforward. The security considerations for the client are the same as for the server. A URL is formed to contact the name registry, and *Naming.lookup* gets called with the server object name.

I.2.5 Compile the code

The three files Hello.java, HelloImpl.java, and QueryView.java are compiled using the javac compiler. Then the skeleton (HelloImpl_Skel.class) and the stub classes (HelloImpl_Stub.class) are generated using RMI compiler on the HelloImpl class: (rmic HelloImpl).

I.2.6 Start the Registry

An object being accessed remotely needs to be entered in a registry of objects. JDK1.2 provides a registry tool (*rmiregistry*) that can be used for this purpose. rmiregistry can run in a separate window or as a background process. It runs independently of the particular server and client but is required by both of them.

I.2.7 Start the Server

Next the server is started using the following command. Here URL is the location where the server class files are located and securityfile is a file which grants permission for a security manager to be installed on the client machine.

```
java -Djava.rmi.server.codebase=<URL>
     -DJava.security.policy=<securityfile> HelloImpl
```

In the final step the client is started.

Appendix J

Java Native Interface

The Java Native Interface (JNI) is the native programming interface for Java, using which, Java code that runs inside a Java Virtual Machine (JVM) can interoperate with applications and libraries written in other programming languages, such as C, C++, assembly etc. In addition, the *invocation API* allows to embed Java Virtual Machine into the native applications.

J.1 Where JNI might be used

Programmers use the JNI to write native methods when an application cannot be written entirely in Java. For example, native methods and JNI might be useful in the following situations:

- The standard Java class library may not support the platform dependent features that an application might have.
- A library or an application already written in another programming language might be needed to be accessed through Java applications.
- Sometimes a small portion of time critical code needs to be implemented in a lower level programming language, such as assembly, and then the Java application might want to call those functions.

Programming through the JNI framework lets you use native methods to do many operations. Native methods may represent legacy applications or they may be written explicitly to solve a problem that is best handled outside of the Java programming environment.

The JNI framework lets your native method utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

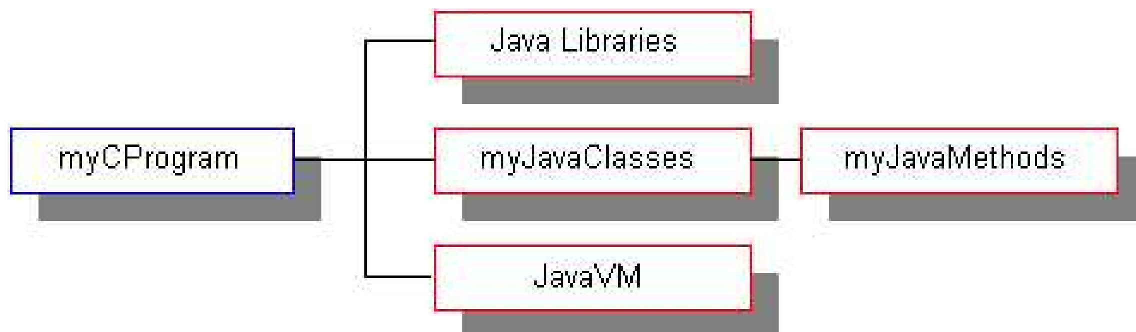


Figure J.1: How legacy programs use the JNI

Native methods can also easily call Java methods. Often, you will already have developed a library of Java methods. Your native method does not need to ”re-invent the wheel” to perform

functionality already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

The JNI enables you to use the advantages of the Java programming language from your native method. In particular, you can catch and throw exceptions from the native method and have these exceptions handled in the Java application. Native methods can also get information about Java classes. By calling special JNI functions, native methods can load Java classes and obtain class information. Finally, native methods can use the JNI to perform runtime type checking.

For example, the above figure shows how a legacy C program can use the JNI to link with Java libraries, call Java methods, use Java classes, and so on.

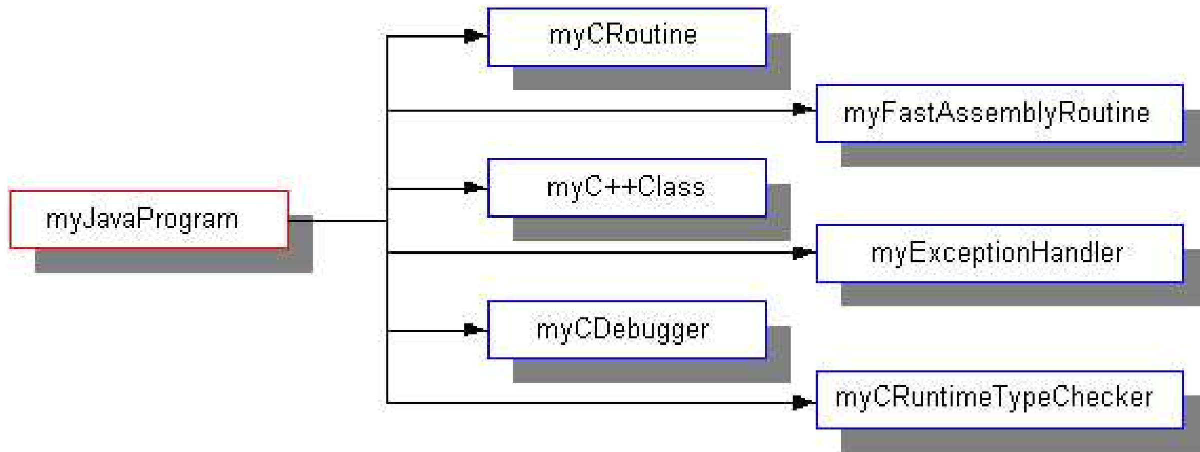


Figure J.2: How Java applications use the JNI

The above figure illustrates calling native language functions from a Java application. This diagram shows the many possibilities for utilizing the JNI from a Java program, including calling C routines, using C++ classes, calling assembler routines, and so on.

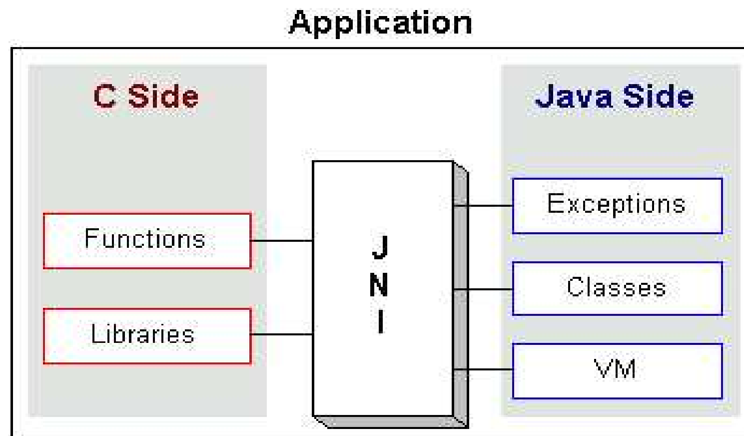


Figure J.3: JNI with Java and native applications

It is easy to see that the JNI serves as the glue between Java and native applications. The above diagram shows how the JNI ties the C side of an application to the Java side.

J.2 Steps to write a JNI application

In the following we will describe briefly the steps involved to invoke a native method from Java applications.

- The first step is to write the Java program. Create a Java class that declares the native method; this class contains the declaration for the native method and also some function which calls the native method. The declaration of the native method looks as follows:

```
public native void displayHelloWorld();
```

- Compile the Java class, which declares the native method and also other functions.

- Generate a header file for the native method, using javah with the native interface flag -jni. Once the header file is generated, the formal signature of the native method is known. A sample signature of a native method is shown in the following:

```
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
                       (JNIEnv *, jobject);
```

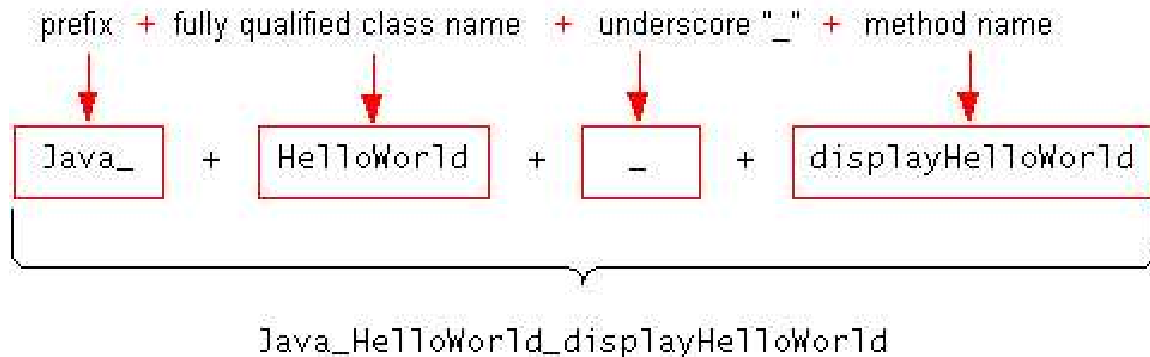


Figure J.4: Constructing the HelloWorld native method name

The name of the native language function that implements the native method consists of the prefix Java_, the package name, the class name, and the name of the native method. Between each name component is an underscore "_" separator. Graphically, this looks as depicted in the above diagram.The first parameter for every native method is a JNIEnv interface pointer. It is through this pointer that your native code accesses parameters and objects passed to it from the Java application. The second parameter is jobject, which references the current object itself. In a sense, you can think of the jobject parameter as the "this" variable in Java.

- Write the implementation of the native method in the programming language of your choice, such as C, C++ etc.

- Compile the header file and the implementation files into a shared library. The runtime system later loads the shared library into the Java class that requires it. Loading the library into the Java class maps the implementation of the native method to its declaration.

- Finally, use the java interpreter to run the program.

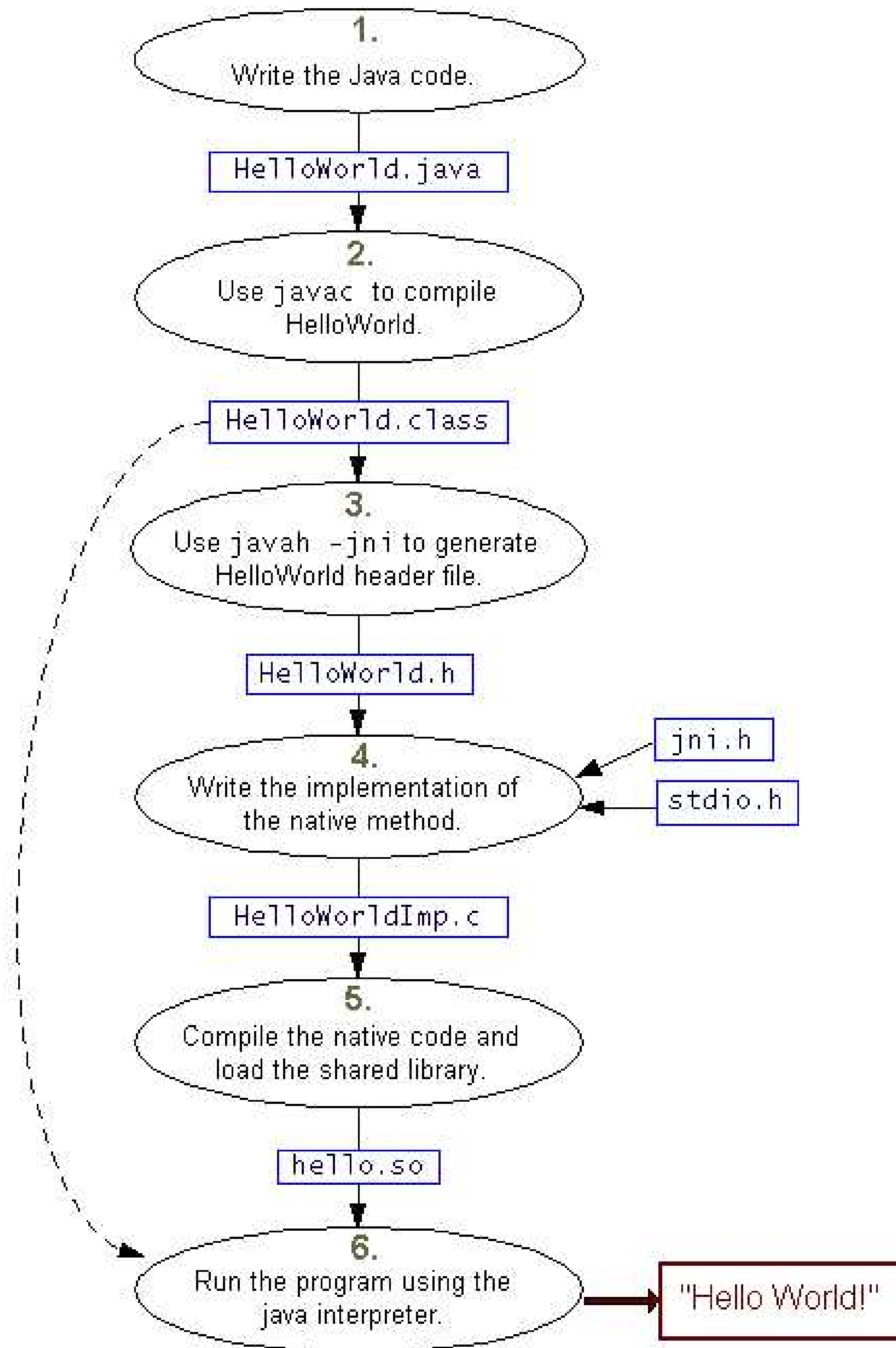


Figure J.5: Steps to write a Java program with native methods