OSHADHI-2
Biodiversity Information System

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Chandramouli B. N.

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

July 1997

Contents

Acknowledgments

I would like to thank my project guide, *Dr. Jayant Haritsa*, for his constant encouragement and help during the project. His affinity towards perfectness has always been the source of inspiration for us.

I am greatful to my advisor *Dr. M. Narasimha Murthy*, who helped me in taking important academic decisions during first three years.

I take this opportunity to thank Nalin, Shrividya, Nars, shreedevi, Binto, Ramesh who made DSL a real encouraging environment.

Cheers to Harsh, Toto, Amit, Sachin, Jha, Viswa, Mrinal and all my batchmates who made my four year stay in IISc an unforgettable moment of my life.

Finally, lots of love to my parents. What ever I am now, is because of them. Lots of love to Rekha, Girisha and Mohana.

Abstract

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

In this project, we overcome some of the limitations of the OSHADHI-1 by adding advanced features and call this enhanced system as OSHADHI-2. First, a powerful query language (**Vaidya**) is designed and implemented over the OSHADHI object model to enable the user to ask complicated queries. Second, some newer indexing methods that are specialized to object oriented databases such as **Nested Index** and **hcC tree** are added to decrease the processing time of a query. Third, we have provided some functionalities that required to process **geographical queries**.

Chapter 1

Introduction

1.1 Motivation

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

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

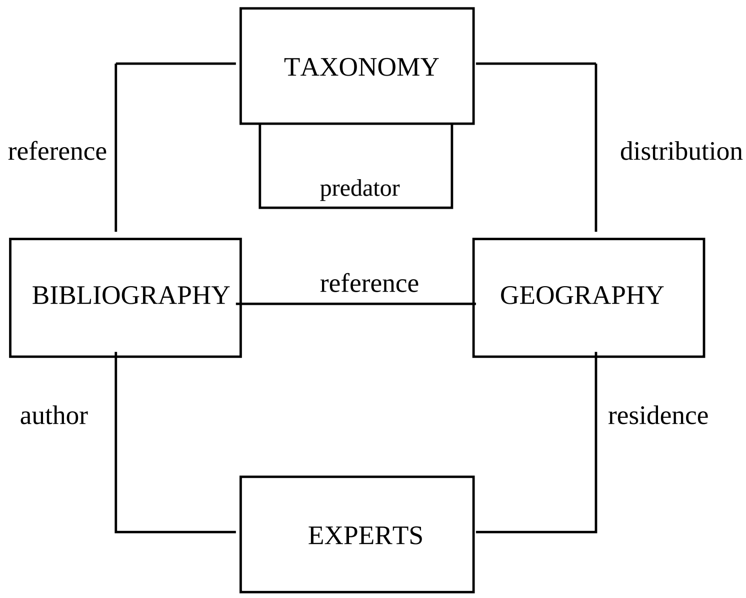

Figure 1.1: The bio-diversity data organization

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

[1] Places where plant specimens are stored

[2] These are sheets that contain a plant specimen and its details.

required is a system to manage the bio-diversity data in an efficient way and to provide quick retrieval facilities. Apart from the need to aid the process of bio-diversity conservation, maintenance of information such as whether a species is a local or foreign species is essential in the light of the *General Agreement on Trade and Tariffs (GATT)*.

To address the lacuna mentioned above the OSHADHI[3] project was started in *Database Systems Lab*(DSL), *Indian Institute of Science*. The project aims to design and implement an object model that best represents the bio-diversity data, and a query engine to enable querying on the database.

The initial prototype of OSHADHI was developed in an earlier work[Vidya95], which we refer to as OSHADHI-1 in this thesis. The various entities of bio-diversity data were identified and modeled, and the OSHADHI architecture was designed. The data model was implemented and the system was tested on a toy database.

In our work, which we refer to as OSHADHI-2, we have added some advanced features to overcome some of the limitations of OSHADHI-1. A query language called **Vaidya** is designed and an interpreter for that language is implemented, this enables the user to ask more complex queries. In addition, we have implemented some new indexing methods that are specific to object oriented models such as the hcC tree[SS94] and the Nested Index[BK89] to improve the performance of the system.

1.2 The OSHADHI system

The goal of the OSHADHI system is to design and prototype a new database system for effectively supporting the management of bio-diversity data. The development of the OSHADHI system has taken place in two phases. In the first phase an initial prototype (OSHADHI-1) was developed and some advanced features were added during the second phase (OSHADHI-2), which are described in this thesis.

[3] 'plant kingdom' in Sanskrit

1.2.1 OSHADHI-1

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

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

- It provides a Graphical User Interface for querying. This supports only limited number of queries such as, "given the characters identify a species with those properties", "display a geographical region", etc. The user often needs to ask more complex queries such as "given a particular species find all species with similar properties".

- Even though the model is an Object Oriented model, only B^+tree and quad tree is used for indexing the taxonomy data. B^+tree is the best method for indexing in relational databases, but it is not suited for Object Oriented databases. The newer indexing methods such as, hcC tree and Nested Index that are specific to Object Oriented databases, take the advantage of hierarchy in object model to enable faster access.

- The support for geographical query was very primitive. Only drawing and viewing of a geographical maps were supported.

We have addressed the above limitations in OSHADHI-2. In the next subsection, we describe the additional features incorporated in OSHADHI-2.

1.2.2 OSHADHI-2

The features that are incorporated in OSHADHI-2 are as follows:

Query Processor

A textual interface is provided in OSHADHI-2. The query language that is designed and implemented is called **Vaidya**[4]. The functionalities provided by Vaidya are inspired by *Object Query Language* (OQL)[Catt94]. The structure of Vaidya query language is similar to that of SQL [Orac88].

Access Methods

The performance of the OSHADHI system was improved by adding indexing methods such as hcC tree [SS94] and Nested Index [BK89]. These indexing methods take advantage of the hierarchy present in the object oriented model and enables faster access.

Geographical Queries

The processing of a geographical query is significantly different from other queries.For example, a query such as, *find a species with petal shape of its flower Oblong, and print its genera name* needs a search on the database for an object with the given properties and traversing through web of related objects. Whereas a geographical query such as, *print the name of the country which contains a particular point (specified by longitude and latitude)* needs retrieval of maps of countries and **computing** whether that point falls within the region or not. We have added some of the functionalities including the one just mentioned, that are required for processing geographical queries to the OSHADHI system. Hence, user can ask more complex geographical queries.

[4]'physician' in Sanskrit

1.2.3 Related work

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

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

1.3 Organisation of the Report

The rest of the report is organised as follows. Chapter 2 gives overview of some of the aspects bio-diversity data and Chapter 3 gives the overview of the work done as part of OSHADHI-1 project and gives architecture of OSHADHI-2. Chapter 4, 5 and 6 describes the features that we have added in OSHADHI-2. These features includes Vaidya query language, newer access methods and supports for geographical queries. Chapter 7 discusses about some implementation details. Finally we will summarize our work and suggest some future works in Chapter 8.

Chapter 2

The Bio-diversity

This chapter gives a brief description of the various aspects of bio-diversity data. We will first describe the classification of the bio-diversity data. In the later sections we discuss about features of bio-diversity database and the kind of queries that need to be answered by a bio-diversity database system. **Most of these materials are taken from [Vidya95]**.

2.1 Classification

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

2.1.1 Taxonomy Data

Each unit of classification is called a *taxon*. The four major taxons in the classification of species are *Order*, *Family*, *Genera* and *Species*. An order contains several related families, which means that families that exhibit certain similar characteristics are grouped into an order. Likewise, a family is represented by a collection of genera and a genera is a collection of related species [Vasi74]. Each taxon has a biological name. The classification hierarchy of species and an example are shown in Figure 2.1.

Species are identified based on their leaf, stem, flower and fruit characteristics. Hence, these characters could be referred to as identifying characters. There are other characters,

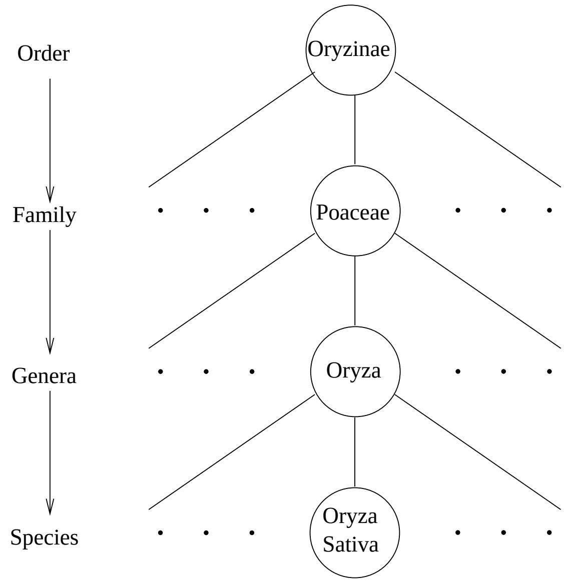

Figure 2.1: The taxonomy of species

for example, the root character, which do not aid in identification of a species. These characters are referred to as non-identifying characters. A flower is composed of several parts namely, petals, sepals, inflorescence, carpels, ovaries, anthers, style and stigma. Each of these characters could be used for identification of a species and hence they are also identifying characters. Similarly, a leaf contains parts such as blades, sheath, etc. Hence, every order, family, genus and species is associated with a set of identifying and non-identifying characters which differentiate it from the other orders, families, genera and species respectively. A species may have synonyms (synonyms are commonly used names for species, eg, Rose is a synonym for Rosa Alba), line drawings and a large amount of textual information associated with it. The textual information may include extracts from journals and textbooks. A species may also have other species as its predators.

2.1.2 Geographic Data

There are essentially four types of geographical organizations – the *bio-geographic provinces* (e.g., Malabar, South Karnataka, etc,.), the *biomes* (e.g., evergreen, decidous, etc,.), the *administrative organizations* (e.g., district, state, etc,.) and the *forests*. These organizations, along with the botanical gardens and herbariums, represent the geographical data.

Every country, state, district and city has a political map associated with it. The environmental conditions (soils, temperature, pressure humidity, etc,.) in the countries, states, districts and cities need to be recorded in two forms. One is the spatial variation (distribution) of the condition and the other is the temporal variation (variation of the condition over a period of time). Similarly, for all the species found in a geographic region, the distribution and temporal variation in population are to be recorded. The information about the distribution of a species could be available at any geographic organisation type. For example, we might have the information that a species is found in the Malabar province or that a species is found usually in evergreen forests or that a species is found in Karnataka.

2.1.3 Bibliographic and Experts Data

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

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

2.2 Features of the Bio-diversity Database

The bio-diversity application requires management of taxonomy data, geographic data, bibliographic data and experts data. A study of the data provided by the *Centre for Ecological Sciences*, *Indian Institute of Science* reveals the following prominent features:

- *Model*

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

- *Types of operations*

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

- *Versions*

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

- *Large Volume of Data*

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

- *Infrequent Updates*

  Since the properties of species will not change with time, the bio-diversity database is characterized by infrequent updates.

- *Schema Evolution*

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

2.3 The Queries

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

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

- *List all predators of a given species*

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

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

Having identified the goals of the project, in the next chapter we will discuss the OSHADHI system that aims to fulfill the specified goals.

Chapter 3

The OSHADHI System

The OSHADHI system was designed for managing the bio-diversity data and to answer related queries. In this chapter we first describe the development of OSHADHI-1. It involved identifying the entities in the bio-diversity domain, and modeling the domain, designing of OSHADHI-1 architecture, implementing the model, and designing and implementing the graphical user interface [Vidya95]. At the end of this chapter we give an outline of the features incorporated in OSHADHI-2.

3.1 OSHADHI-1

3.1.1 Object Model of OSHADHI

The object oriented approach to software development presented in [Rumb91] is used in the development of the OSHADHI system. This approach suggests the design of a detailed object model of the system prior to system implementation. Initially, the bio-diversity domain was studied and its characteristic features were analyzed. Then the requirements of the system were specified. During the analysis, the nature of the data and type of queries were identified. The design of data model involved identifying the objects and their attributes and recognizing the inheritance, aggregation and association relationships between them. The entities such as order, family, genera, species, country, state, city and forest formed the major objects of the bio-diversity data. The details of the complete object model are given in [Vidya95]. The design of OSHADHI-1 system architecture was the next step.

3.1.2 The system design

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

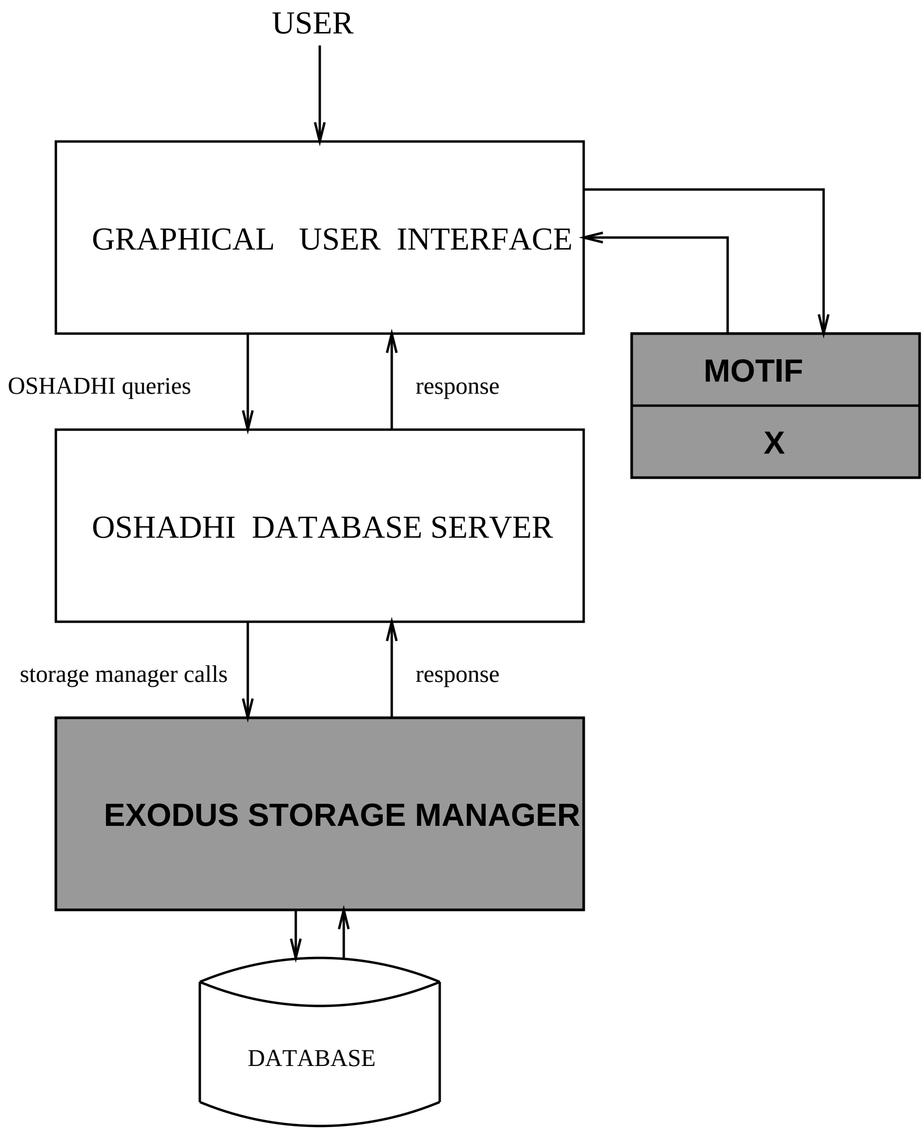

Figure 3.1: System Structure of OSHADHI-1

- **The EXODUS Storage Manager Server**

  The storage manager server of EXODUS [Care86a] forms the lowest layer of OSHADHI. The server receives requests from clients for various operations on objects and collections. The server is responsible for concurrency control, recovery, transactions and storage management. EXODUS provides a programming language called **E** which is a superset of C++. E hides details of communicating with EXODUS storage manager through socket, and gives a neat interface. Hence, E was chosen programming language for writing OSHADHI system. We highlight some features of E. More details on the programming constructs of E are given in [Rich87].

  Persistence

  E provides database types called *DB Types* analogous to the C++ data types. A dbtype could one of the following: *dbshort, dbint, dblong, dbfloat, dbdouble, dbchar and dbvoid.* Every C++ data types has a corresponding dbtype. Objects of dbtypes are similar to their non-db counterparts except that an object of a non-db type is volatile (whose lifetime is bound by the program run) whereas an object of db-type could be declared **persistent**. A persistent object survives across program runs and crashes.

  Iterators

  This is a feature that aids database querying. Iterators are control abstractions which have two agents that work together. One is an iterate loop and another is the iterator function. The iterate loop invokes the iterator function which yields a stream of values one-at-a-time to the client loop which then processes it. The iterator function therefore needs to retain its state between two yields. This feature is particularly useful in performing operations on a group of related objects since the iterator function could be written independent of the client loops.

Collections

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

Transactions

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

- **The Database Server**

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

- **The Interface**

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

3.1.3 Implementation

After the completion of the design of bio-diversity object model and the system architecture for OSHADHI-1, the OSHADHI-1 server and GUI were implemented. The major components implemented in the OSHADHI-1 system are described below.

- OSHADHI Server

  The OSHADHI server consists of the bio-diversity object model and the routines which communicate with the client(GUI) and process the query received from the client. We highlight some implementation details of the OSHADHI server.

  - E is used as the programming language.
  - The attributes that represent the characters of a taxon such as sepal color and leaf shape, were implemented as *enumerated* data, ie, each possible value of these attributes were associated with an integer value, and their integer values are stored in the objects.
  - The *association* relation is implemented as a collection of pointers to the object.

- A Graphical User Interface is provided for interacting with the database. Using this GUI user can enter the data, list the current entries in the database, identify the species based on the characters, etc.

- B^+tree and Quad tree were used for indexing on the objects (Quad tree is used to index geographical locations and B^+tree is used to index all other objects of all other classes).

3.2 OSHADHI-2

We enhanced the basic OSHADHI system during the second phase of the project. The query language Vaidya was designed based on the suggestion given by students of *Centre for Ecological Sciences*, *Indian Institute of Science*. The performance of the system is improved by incorporating hcC tree and Nested indexing methods. The architecture of the OSHADHI-2 is shown in Figure 3.2

We have enhanced the OSHADHI module by adding hcC tree and Nested indexing methods to it. The VAIDYA module represents the Vaidya query processing system. This interacts with the user at the one end and OSHADHI module at the other. The internal architecture of the VAIDYA module is explained in the Chapter 7.

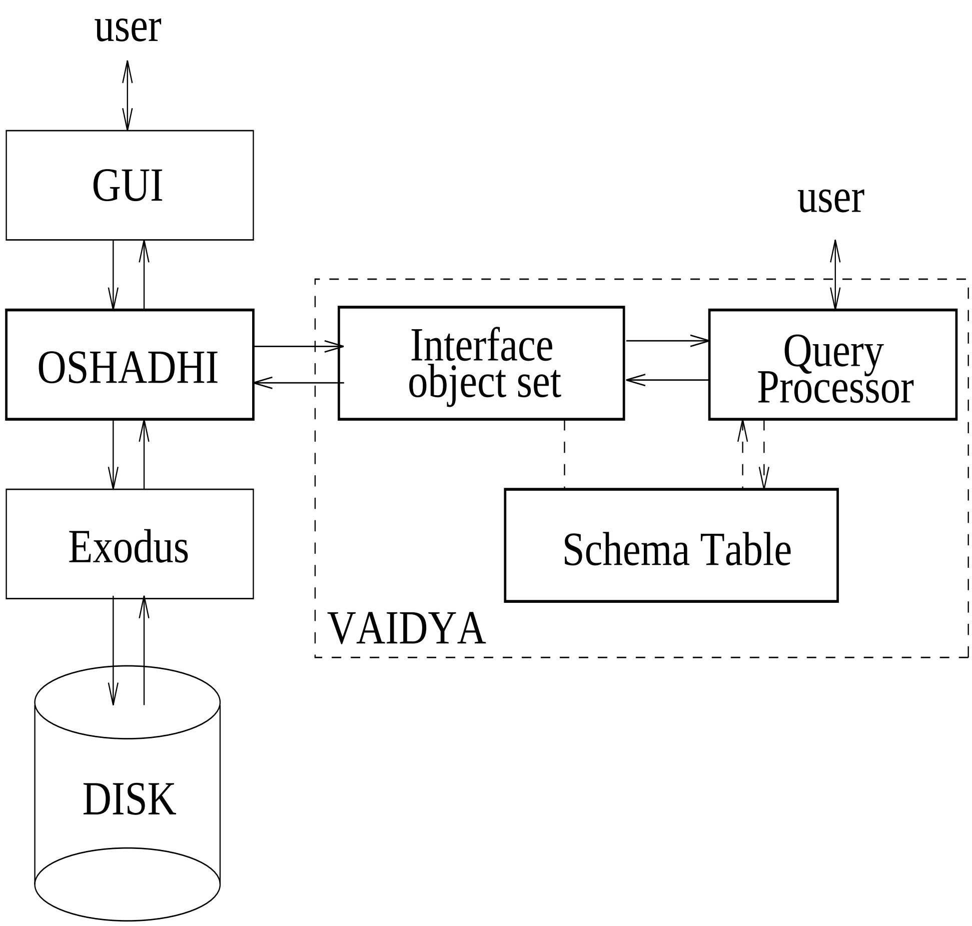

Figure 3.2: The architecture of OSHADHI-2

Chapter 4

The Vaidya Language

The aim of designing the language Vaidya was to give the user more querying capability yet maintaining the simplicity of the language. The design of Vaidya is inspired by that of *Object Query Language* (OQL)[1] [Catt94]. The basic structure of Vaidya is similar to that of SQL [Orac88]. In this chapter we will first describe the goals of the design. In the later sections we will describe the language Vaidya along with examples.

4.1 The Problem Statement

The design goals for Vaidya were the following:

- *Simplicity*: The language should be easy to learn and use. This requirement becomes prominent because the intended users of this system are biologists and generally they will have less familiarity with computer languages.
- *Generality*: The user should be able to express any bio-diversity related query in his/her mind using this language.
- *Extendibility*: As mentioned earlier, the schema of bio-diversity database maybe changed over the time. The language should be adaptable to such changes easily.

[1] OQL is a part of *Object Database Standard: ODMG* [Catt94]

Having specified the goals of the design, we now list some of the queries which have to be expressible in Vaidya language. These are some of the queries we have used as guide lines while designing the language.

- Select genera *Rosa* and list its
  flower.inflorescence.fertile_part and
  flower.sepal.symmetry.

- Identify a species with
  flower.inflorescence.fertile_part is *Cylindrical* and
  flower.sepal.symmetry is *Slightly asymmetrical.*

- Given a species *RosaAlba* (Rose), find all the species which have
  same *flower.inflorescence.fertile_part* and *flower.sepal.symmetry.*

- List all species with
  flower.inflorescence.fertile_part is *Cylindrical* and
  flower.sepal.symmetry is *Slightly asymmetrical* that are found in *India*

- List the names of the places where two given species, *RosaAlba* and *CoffeaArabica* (Coffee), are found simultaneously.

In the next two sections we describe the conventions that we use for describing the language. Then, in the later sections, we describe the Vaidya language.

4.2 The Conventions

The conventions and terminologies we have used for describing the language, are listed in this section. These conventions are used throughout the thesis.

- **Relationship Attributes**: In OSHADHI, relationships are implemented as attributes of objects participating in the relationship. We refer to these attributes as *Relationship Attributes*. The value of the *Relationship Attribute* of an object is the set of references to all objects that are related to that object. For example, all *Species* have an attribute called *foundin*. The value of this attribute is the set of references to all the geographical locations where that species are found (*foundin*).

- **Path Expression**: A string $C_1.A_1.A_2.A_3...A_m$ is said to be a *Path Expression* if
  - C_1 is a class in the object model or an identifier declared to be representing a class
  - A_1 is an *Relationship Attribute* of class C_1
  - A_i is an *Relationship Attribute* of class C_i where C_i is the domain of the attribute A_{i-1}
- **Subpath expression**: If $C_1.A_1.A_2...A_m$ is a path expression then, C_1 and $C_1.A_1.A_2...A_l$ for $1 \leq l \leq m$ are called as *Subpath expressions* of $C_1.A_1.A_2...A_m$.

Example 1:

The class *Species* in the OSHADHI object model has an attribute called *name* and a relationship attribute called *foundin* representing the foundin relationship with the class *GeoLocation*. GeoLocation is the class of the objects that stores information about geographical locations. Assume the following scenario

- A species called "HaliauthusAnnus" is in the database
- *HaliauthusAnnus* are found only in India, Srilanka and China
- X and Y are the variables of the class *Species*

Then, the following statements are true:

- *HaliauthusAnnus.foundin* is the set of references countries India, Srilanka and China
- *Species.foundin.name* is a path expression
- *Species* and *Species.foundin* are subpath expressions of the path expression *Species.foundin.name*

- X and $X.foundin$ are subpath expressions of the path expression $X.foundin.name$
- X and $X.foundin$ are not subpath expressions of the path expression $Y.foundin.name$ (regardless of the values of X and Y

The Enum Data

The enum data type in the object model are viewed as string data type which can take only some possible values. For example, the attribute stalk of the class flower can take the values "Sessile", "Sub-Sessile", "Short-pedicillate" and "Long-pedicillate".

4.3 The Language

As we have mentioned in the beginning of this chapter, Vaidya language is inspired by the *Object Query Language* (OQL) [Catt94]. Vaidya provides keywords *Insert*, *Delete*, *Select* and *Relate*, for inserting an object, for deleting an object, for querying and for creating a relationship between two objects, respectively. The usage of these keywords are described in this section. Vaidya does not provide any methods to change the schema of the OSHADHI object model. However, if any changes are made to the schema by a person with knowledge of E programming language, adapting Vaidya to the new schema is easy.

4.3.1 Insert

The format of an insertion is as follows:

Insert class_name
value (attribute1 = value1, attribute2 = value2, ..., attributen = valuen) ;

Here, the *class_name* is the type of the object being inserted (Species, Genera, Family, etc.). *value* and *Insert* are keywords.

Example 2:

Insert Species
value (name = "RosaAlba", genera = "Rosa",
flower.petal.shape = "Oblong") ;

This will insert a species called by the name "RosaAlba" and petal shape of its flower is "Oblong", and also this will establish a *relationship* called *belongsto* with the genera "Rosa". Genera should be specified while inserting a species. It is an error if the genera "Rosa" does not exist in the database.

4.3.2 Delete

The format of a delete operation is as follows.

Delete variable_X
from class_name varlist, class_name varlist, ..., class_name varlist
where (predicate) ;

Here, *varlist* is list of variables separated by comma (","). These variables are referred to as *Object Variables*. These object variables play similar roles to that of played by *tuple variables* in SQL. *variable_X* should occur exactly once in the *from clause*. For each instantiation of path expression such that *predicate* holds true, the object corresponding to *variable_X* is deleted from the database.

Example 3:

Delete rose // *rose* is a variable of the type Species.
from Species rose
where (rose.name == "RosaAlba") ;

This will delete "RosaAlba" from the database.

4.3.3 Relate

The current implementation of the Vaidya query processor does not support assigning or comparing sets. For adding elements to the relationship attribute representing many-to-many relation, one has to use the keyword *Relate*.

Relate variable_X relationship_name variable_Y
from class_name varlist, class_name varlist, ..., class_name varlist
where (predicate) ;

For each instantiation of path expression such that *predicate*s holds true, the objects corresponding to *variable_X* and *variable_Y* are related by the relationship *relationship_name*, ie, The relationship attributes of the objects participating in the relationship are consistently modified.

Example 4:

Relate rose foundin india
from Species rose, Country india
where (rose.name == "RosaAlba" and india.name == "India") ;

This will relate "RosaAlba" and "India" by the relationship *foundin*.

Relate china houses coffee
from Country china, Species coffee
where (china.name == "China" and coffee.name == "CoffeaArabica") ;

This will relate "China" and "CoffeaArabica" by the relationship *houses*.

4.3.4 Query

The general structure of a query is:

Select path_expression_list
from class_name varlist, class_name varlist, ..., class_name varlist
where (predicate) ;

Here, path_expression_list is the list of path expressions separated by a comma. varlist is the list of variables separated by comma. Predicates are put on the path expressions. For example

species.name == "RosaAlba"

genera.species.foundin.Population <= 1000.0

are valid predicates.

Example 6:

List all the species that belong to the genera which contain "OryzaSativa".

```
Select   rice.genera.species
from     Species rice
where (  rice.name == "OryzaSativa" ) ;
```

Another way of writing the same query is

```
Select   spec.name
from     Species spec
where (  spec.genera.species.name == "OryzaSativa" ) ;
```

Example 7:

List all locations where both "RosaAlba" and "CoffeaArabica" are found.

The user may incorrectly write the query as

```
Select   cntry.name                    // Incorrect.
from     Country cntry
where (  cntry.houses.name == "RosaAlba" and
         cntry.houses.name == "CoffeaArabica" ) ;
```

This will produce no output. Because, both occurrence of the path expression *cntry.houses.name*, either will be "RosaAlba" or "CoffeaArabica". Hence, both predicates are never satisfied simultaneously.

The correct way of writing the above query is

Select cntry.name
from Country cntry, temp
where (cntry.name == temp.name and cntry.houses.name == "RosaAlba"
and temp.houses.name == "CoffeaArabica") ;

4.4 Member Function Call

Member function call is an important functionality provided by Vaidya. The user is allowed to call some member functions of the classes in the object model. In the current implementation, few member functions of only *Polygon* class can be called. *Polygon* stores maps of the geographical locations in the form of approximated polygon. These member functions can do arbitrary complex computations (Since, they are written in E language). Hence, using these the user can do certain computations which are not inherently possible by Vaidya. For example, The member function IsPointInside(float X, float Y) computes whether the point (X, Y) lies within the the region represented by the polygon or not. It returns a boolean value TRUE if the point lies with the region, returns FALSE if not. The user can use this function for finding a country containing a given point.

Example 8:

Find country containing the point with longitude = 15.0 and latitude = 40.0.
Select cntry.name
from Country cntry
where (cntry.map.IsPointInside(15.0, 40.0) == TRUE) ;

If the Function call is done in the select clause, the function call is done for all those objects satisfying the predicates and the return value is printed.

We will discuss more about these member function calls in the Chapter 6.

4.5 Data Definition Language

Vaidya is a Data Manipulation Language (DML), data definitions cannot be changed using Vaidya. In the current implementation of OSHADHI (ie. OSHADHI-2), the only way to change the data definition is to change the source code of OSHADHI. However, due to the modular design of the Vaidya query processing system, there is a scope for implementing a Data Definition Language (DDL) to work with Vaidya. However, we leave this as a future work.

Finally, we write the guideline queries mentioned in the section 4.1 in Vaidya language.

- Select genera *Rosa* and list its *flower.inflorescence.fertile_part* and *flower.sepal.symmetry*.

```
Select   gen.flower.inflorescence.fertile_part,
         gen.flower.sepal.symmetry
from     Genera gen
where ( gen.name == "Rosa" ) ;
```

- Identify a species with *flower.inflorescence.fertile_part* is *Cylindrical* and *flower.sepal.symmetry* is *Slightly asymmetrical*.

```
Select   spec.name
from     Species spec
where ( spec.flower.inflorescence.fertile_part == "Cylindrical" and
         spec.flower.sepal.symmetry == "Slightly-asymmetrical" ) ;
```

- Given a species *RosaAlba*, list all the species which have same *flower.inflorescence.fertile_part* and *flower.sepal.symmetry.*

```
Select   spec.name
from     Species spec, rose
where (  rose.name == "RosaAlba" and
         spec.flower.inflorescence.fertile_part ==
                   rose.flower.inflorescence.fertile_part and
         spec.flower.sepal_symmetry == rose.flower.sepal.symmetry ) ;
```

- List all species with *flower.inflorescence.fertile_part* is *Cylindrical* and *flower.sepal.symmetry* is *Slightly asymmetrical* and that are found in *India*

```
Select   spec.name
from     Species spec
where (  spec.flower.inflorescence.fertile_part == "Cylindrical" and
         spec.flower.sepal.symmetry == "Slightly-asymmetrical" and
         spec.foundin.name == "India" ) ;
```

- List the names of the places where two given species, *RosaAlba* and *CoffeaArabica*, are found simultaneously.

```
Select   cntry.name
from     Country cntry, temp
where (  cntry.name == temp.name and cntry.houses.name == "RosaAlba" and
         temp.houses.name == "CoffeaArabica" ) ;
```

Chapter 5

Access Methods

In object oriented database systems, query processing involves traversing complex structures represented by the object oriented query model. For example, object oriented query processing systems need to support queries on objects forming a hierarchy. The hierarchy itself maybe a result of specialisation or aggregation and queries based on each of these hierarchies need to be processed differently. The traditional access methods such as B^+ tree [Come79], which gave relational database systems performance benefits by providing associative access to data, are not powerful enough to deal with complex data. Thus, the search for methods to deal with the properties of complex data has led to a variety of access techniques appearing in the literature.

In order to discuss the various index organisations, we need to refer to some topics concerning query processing and execution strategies. We borrow the concept of *query graph* from [Kim89] for this purpose. The two basic strategies that are devised to traverse a query graph in that paper are:

- *Forward traversal*: The first class visited is the target class of the query(root of the query graph). The remaining classes are traversed starting from the target class in any depth order.

- *Reverse traversal*: The traversal of the query graph begins at the leaves and proceeds bottom-up along the graph.

We also use the following definitions given in that paper.

- *Path Instantiation*:

  A path instantiation refers to the sequence of objects obtained by the instantiation of the path.

  A path instantiation is said to be *complete* if it contains an instantiation of the root node of a complete path as well as that of the leaf node in the path.

  A path instantiation is said to be a *left-partial* instantiation if the path instantiation does not include the root node in a complete path. Similarly, a path instantiation is said to be a *right-partial* instantiation if the path instantiation does not contain the node representing the domain of the path (leaf node).

In this chapter, we attempt to classify the access methods and provide a survey of access methods that are specific to a classification but independent of any problem domain. **Most of the materials in this chapter are taken from [Krish96]**.

5.1 Classification of queries

In this section, we classify queries based on the various associations that can be identified in any object oriented model. The classification provides a direct mapping to the type of queries or a combination of them that the database system maybe called upon to handle. Queries can be broadly classified as shown in Figure 5.1.

- *Simple Value based queries*

  Simple Value based queries are queries that essentially have simple literals or base types in their search predicates. It can be seen that all queries in relational database systems fall under this category due to the absence of complex relationships (for example, hierarchies). Thus, techniques used in the context of relational database systems can be exploited for simple value based queries.

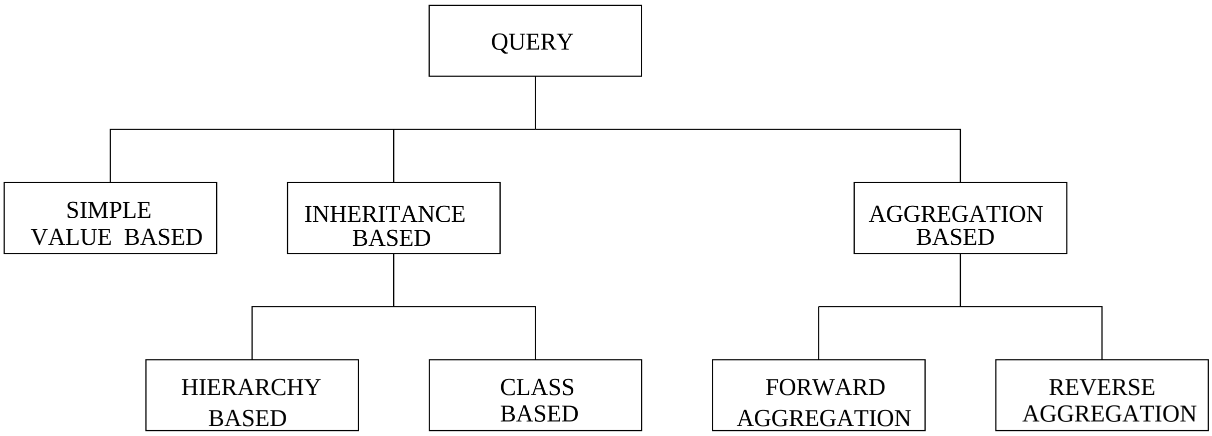

Figure 5.1: Classification of Queries

- *Inheritance based queries*

  To a query processor, a query based on the inheritance relationship between classes, poses a number of problems. It maybe necessary for the query processor to recognize all objects with a specified value for an attribute which is at the base of an inheritance hierarchy. If this attribute has been inherited by other objects, it becomes necessary to identify all such objects as well. Thus, any access method that deals with *inheritance based queries* needs to be able to answer queries based on attribute values which are part of a hierarchy. These queries can thus be called *inheritance based class hierarchy* queries. It also maybe a requirement in some systems to be able to answer pinpoint queries which are directed at a particular class in the inheritance hierarchy. These queries can be called *inheritance based single class* queries.

  Thus, in general, the query processor in an object oriented system must be capable of handling both hierarchy based queries as well as single class based queries targeted at inheritance relationships in the data model.

- *Aggregation based queries*

  An aggregation hierarchy is rooted at one object and a *forward* (top down) traversal, of the hierarchy is required to extract an object at the leaf of the hierarchy in the absence of any access method. It can be seen that a path exists for the navigational retrieval of the required object. Navigational retrieval maybe expensive especially when object accesses are expensive. A potentially more difficult problem is to have a reference to an object at the leaf of the hierarchy and then attempt to find the root object of the hierarchy which requires a *reverse* (bottom up) traversal of the aggregation hierarchy. This problem is further amplified in systems that do not maintain reverse references along an aggregation hierarchy.

5.2 Simple value based queries

Simple value based queries in object oriented databases are similar to the queries seen in the context of relational database systems where queries were based on a single attribute or key value. In this section, we briefly highlight the characteristics of some of the popular indexing mechanisms used in relational databases systems and which can be exploited in the context of object oriented systems.

5.2.1 B^+ tree

The B^+ tree [Come79] is a n-ary tree which indices all the records with the help of a balanced tree structure. The nodes of the tree are stored on disk and hence to access a record would require disk accesses proportional to the height of the tree. Being a balanced n-ary tree, it has rules associated with it for inserting or deleting records from the index. Since the organisation of data indexes is similar to the principles of the binary tree, it can be seen that the data appear in a sorted manner as the leaf nodes are traversed from left to right if the leaf nodes are chained. This property of the B^+ tree makes it extremely useful for range queries wherein data records whose indexed attributes fall within a certain range are required by the query. Since most commercial applications have such requirements, the B^+ tree is widely used in relational database systems.

5.2.2 Extendible hashing

Extendible hashing [Fagi+79] is another method used to index data records. In this method, attributes on which the records are indexed are hashed onto buckets which in turn point to disk addresses. The hash structure itself is capable of dynamically changing in size in proportion to the size of the database without the need to change the hashing function thus avoiding problems associated with static hashing. Also, only a single disk access is required to retrieve the indexed record. However, hashing in general destroys the order of data and hence it is cumbersome to obtain a sorted order using the hash table. Thus, this structure is unsuitable for range queries and its utility remains limited to exact match queries.

5.3 Aggregation based Access Methods

In this section, we present a collection of access methods to address the problem of handling aggregations in an object oriented model. We survey nested indices, path indices, multi index organisation and multi index with path configuration. A detailed survey of some of the access methods maybe found in [FMV94].

The queries targeted on aggregation hierarchies require retrieval of the objects that form the path given the root node value or the leaf value, that is the forward traversal and reverse traversal respectively. For illustration purpose we consider instantiating the path expression $C_1.A_1.A_2.A_3.A_4$ by the instances O_i of the classes C_i, $i = 1$, ..., 5 where O_1 is an instance of the class C_1 and C_j is the domain class of the attribute A_{j-1}, j = 2, ..., 5. The problem, is therefore to be able to retrieve all objects of type C_1 given a value for O_5. Normally, this requires an exhaustive search, but the presence of access methods facilitates efficient retrieval of the required objects.

5.3.1 The Nested Index

Structure

The nested index [BK89] addresses the problem of traversals along a complete path instantiation by providing a *direct* link between the root and the leaf node of the path instantiation. The nested index was originally designed to facilitate queries that require reverse traversals of query graphs, but can be adapted to facilitate forward traversals as well. The nested index basically uses the leaf node of an aggregation hierarchy (which is usually a literal) as the key for a search and maps this value to the objects at the beginning of all path instantiations that result in having the required value for the leaf node. This idea of a nested index is best represented pictorially as shown in Figure 5.2.

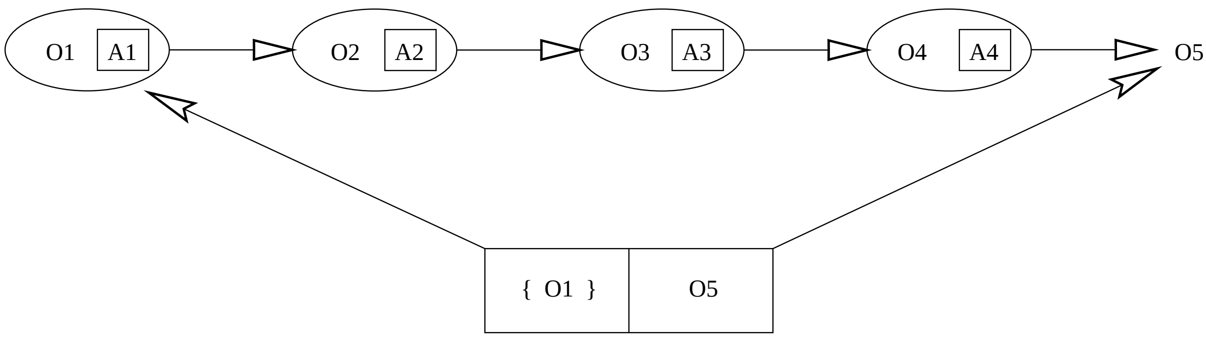

Figure 5.2: Nested index

It can be seen that the key value is the value of O_5 and the index contains the set of objects that contains this value through an aggregation hierarchy. The Figure 5.2 shows only one element of the set, that is O_1. The number of path instantiations that lead to attribute A_4 having value O_5 determines the cardinality of the set.

The collection of such nodes representing the associations between the leaf and root of path instantiations can be maintained in a B^+ tree with key value as the leaf node's value. This organisation wherein a B^+ tree is used to organise the nodes gives additional power in that range queries can also be addressed by taking advantage of this organisation and the capability of a B^+ tree to preserve order. The leaf node of such an organisation can be modeled as a node containing the record length, key length, key value, the number of elements in the set containing references to the root nodes of path instantiations and the list of OIDs of the objects which form the root nodes of the path instantiations.

Insertions and Deletions

Insertions are handled by updating the entries in the leaf node wherein the new OID of the root object of a path instantiation having the key value for the leaf object is inserted into the list of OIDs and the number of elements of the set updated. If a nested index node does not exist on an insertion, a new node is created and appended at the appropriate location in the B^+ tree. Deletions can similarly be handled by deleting the corresponding OID of the root object from the list and updating the count of elements in the set. The node itself is deleted in the case where the cardinality of the set becomes zero. Notice that if the path is of length one, the nested index organisation becomes identical to the simple B^+ organisation used in most relational database systems.

Partial Instantiations

The nested index structure also stores objects at the root of right partial instantiations apart from storing objects of complete path instantiations. The right partial instantiation, by definition, does not have the leaf object of the path in its instantiation and hence is represented by a NULL key value. Thus, all instances of the class at the root of a path are mapped by the nested index organisation. However, from the index it is not possible to determine the nature of the right partial instantiation in that it is not possible to determine the actual length of the complete path instantiation of which it forms a subset. The nested index organisation does not record any left partial instantiations since it stores only the root nodes of a path instantiation which by definition is absent in left partial instantiations.

Retrievals and Updates

It can be seen that nested index facilitates speedy retrievals. However, updates cause a problem. Updates refer to an operation where the value of an attribute (which could be an object reference in the aggregation hierarchy) is changed. In this case, one forward traversal is made to ascertain the leaf value with the original attribute value, another is made to ascertain the leaf value with respect to the new attribute value and a reverse traversal is made to obtain the object reference (OID) of the object at the beginning of the path instantiation. The nested structure is then updated using the result of these traversals. It can be seen that reverse references need to be maintained for this configuration to facilitate reverse traversals. Thus, the nested index configuration is most suitable for systems which primarily serve data retrieval requirements and where data updates are rare.

5.3.2 The Path Index

Structure

The path index [BK89] addresses the problem of traversals along a complete path instantiation by providing *direct* links between the objects of the path instantiation. The path index, like the nested index, uses the leaf node of an aggregation hierarchy as the key for a search and maps this value to all path instantiations that result in having the said value for the leaf node. This idea of a path index is shown in Figure 5.3.

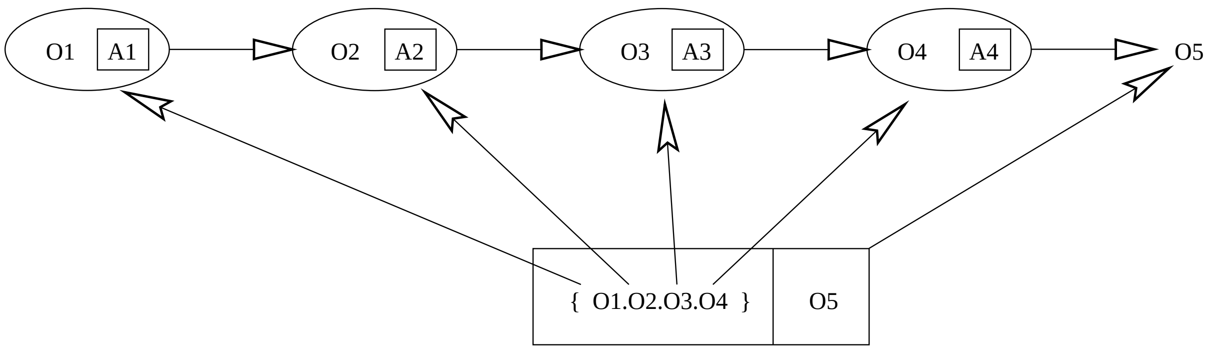

Figure 5.3: Path index

The key value used to map the path instantiations is the value of O_5 and the index itself contains all path instantiations that lead to the value of O_5 (The Figure 5.3 shows one such path). It can be seen that the path index contains a set of paths each path identified by a set of references (OIDs).

The collection of such nodes mapping the key values to the path instantiations can be maintained in a B^+ tree with key value as the leaf nodes' value (as in the case of the nested index). The leaf node in such an organisation can be modeled as a node containing the record length, key length, key value, the number of path instantiations and the list of path instantiations. A possible leaf node structure is shown in Figure 5.4. It maybe necessary to order the list of OIDs in each path instantiation depending on the implementation.

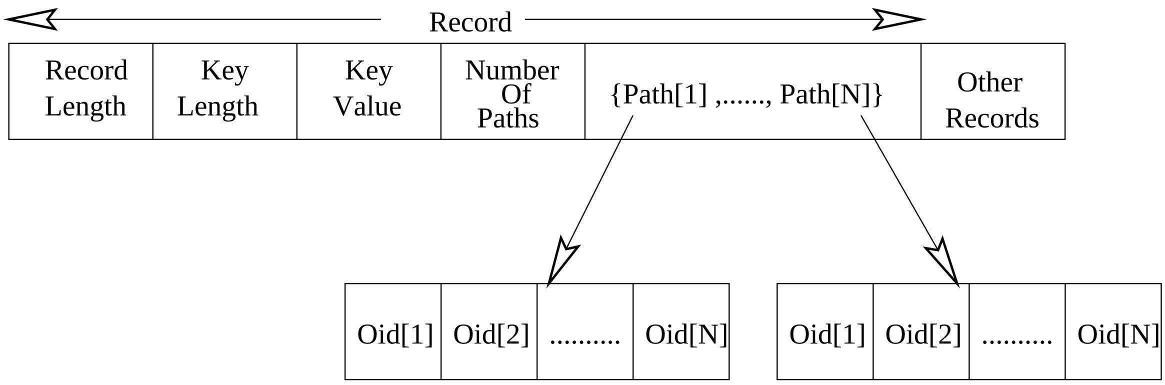

Figure 5.4: Leaf node in a path index

Insertions and Deletions

Insertions are handled by first finding the OIDs that contribute to a path instantiation to be inserted and then inserting this set of OIDs into the set of path instantiations. The count of path instantiations and other parameters are then updated. If a path index node does not exist on an insertion, that is, a path instantiation for the given mapping key value does not exist, a new node is created and appended at the appropriate location in the B^+ tree according to the rules of the B^+ tree insertion. Deletions are similarly handled by deleting the set of OIDs representing the path instantiation and decrementing the count of the path instantiations. The other parameters are also updated. If the set of path instantiations for a given key value becomes zero as a result of a deletion of a path instantiation, then the node itself is deleted from the B^+ tree following the rules for deletion in a B^+ tree. As in the case of the nested index, if the path is of length 1, the path index organisation becomes identical to the simple B^+ tree organisation used in most relational database systems.

Partial Instantiations

The path index, unlike the nested index, stores left and right partial instantiations in addition to storing complete instantiations. A NULL key value specifies a right partial instantiation while a left partial instantiation can be detected by the cardinality of the set of OIDs in a path instantiation. The path index, holds the OIDs of all the objects involved in a path instantiation and hence can be used to answer queries against nested objects with a single index lookup. It can be seen that this is not possible in a nested index. This retrieval of the entire path instantiation enables a special kind of operation known as *path projection*. This operation can be used to project certain objects from a path instantiation instead of the whole path instantiation in order to address queries that require such an arrangement. This operation itself can be done dynamically depending on the nature of the query and the nature of the path instantiations (left partial, right partial or complete).

Retrievals and Updates

Updates in a path index do not possess the disadvantages faced by the nested index configuration. Two forward traversals are necessary to determine the key value with respect to the new path. and with respect to the old path. Since the entire path instantiation is available in the index node itself, the root node of the path instantiation can be determined from the index itself. It can be seen that no reverse traversal is required in order to update the index. Thus, this configuration can be used in database systems that do not maintain explicit reverse references as well. The main disadvantage of the path index configuration is in the fact that the leaf nodes now contain more information about every path instantiation thus reducing the number of such records that can be stored in a node. Thus, it maybe necessary to scan more leaf nodes than if a nested index were maintained in order to retrieve information. In summary, retrievals in the path index is *costlier* than the nested index while updates are *cheaper*.

5.3.3 Multi index

Structure

The multi index [MS86] is an organisation that essentially breaks a path into its basic components and uses a separate index (for example, a B^+ tree) to maintain all path instantiations of each basic component. Therefore, given a path $C_1.A_1.A_2 \ldots A_n$ $(n \geq 1)$, traversing n classes $C_1 C_1 C_2 \ldots C_n$, a multi index is defined as a set of n simple indices (called *index components*) $I_1, I_2, \ldots, I_n$ where I_i is an index defined on $C_i.A_i, (1 \leq i \leq n)$. All indices $I_1, I_2, \ldots, I_{n-1}$ are identity indices, that is, they have OIDs as key values. This definition is similar to the one found in [FMV94]. The last index, I_n, may or may not be an identity index depending on the domain of the attribute A_n. Thus, the multi index tries to provide an index at every basic component of the path thus indexing the entire path and hence the all path instantiations. Each of the indices are generally B^+ tree indices but could use other techniques as well depending on the nature of the path instantiations. A sample multi index organisation is shown in Figure 5.5.

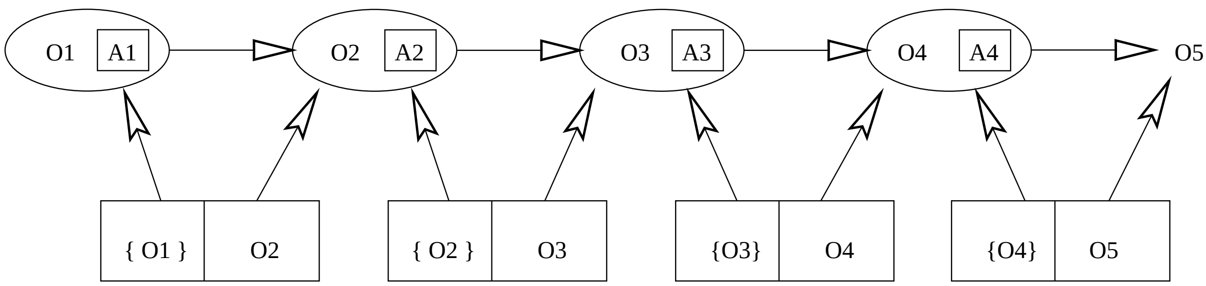

Figure 5.5: Multi index

Thus, a lookup requires reverse traversal of a number of indices equal to the path length of the path instantiation. Consider the single path instantiation as shown in Figure 5.5. Performing a lookup involves traversing the index keyed on value O_5 and obtaining the object references (OIDs) to instances which satisfy the criterion. For each OID in this set obtained (in this case, only O_4) a lookup is make in the adjacent index to obtain the set of references satisfied by the lookup. A similar process is followed backwards until the root node of the path instantiation is encountered. Now, the set of instances obtained from the root index gives the resultant set of all objects that satisfy the criterion of having the value equal to that in O_5.

Insertions and Deletions

Insertions of a path instantiation are handled by updating the entries along all the indices along the path to accommodate the new set of values. Thus, the insertion operation requires the insertion of a value in a number of indices which is equal to the path length of the path instantiation. The deletion operation is similar in that, it requires the deletion of the corresponding entries in all the indices that have information about the path instantiation being deleted. Thus, deletion also requires traversal of a number of indices equal to the path length of the path instantiation being deleted.

Partial Instantiations

The multi index organisation stores information about both left and right partial instantiations in addition to complete path instantiations. Right partial instantiations are recorded with the help of a NULL key value mapped to instances of the last class in the path. Left partial instantiations, however, require no such special identifier and are accommodated by the respective indices which span the partial instantiation.

Retrievals and Updates

Updates in the multi index organisation are the least expensive among the organisations discussed. This is because it requires updation of only a single simple index unlike in the cases of the path and nested index which required traversals along the path instantiation. An object updation thus is done *locally* in the sense that it is local to the basic component of the path in question and is independent of the other components that make the path. However, it can be seen that retrievals are costly due to the necessity of traversal of a number of indices equal to the length of the path instantiation in question. It may also be noted that this organisation facilitates queries that require *reverse traversal*.

5.3.4 Multi index with path configuration

Structure

To motivate the necessity of a path configured multi index, we summarise the characteristics of the nested, path and multi index. Nested and Path indices have high update costs especially if the path indexed is of length greater than 3. However, they have low retrieval costs. In contrast, the multi index has high retrieval costs while having low update costs. Thus, it can be seen that an intermediate solution can be obtained by splitting the path into subpaths and if these subpaths are optimal for the given path, the lowest cost solution can be obtained.

An algorithm for finding the optimal configuration can be found in [Bert94]. This algorithm takes as input the frequency of various operations (insertion, deletion, retrieval) for the classes along the path besides taking into account whether reverse references are maintained along the path. Then the algorithm outputs the optimal subpath configuration for the given path and also assigns the type of index (nested, path or simple) that should be assigned to each subpath. Most importantly, the algorithm decides if some subpath does not require any index. The algorithm also identifies circumstances under which the absence of indices along a path may turn out to be an optimal configuration. This happens when there is minimal sharing of subpaths between various paths and reverse references exist along the path. Thus, reverse references maybe the best solution in systems that possess the above characteristic and are constrained in terms of space which is a primary requirement in order to be able to build indices.

For the purpose of illustration, Figure 5.6 shows indices allocated to subpaths as a possible result of application of the algorithm to a path having a higher frequency of operations on the entire right partial instantiation represented by the subpath $C_1.A_1.A_2.A_3$. The subpath represented by $C_3.A_3.A_4.A_5$ has a low frequency of operations on the entire subpath. Thus, it can be seen that the right partial instantiations mentioned in this

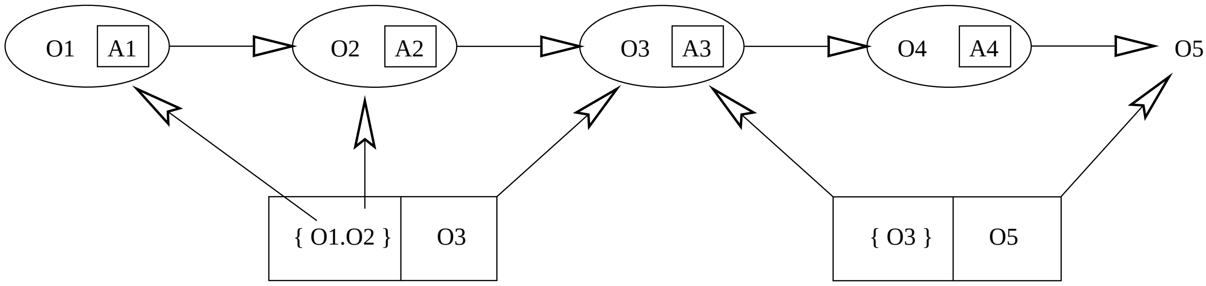

Figure 5.6: Multi index with path configuration

example are indexed by a path index while the left partial instantiations are indexed by a nested index.

Insertions and Deletions

Insertions require addition of the subpaths of the path instantiation into the respective indices allocated for the path instantiation. Deletions, similarly, require the subpaths (of the path instantiation being deleted) to be deleted from the corresponding indices.

Partial Instantiations

This multi index organisation may store left partial or right partial instantiations depending on the path organisation. They differ from the simple multi index organisation in that all nodes along a path may not be indexed. Thus, the entire partial instantiation may not be stored.

Retrievals and Updates

Retrievals are cheaper than in the multi index case since the number of indices that need to be traversed in the retrieval is smaller in this case. The retrieval operation itself is similar to the one in the multi index case where the key value for the search is used to obtain the list of OIDs of instances (along path instantiations that map to this key value) from the index at the end of the path instantiation. This list is then used on the relevant index at this portion of the path instantiation to obtain OIDs that further aid in the *backward* construction of the path. This process is continued till the root of the path instantiation is obtained. Updates are cheaper than in the case where only a nested or a path index has been used. This is because any update requires forward (and possibly reverse) traversal only on the subpath indexed and not on the entire path instantiation. Thus, it can be seen that the multi index with path configuration provides a structure wherein a tradeoff is obtained with respect to retrievals and updates. The main disadvantage of this organisation is that, to obtain a optimal path configuration, statistical data is required which may not be readily available or may not be a precise indicator of the frequency of operations.

5.4 Inheritance based access methods

In this section, we survey techniques used to address the a problem orthogonal to the problem of indexing aggregation based queries. The data model allows a hierarchy of classes to be involved in such a relationship. Since an attribute of a class C is inherited by all classes in the hierarchy rooted at C, the scope of a query directed at the class C is, in general, an inheritance hierarchy rooted at C. An example of such an inheritance hierarchy is represented as a query graph shown in Figure 5.7.

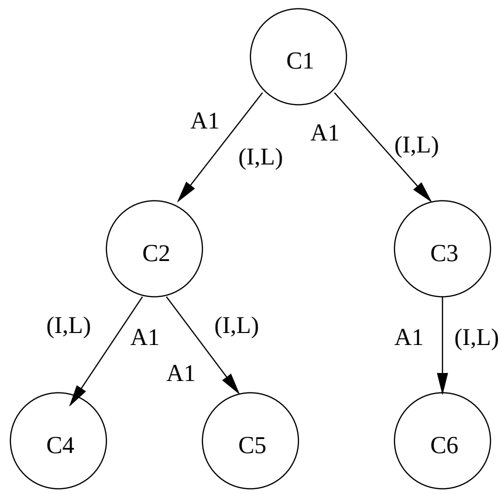

Figure 5.7: A query graph of an inheritance hierarchy

It can be seen from the figure that an attribute A1 of the class C1 is inherited by all subclasses of C1 and subsequently their subclasses. A solution to index such an organisation of instances is to maintain a separate index for instances of each class. This organisation is called a *multi-level single-class* index. However, this organisation does not take into account the relationships that exist among the objects and hence more sophisticated mechanisms exist some of which are surveyed here.

5.4.1 The CH tree

Structure

The CH tree [KKD89b] or the class-hierarchy tree is an index organisation that attempts to maintain an index on an attribute for all instances of all classes in a class hierarchy rooted at a particular class. Thus, unlike in the single-class index, one index is maintained for an entire inheritance hierarchy.

The CH tree is essentially a modification of the B^+ tree. The internal nodes of the CH tree are similar to that of the B^+ tree. The leaf nodes differ from that of the B^+ tree in that they attempt to store information about an entire hierarchy. The structure of a leaf node of the CH tree is shown in Figure 5.8.

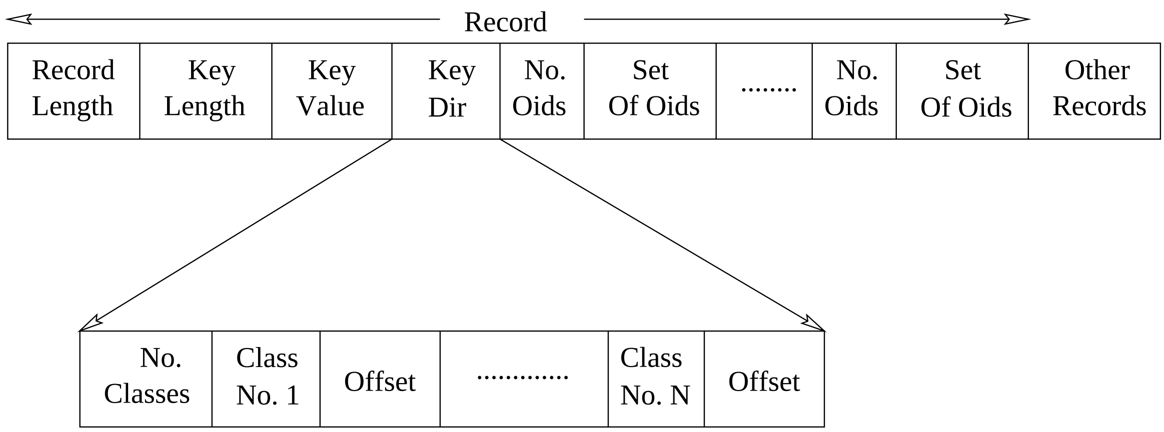

Figure 5.8: Leaf node in a CH tree

The leaf node consists among other information a key directory. The key directory contains the number of classes that contain objects with the key value in the indexed attribute. For each such class, the class identifier and the offset of the list of OIDs of objects of the class are specified. The OIDs are themselves grouped according to their types. This organisation facilitates queries directed at particular classes in the hierarchy as well.

Insertions and Deletions

Insertions are handled by inserting the OID into the corresponding list if one exists. A new list is created if no list is found. If the size of the index record becomes greater than that of an index page, additional pages are allocated to it and an overflow page pointer is maintained in the first page (not shown in the figure). Deletions are done by deleting the corresponding OID from the list. It can be seen that this organisation facilitates deletion of all instances belonging to a particular type as well. Merging of index pages can be done if the size of an index record is greater than an index page before a deletion and becomes less than an index page after the deletion.

Retrievals

Performance studies based on a cost model [KKD89b] have shown that the CH tree performs better than a single-class index if the number of classes in the hierarchy is at least two. In terms of disk space used for the storage structure itself, a conclusive result could not be found. The CH tree organisation itself is uniform in its treatment of queries against class hierarchies and against single classes in the class hierarchy. Queries that are directed against a particular class in the class hierarchy follow the same retrieval procedure as those for retrieving an entire hierarchy. Hence, the index page containing the list of OIDs for the class in question is obtained along with all the other classes in the hierarchy which is an overhead for this nature of queries. Hence, in general, CH trees perform efficiently against queries directed at the hierarchy itself and may not be a cost effective solution for queries directed at particular classes in the class hierarchy.

5.4.2 The H-tree

Structure

The H-tree [LOL92] attempts to alleviate some of the disadvantages of the CH tree. The H-tree organisation tries to preserve the superclass-subclass relationship while trying to improve the performance for single-class queries. A H-Tree structure is maintained for each class of a class hierarchy and the trees are nested according to their superclass-subclass relationship.

The H-tree itself is a modification of the simple B^+ tree index. The internal nodes of the H-tree differ from that of the B^+ tree in that they contain pointers to nodes of H-trees of subclasses in addition to storing key and the corresponding node pointers. The leaf node itself is similar to that of the B^+ tree node. The internal node of an H-tree is shown in Figure 5.9.

No Keys	B1	K1	.....	Kn	Bn+1	No Of Nodes	Set Of Pointers To Nodes

Figure 5.9: Internal node in a H tree

The internal node has all the fields that can be seen in a B^+ tree and in addition to those is present a count of the number of references to corresponding nodes of other H-trees representing the subclasses and the pointers themselves. Two classes involved in a superclass-subclass relationship would thus have two such trees allocated for them with added associations among the trees to facilitate hierarchy traversals. The internal node of the tree representing the superclass would have a direct link to a corresponding internal node in the tree representing the subclass. These links thus create a hierarchy of index trees. The range of values that fall under the subtree rooted at the referenced node can be determined from its parent node. This extra reference is generally avoided by maintaining the minimum and maximum values of the node along with the reference in the superclass tree node itself.

Insertions and Deletions

Creation of the H-tree begins with the creation of the tree component for the most specialised class and so on till the most generalised class. Insertions are made in a manner similar to the B^+ tree and nodes may split which may require readjusting in the set of pointers it contains in addition to readjustment in the set of pointers in the superclass tree node that references this node. Deletion causes similar readjustments if it results in merging of nodes.

Retrievals

Retrieving instances along a hierarchy involves complete traversal of the index maintained on the class at the root of the hierarchy and partial traversal of indices maintained on the subclasses taking advantage of the links provided. Queries directed at single classes are handled by referencing the corresponding index for the class.

This organisation essentially improves on the single-class index by providing links between the trees of classes involved in a superclass-subclass relationship. While the single-class index made no special provision for class hierarchy queries, the H-tree provides such an access which only requires partial traversal of trees for the subclasses involved. The H-tree removes the disadvantage of the CH tree which read and discarded information about irrelevant classes (in a query directed against a single class). For such queries, only the relevant class tree is accessed and hence no irrelevant information is read. However, for queries on entire hierarchies, the H tree organisation may need to make more accesses than with the CH tree.

5.4.3 The hcC tree

Structure

The hcC tree [SS94] is an index structure proposed which can handle both single-class based as well as hierarchy based queries efficiently. The hcC tree is a modification of the B^+ tree index. The internal nodes are very similar to those of the B^+ tree except for an additional bitmap at each node giving advance information and to prevent an unnecessary traversal of the tree.

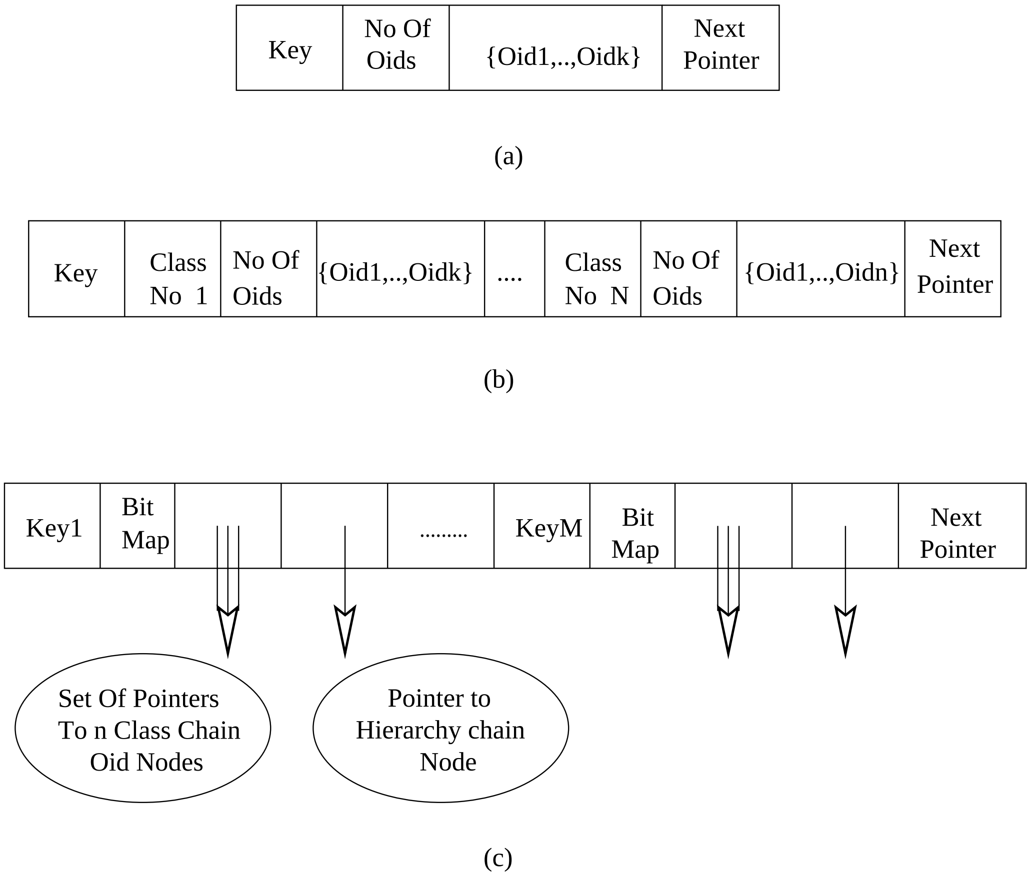

Figure 5.10: Nodes in a hcC tree

This bitmap stores information on whether a particular class has any instances at all. If not, a complete traversal of the tree is not required to find a null answer. The leaf node of the tree maintains the key value, the bitmap and a set of n+1 pointers. Of these n+1 pointers, n pointers point to the instance list of classes that make the class hierarchy and one pointer points to the instances that make the hierarchy itself. The nodes that the n+1 pointers point to are called OID nodes. OID nodes representing single class instantiations are called *class-chain* OID nodes while the OID nodes representing the hierarchy are called *hierarchy-chain* OID nodes. These OID nodes themselves form the last level in the hcC tree and corresponding OID nodes are chained together. That is, the OID nodes representing a class C are all chained together and all the hierarchy nodes are chained together as well. The Figure 5.10(a) shows a class-chain OID node, Figure 5.10(b) shows a hierarchy chain OID node and Figure 5.10(c) shows a leaf node of the hcC tree.

Insertions and Deletions

Insertions into the hcC tree involve inserting the value at the class-chain OID node as well as the hierarchy-chain OID node. Deletions cause a similar operation where the entry is deleted from the class-chain as well as the hierarchy-chain node. The advantage of the hcC tree is that it facilitates both single class queries as well as hierarchy queries. Both types of queries require the traversal of one index tree (unlike the H trees) and retrieval of the corresponding hierarchy or a subset of the classes in the hierarchy.

Retrievals

The performance results obtained in [SS94] showed that the H-trees, in general performed poorly against the hcC tree in the context of hierarchy based queries while the CH tree performed poorly in the context of single class based queries. Thus, it can be seen that the hcC tree provides a efficient mechanism in systems which encounter both types (hierarchy and single class based) queries.

Chapter 6

Geographical Queries

The geographical data consists of information about bio-geographic provinces, biomes, administrative organizations, forests, botanical gardens and herbariums. Each of the administrative organizations, forests, etc., has a map associated with it. The maps are stored as polygons which approximate the region. Processing of queries on these maps requires mathematical computations. Vaidya does not support any mathematical operations other than comparisons. However, as mentioned in the Chapter 4, the user can call some member functions of the class *Polygon*. These functions can be used for doing the required computations.

In this chapter we list out the functions that are supported in the current implementation of Vaidya.

6.1 Load

This function takes a file name as an argument, in the form of string. It reads the polygon specifications from the file. This file should contain the polygon specification in the form of an ordered set of points forming the polygon where each point is a pair of two floating point numbers. The function returns the number of points forming the polygon. An example input is given in Table 6.1.

0.1	1.0
1.0	1.0
1.0	0.1
0.1	0.1

Table 6.1: An Example File

Example:

Load the map of india from the file “indiamap”.

Select cntry.map.Load(“indiamap”)
from Country cntry
where (cntry.name == “India”) ;

6.2 IsPointInside

This takes two floating point numbers X and Y as argument and returns TRUE if the point (X, Y) is inside the polygon, returns FALSE otherwise. The algorithm used to compute is as follows:

- Find out how many line segments that forms the polygon intersect with the line segment (X, -90).
- The point is inside if the number of intersection is odd, the point is outside if the number of intersections is even.

(This algorithm does not works if polygon contains south pole).

Example:

print the country name containing the point (15.0, 40.0).

Select cntry.name
from Country cntry

where (cntry.map.IsPointInside(15.0, 40.0) == TRUE) ;

6.3 IsCompletelyInside

The arguments for this functions are two points (X_1, Y_1) and (X_2, Y_2). This returns TRUE if the polygon is completely inside the region represented by the rectangle (X_1, Y_1), (X_1, Y_2), (X_2, Y_2), (X_2, Y_1). Returns FALSE otherwise.

Example:

print all the cities that are in the region (10.0, 30.0) and (20.0, 50.0).

```
Select  city.name
from    City city
where ( city.map.IsCompletelyInside( 10.0, 30.0, 20.0, 50.0 ) == TRUE ) ;
```

Chapter 7

Implementation

In this chapter we discuss about the architecture, implementation issues of the query processing system, and the access methods used in the system.

7.1 Query Processing System

The query processing system consists of three modules, namely *Interface Object Set*, *Schema Table* and *Query Processor* as shown in Figure 7.1 (this figure is a reproduction of a portion of the Figure 3.2). The *Query Processor* is designed to support any object oriented application running with EXODUS. The query processor requires *Interface Objects* to interact with the database. An interface object provides a set of methods to perform operations such as, retrieve, access an attribute, etc., on the objects in the database of the class corresponding to that interface object. *Interface Object Set* is a set of *Interface Object*s. This set contains an Interface object corresponding to each class in the object model. Thus, the number of the interface objects is the same as the number of classes in the object model. These interface objects are in-memory objects and are part of the query processing system, they are not part of the database, they are created at system start time.

The query processor is also responsible for the checking of semantic validity of the query. For doing this, the query processor refers to the *Schema Table*. Schema table is

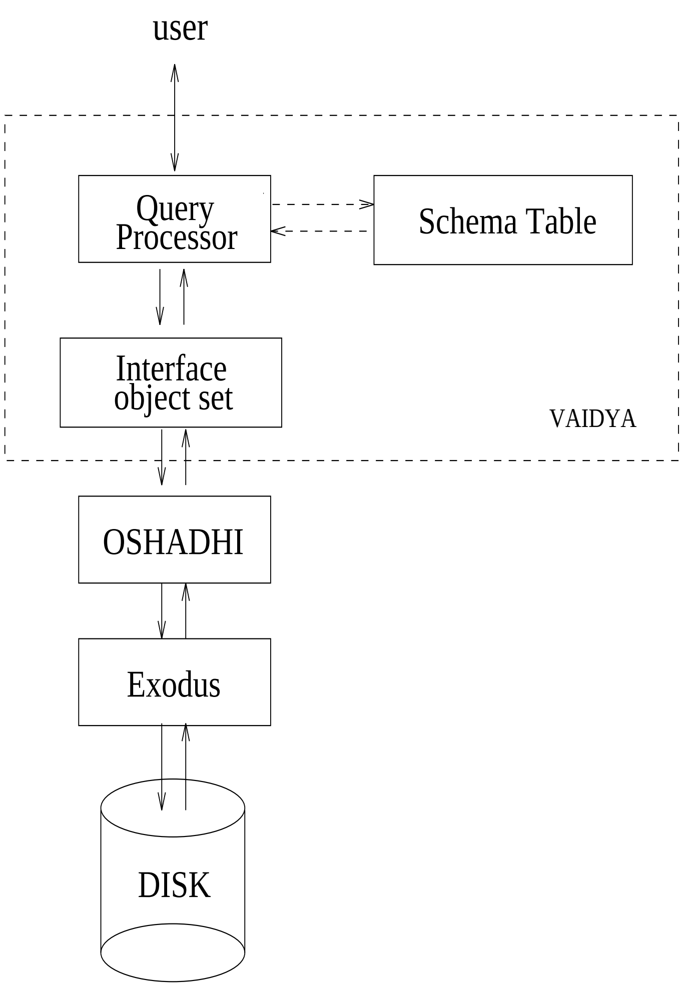

Figure 7.1: The Architecture of OSHADHI

an in-memory representation of the schema of the object model. This in-memory representation is necessary because, even though EXODUS maintains schema information, the program written in E language such as, the OSHADHI system, by the knowledge of a name of an attribute (in the string form) cannot find out the type of that attribute by any means. This effect is similar to what happens with identifier names of a C/C++ programs after compilation. The *schema table* is a collection of *Class Tables*. There is a class table corresponding to each class in the object model. Thus, the number of class tables in the schema table is the same as the number of classes in the object model. A class table is a set of records, each record describing information related to an attribute of the class represented by it. Thus, the number of records a class table has is the same as the number of attributes of that class. A part of the class table corresponding to Species is shown in

Figure 7.2.

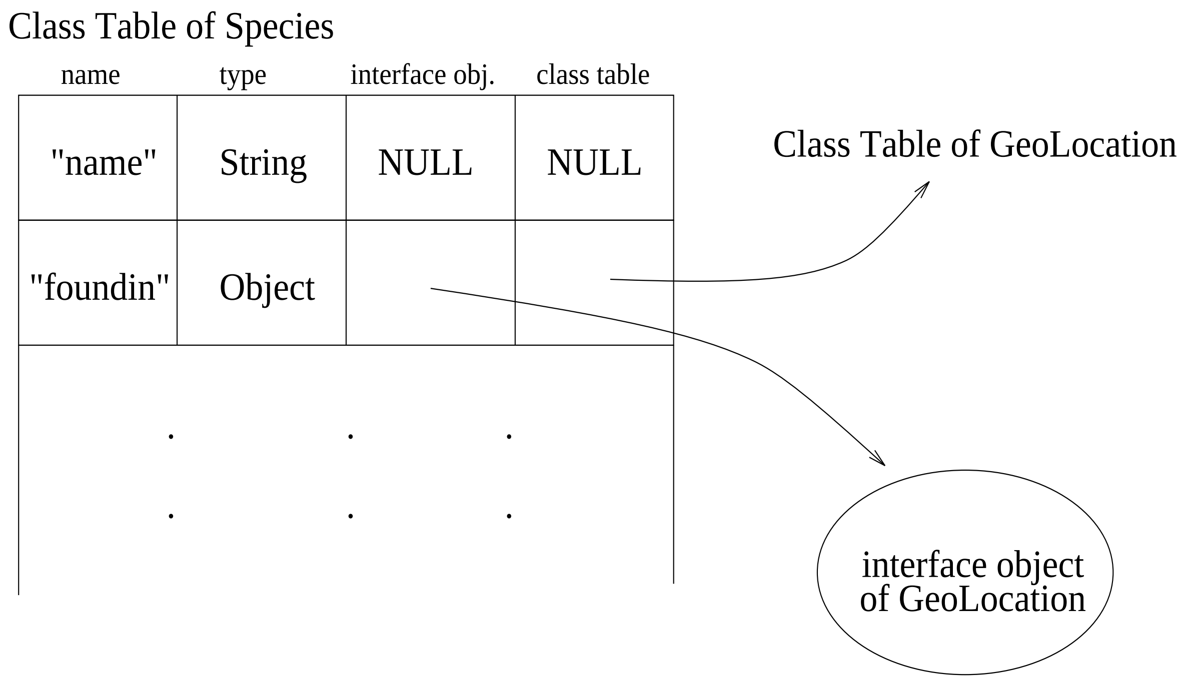

Figure 7.2: Part of Species Class Table

Each record contains name of the attribute, type of the attribute, a pointer to an interface object and a pointer to class table corresponding to the domain class of that attribute. The interface object pointed to by the record is used to perform operations on that attribute. If the attribute type is a basic type other than enum data (string, int, float, etc.,) then the two pointers in the record (the pointer to interface object and the pointer to class object) are set to NULL. In case of enum data type, the pointer to the interface object is set to NULL and the pointer to the class object points to a table, that stores the possible values of the attribute.

As it can be observed, the modules Interface Object Set and Schema Table are schema dependent. Thus, if any changes are made to the schema the programmer has to change these modules also.

7.2 Query Processing

The query is processed in four stages namely, parse, validation, query planning and execution. We will discuss these stages in this section.

7.2.1 Parser

The Parser module of the query evaluator is responsible for the parsing of the input query specification and the creation of a parse tree. The query language grammar itself was specified as input to a parser generator tool (yacc) and a parser was generated. The lexical analyzer, to feed the parser, was generated using a token specification to the lexical analyzer generator (lex). Thus, at run time, the lexical analyzer feeds tokens read from the query specification to the parser, which ascertains the correctness in the syntax of the specification. Thus, syntax checking is done as part of the parsing operation. As the parser scans the input query specification, it builds an internal data structure which transforms the input query into an internal format.

The root node of the parse tree of a query is of the type *QueryClass*. It has three field namely *SelectClause*, *FromClause* and *WhereClause*. The SelectClause field stores a linked list of *PathExpression*s to be printed. Each PathExpression is a linked list of attribute names constituting that path expression. The FromClause field stores variables declared, along with its class. The WhereClause field stores a pointer to the parse tree of predicates.

Example: The parse tree of the query

```
Select  rose.name, rose.foundin.name
from    Species rose
where(  rose.name == "RosaAlba" ) ;
```

is as shown in the Figure 7.3.

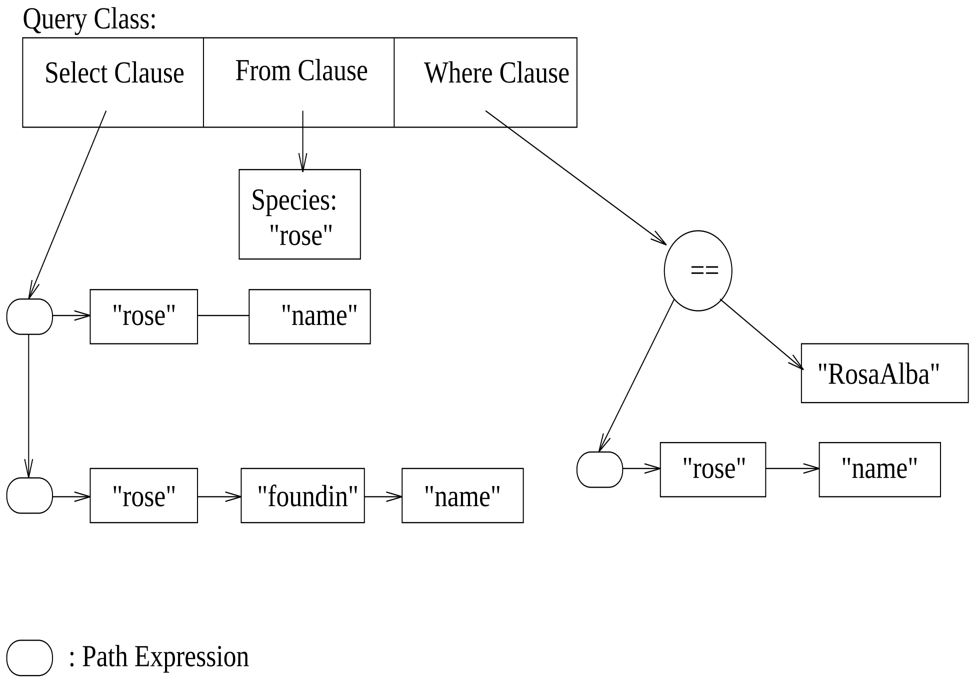

Figure 7.3: Query Parse Tree

7.2.2 Validation

In the validation stage correctness of the input is tested. This includes the following checks:
(i) All identifiers used are declared exactly once in the from clause
(ii) All attribute names in all path expressions are valid
(iii) The attributes being compared are of the same type

Testing for declaration is done using a simple algorithm. Validating path expression is done using the information in the schema table. Finally, checking for compatibility is done by checking whether, types of left hand side and right hand side are same. In case an *enum* data type is being compared with a constant string, a check is done whether that string is one of the possible values of enum data.

In the validation stage apart from checking for validity, the pointers of interface objects that need to be accessed are put in a global set called *display*[1]. This set is used later for planning and executing the query.

[1] A term taken from compilers

7.2.3 Query Planning

The query processor calls the routine *InitFetch( )* provided by interface objects. These routines do the query planning.

7.2.4 Execution

After query planning the query processor fetches objects from the database, checks whether the predicates hold, and prints the results. The routine *GetNext( )* is used to scan through the objects and *Print( )* is used to print the attributes of selected objects. The Delete and Relate operations are performed in a similar fashion. Insertion operation is done using *Insert( )* routine provided by the interface object.

7.3 Accesses methods

We have implemented some of the access methods discussed in Chapter 5. We will list those access methods in this section. **(The source code of these indexing methods are taken from krishna's work [Krish96])**

Simple Value Based access methods

To answer the queries based on simple values (literals) the $B^{+}tree$ index provided by EXODUS is used extensively. The trees were built using the names (string) as key.

Aggregation based access methods

For bio-diversity domain, updates are expected to be infrequent. Therefore, the bottleneck is expected to be retrievals. Hence, nested index is chosen.

Inheritance based access methods

The hcC tree is implemented to address the issue of inheritance based queries in bio-diversity database. This was done since the hcC tree performs well against both single class and class hierarchy queries.

Chapter 8

Conclusions

The aim of the project was to give the prototype for managing the bio-diversity data and answering the queries on it. OSHADHI-1 addressed the problem of modeling the bio-diversity data and providing a simple interface to it.

In our work, we made some improvements to the OSHADHI-1 system, We have implemented a query processor over OSHADHI system to process more complex queries. We added some newer indexing methods to improve the performance of the system. We also provided some functionalities that are required to process geographical queries.

The basic design of the query language Vaidya is inspired by Object Query Language(OQL). Vaidya differs from OQL by allowing member function calls. This member function call is used to provide support for geographical queries.

8.1 Future Work

Data Definition Language

The schema of bio-diversity data may change over time. In the current implementation the only way to change the schema is to change the source code of the OSHADHI system. One may design a Data Definition Language(DDL) for defining the schema of the bio-diversity data.

Two modules of the Vaidya query processing system namely, *Interface object set* and

Schema Table are dependent on the schema of the object model. However, generation of these modules from a DDL, may be automated.

Conversion from DELTA Format

DELTA, the DEscriptive Language for TAxonomy [Poun95], is a data format designed for use by non-computer specialists working in the area of biological sciences. Since this format is widely used by the biologists, it would be useful to provide an automatic conversion tool from the DELTA format to the OSHADHI database objects.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

[Come79] Comer, D., "The Ubiquitous B-Tree", *ACM Computing Surveys* 11(2), June 1979.

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

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

[Catt94] R. Cattell, *The Object Database Standard: ODMG-93*, Morgan Kaufmann, 1994.

[Kim89] W. Kim, "A model of queries for object oriented databases", *Proc. of the 15th Intl. Conf. on VLDB*, 1989

[Fagi+79] R. Fagin et al., "Extendible hashing - A fast access method for dynamic files", *ACM Trans. on Database Systems*, 1979.

[FMV94] J. Freytag, D. Maier, G. Vossen, *Query Processing for Advanced Database Systems*, Morgan Kaufmann, 1994.

[MS86] D. Maier, J. Stein, "Indexing in an object oriented database", *Proc. of IEEE Workshop on Object-Oriented DBMSs*, 1986.

[Bert94] E. Bertino, "Index Configuration in object oriented databases", *The VLDB Journal*, 1994.

[KKD89b] W. Kim, K. Kim, A. Dale, "Indexing Techniques for object oriented databases", In *Object-Oriented Concepts, Databases and Applications*, W. Kim, F. Lochovsky, eds., Addison-Wesley Publishing company(ACM Press), 1989.

[Vidya95] R. Vidya, "OSHADHI: A Biodiversity Information System", *ME Thesis, CSA Dept, Indian Institute of Science*, 1995.

[Krish96] M. Krishna, Query Processing and Bulkloading in OODBMS, *ME Thesis, CSA Dept., Indian Institute of Science*, 1996.

[LOL92] C. Low, B. Ooi, H. Lu, "H trees: A Dynamic Associative Search Index for OODB", *Proc. of ACM SIGMOD Intl. Conf. on management of data*, 1992.