Database Habitats for Biological Species:

Handling Taxonomy Data in OSHADHI

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

by

RAJARAO TADIMETI

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

January 2000

Dedicated to my loving sisters
Gayatri, Aparna and Anagha

and

to my little niece
Neeharika

Contents

List of Figures

Acknowledgements

I would like to thank my project guide *Dr. Jayant Haritsa* for his excellent e-guidance through out the course of this project. I am grateful to him for teaching me the fundamentals and the research issues involved in Database Management Systems.

I would like to thank my advisor cum project guide *Dr. Ramesh Hariharan* for the freedom that he has given to me in selecting the courses during the course work. I am also grateful to him for teaching me the formalism in computer science and for the helpful hand that he has given during the final stages of my project.

I would specially thank my colleague *Srikanta* for his useful suggestions in the project work and for his ever helping hand. I would express special thanks to my colleagues *Satheesh* and *Madhav* for giving me their support in crucial times and for their encouragement during my project work. I would always cherish the time that I had spent with them. I thank the remaining *DSLites*: *Bablu*, *Maya* and *Vikram* who made DSL a homely place to work.

I would thank my batch-mate *Krishna*, who is my first friend in IISc, for his constant encouragement and the affection that he has shown on me. I would like to thank all *CASLites*, *Manjunath* and *Nitin* for making my stay in IISc a memorable one. I am also grateful to all of my remaining friends in IISc for treating me as a good friend.

Finally, I am grateful to my entire family for giving me constant support and inspiration, without which this project would not have materialized.

Abstract

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

In this thesis, we describe the object oriented design and implementation of taxonomy module in OSHADHI[1] - a biological information system. We have chosen Path Dictionary Index and MT-index as aggregation and class hierarchy indexes respectively. The reasons for these choices are presented and extensions to the Path Dictionary Index method to support N:M relationships are explained.

[1] 'plant kingdom' in Sanskrit

Chapter 1

Introduction

1.1 Motivation

OSHADHI is a biological information system which is currently under development at Database Systems Lab, Indian Institute of Science. An important component of *bio-diversity data* is *taxonomy data* (i.e., information related to various species and their characteristics). Taxonomy classification system forms a natural hierarchy consisting of Order, Family, Genera, Species and their characteristics. Hence, an object oriented design seems more appropriate instead of traditional approaches.

Having chosen object oriented design, we have to model the taxonomy data and design efficient access methods for this model. The taxonomy data model has already been done by Vidya [Vid95] on a platform called EXODUS [C$^+$86]. Since we chose SHORE [C$^+$94] as our underlying platform (for the reasons explained in Chapter 5), we decided to port the data model onto SHORE from EXODUS. For designing access methods, we need to be able to handle aggregation and class hierarchies. Within class hierarchies, we need to handle both single class and class hierarchy queries. These involve conflicting requirements, hence careful design is required. We have chosen Path Dictionary Index [LL98] and MT-index [MP97] as aggregation and class hierarchy indexes respectively. While original implementation of Path Dictionary Index supports only 1:1 and 1:N relationships, we need to also model N:M relationships in taxonomy data. Hence, we have extended the

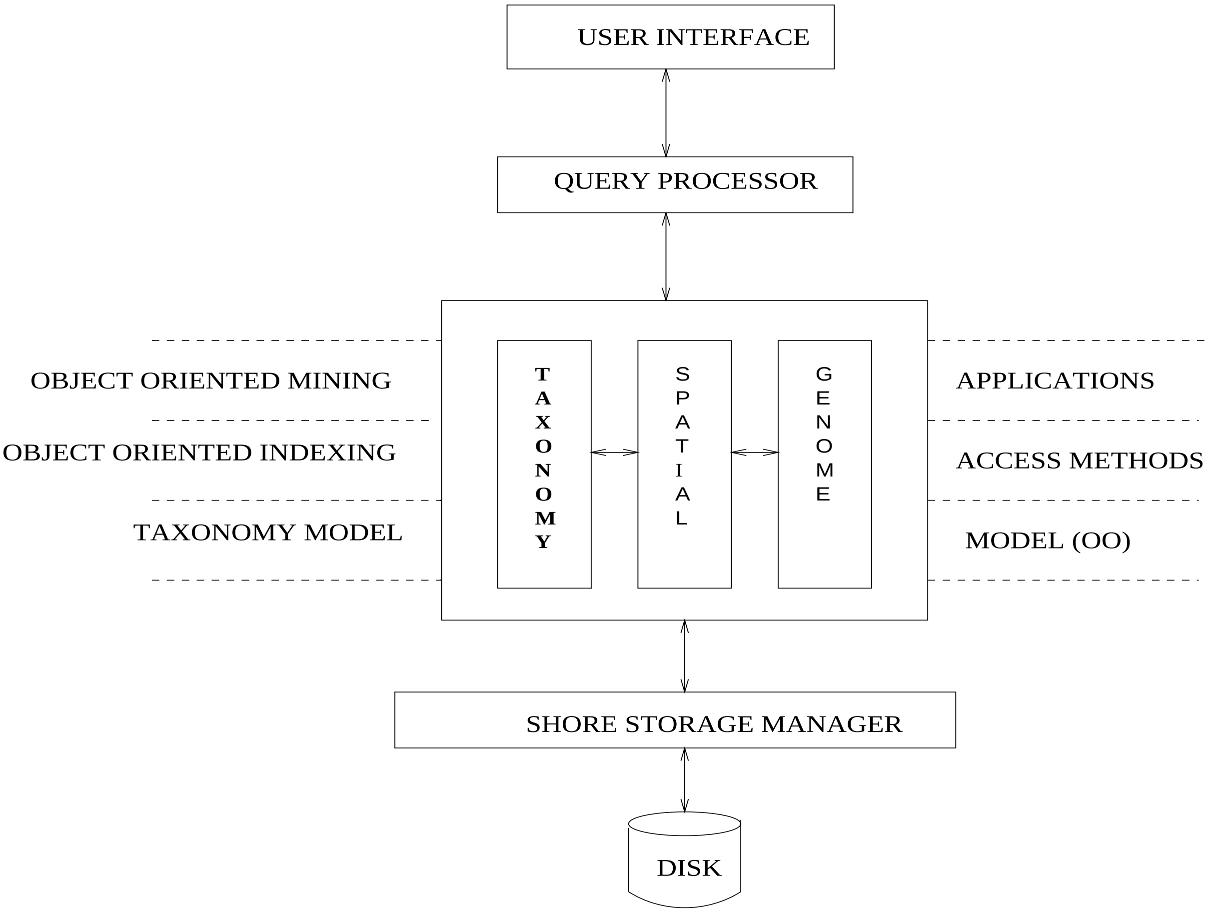


Figure 1.1: OSHADHI System Architecture.

implementation of Path Dictionary Index method to handle this scenario. All these issues are discussed in detail in the remainder of the report.

1.2 The OSHADHI System

OSHADHI is an object oriented biological information system that has been built on an object oriented micro kernel called SHORE (Scalable Heterogeneous Object REpository), developed at University of Wisconsin [C+94]. SHORE alleviates some of the problems suffered by EXODUS, like weak typing of persistent objects, *client level server problem* [C+94] etc. The system architecture of OSHADHI is illustrated in figure 1.1.

OSHADHI system implementation is divided mainly into 5 modules. They are *taxonomy, spatial, genome, query processor* and *user interface*. The taxonomy, spatial and genome modules are responsible for implementing respective data models, supporting efficient access methods, and providing useful applications. These three modules also may interact with each other. For example, the taxonomy module implements the taxonomy data model, access methods and it may support object oriented mining as an application. Taxonomy module will interact with the spatial module and the genome module for retrieving the spatial and the genetic information associated with a species. Query processor does the processing and the optimization of the query and user interface provides the visualization of the query results. End user can either use canned queries or issue custom queries through the user interface.

1.3 Organization of the Report

The rest of the report is organized as follows. Chapter 2 discusses about the object oriented data model for taxonomy. Different class hierarchy and aggregation hierarchy indexing methods which can be used to efficiently retrieve taxonomy data are discussed in chapter 3 and chapter 4 respectively. Chapter 5 discusses about the underlying platform SHORE. Implementation issues are discussed in Chapter 6. Chapter 7 states the current status of the project and possible future work. Chapter 8 concludes the thesis.

Chapter 2

Taxonomy Data Model

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

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

2.1 Analysis

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

2.1.1 The Problem Statement

In taxonomy, each unit of classification is called *taxon*. The four major taxa in the classification of species are *Order, Family, Genera* and *Species*. An order contains several related families, which means the families that exhibit certain similar characteristics are grouped into an order. Likewise, a family is represented by a collection of genera and a genera is a collection of related species. Each taxon has a biological name [Vas74]. The classification hierarchy of species is shown in Figure 2.1

Species are identified based on their leaf, stem, flower and fruit characteristics. Hence,

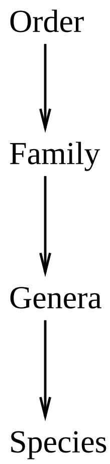


Figure 2.1: The taxonomy of species

these characteristics could be referred to as *identifying characteristics*. There are other characteristics, for example, the characteristic of root, which do not aid in the identification of species. These characteristics could be referred to as *non-identifying characteristics*. A flower is composed of several parts, namely, petals, sepals, inflorosence, carpels, ovaries, anthers, style and stigma. Each of these characters could be used for identification of a species and hence they are also identifying characters. Similarly, a leaf consists parts such as blades, sheath, etc. Hence every order, family, genera and species is associated with a set of identifying and non-identifying characteristics which differentiate it from the other orders, families, genera and species respectively. A species may have synonyms, the geographical location details and the textual information associated with it.

2.1.2 The Queries

In the last section, we described the data and the relationships that need to be maintained in the database. We now focus on the query processing requirements of the biodiversity database. The following are some of the queries that need to be answered by the system:

- *Select an order, genus, family or species and list some/all of its characters*
- *Identify a species based on one or more of the flower/fruit/seed/leaf characters*

- *List all species in a given geographic location*
- *List all species with specific characters in a given geographic location*
- *Display the distribution of a species in a geographic location*
- *List all predators of a given species*

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

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

2.2 The Object Model

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

2.2.1 Identifying Classes

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

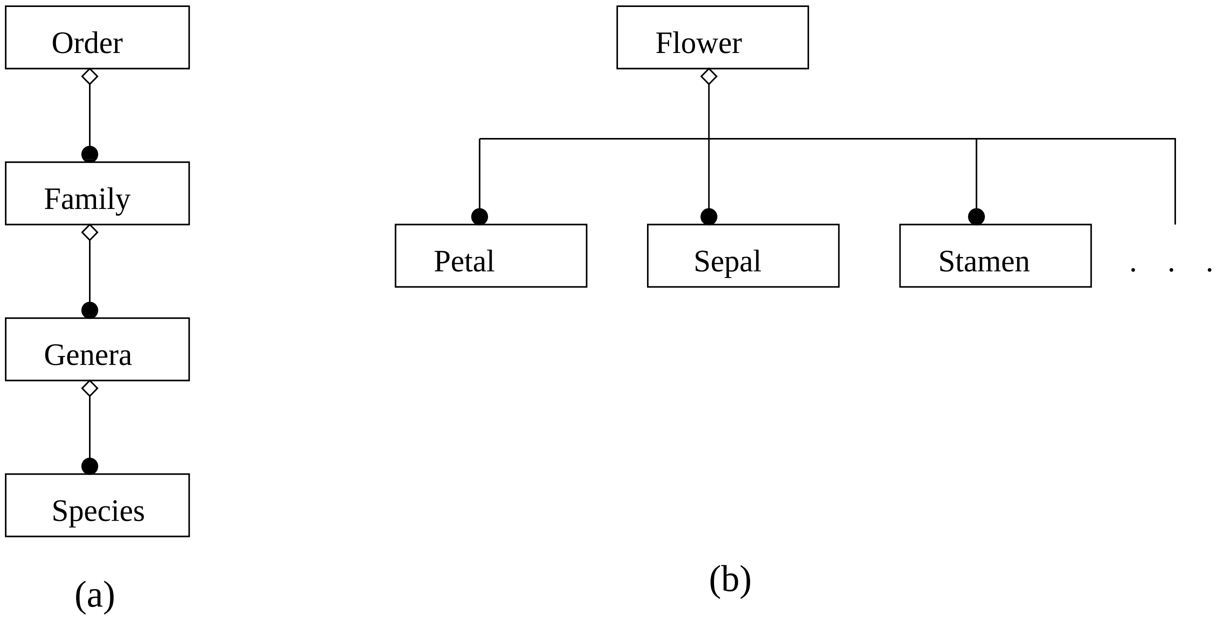


Figure 2.2: Aggregations in taxonomy data model.

2.2.2 Identifying Aggregations

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

- An order *contains* several families. A family in turn *contains* several genera and a genera *contains* closely related species. Since a family cannot belong to two orders, the order-family relationship is modeled as an aggregation rather than an association. Similar arguments hold for genera and species also. Hence, the taxonomy hierarchy is modeled as a four level containment hierarchy in OSHADHI (Figure 2.2a).

- A flower is an aggregation of inflorescence, sepal, petal, anther, stamen, style and stigma(Figure 2.2b).

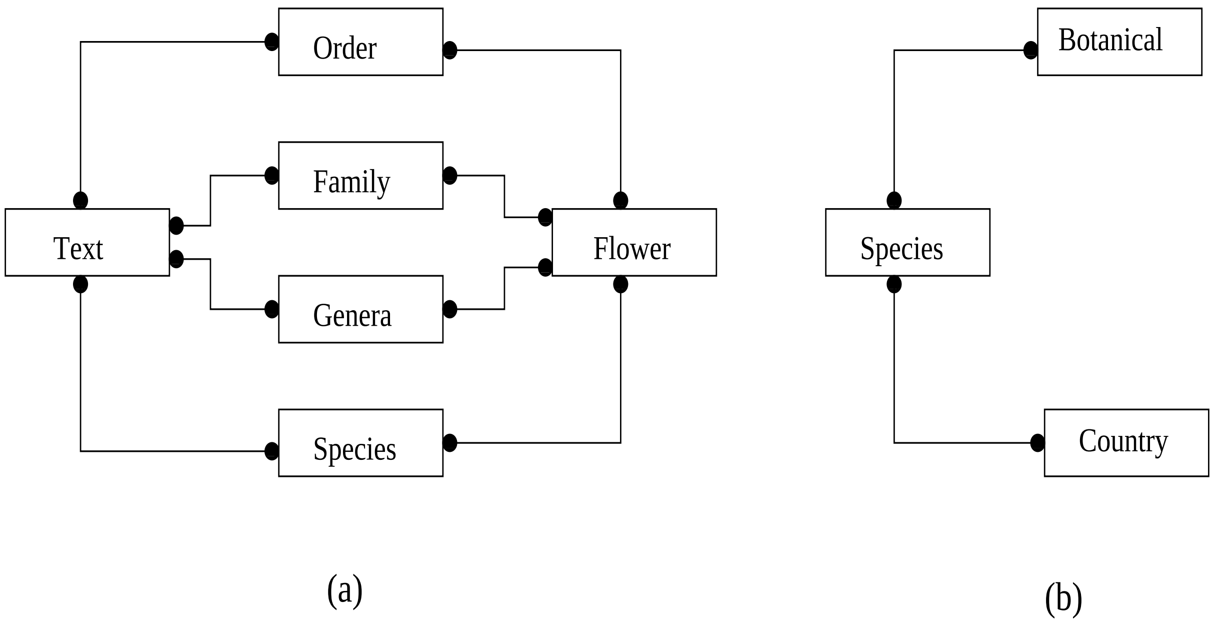


Figure 2.3: Other associations in taxonomy data model

2.2.3 Identifying other Associations

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

- Every order, family, genus and species is associated with some identifying characters. Since taxa exhibit a large amount of sharing of characteristics, an identifying character could be associated with many taxa. Therefore, an identifying character cannot be said as *belonging to* a specific taxon. Hence, the relationship between the taxa and the identifying characters is modeled as an association rather than an aggregation between the corresponding classes. Every taxon is also associated with the text class. The text object models the textual information about any entity (Figure 2.3a).

- A species is found in many geographic organizations and botanical gardens. This feature is modeled as two associations, one between species and geographic organization (could be a country, city, state, etc.), and the other between species and botanical gardens. A geographic organization or botanical garden may have several species associated with it (Figure 2.3b).
- Information about a species is found in many articles and this is represented by the association between the species class and the article class.

2.2.4 Identifying the Attributes and Operations of Classes

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

2.2.5 Organizing and Simplifying Classes using Inheritance

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

The Taxon Class and Ident Taxon Class

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

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

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

Chapter 3

Class Hierarchy Indexing Methods

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

3.1 Basic Problems

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

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

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

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

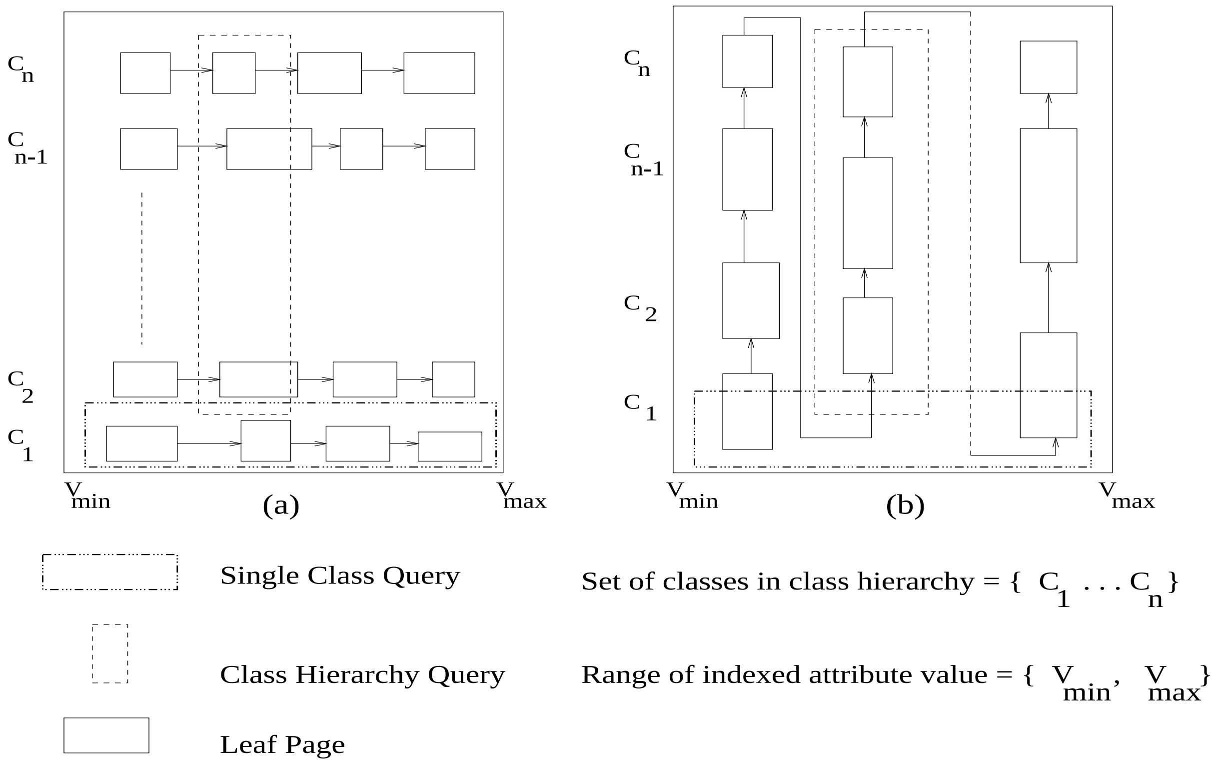


Figure 3.1: Space Partitioning: (a) class-based (b) attribute-based

Space partitioning of both approaches is illustrated in figure 3.1. While class dimension-based approach supports single-class queries efficiently, it is not effective for class-hierarchy queries due to the need for traversing multiple single-class indexes. On the other hand, the attribute-dimension-based approach generally provides efficient support for class-hierarchy queries on the root class, but is inefficient for single-class queries or class-hierarchy queries on a sub-hierarchy of the indexed class hierarchy, as it may need to access many irrelevant leaf nodes of the single index structure.

3.2 Related Work

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

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

3.3 Our Choice: MT-index

To support both single class and class hierarchy types of queries efficiently, the index must support both ways of data partitioning. *Multikey Type index* (MT-index) [MP97] is an example for this kind of approach.

3.3.1 Linearization Algorithm

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

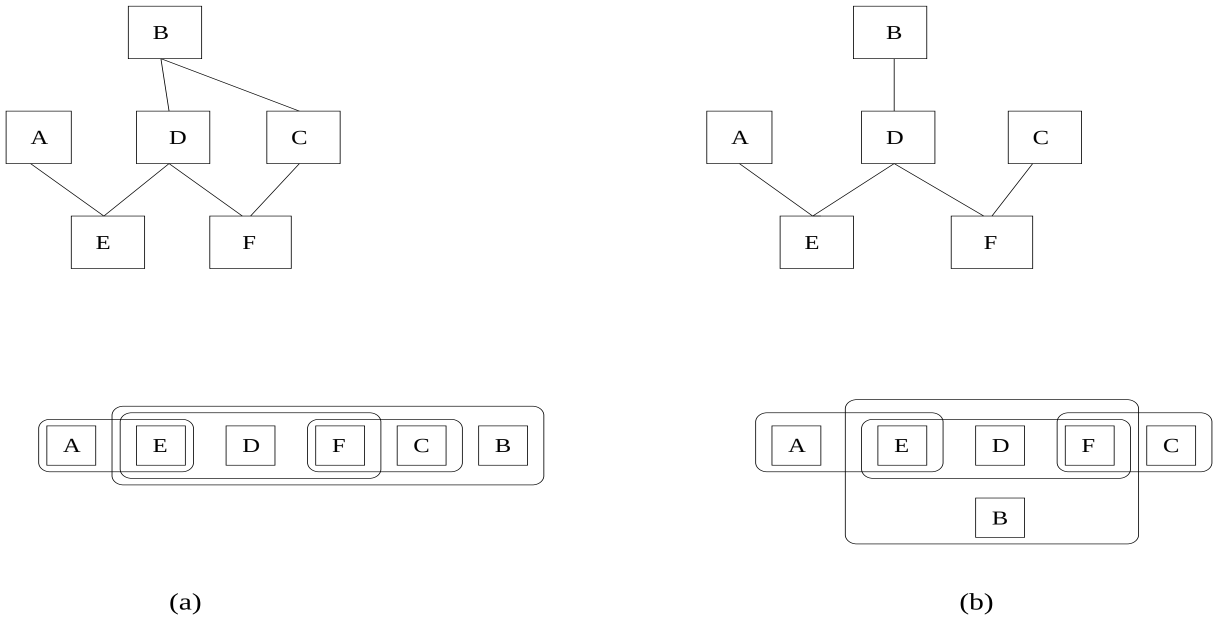


Figure 3.2: Type hierarchies and their corresponding set diagrams (a) with and (b) without optimal linearization.

multidimensional data structure such as hB-tree [LS90] or R^*-tree [B$^+$90] respectively, can be used to implement the index.

3.3.2 Optimal Linearization

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

3.3.3 Necessary and Sufficient Conditions for Optimal Linearization

Sometimes it may not be possible to obtain an optimal linearization of a hierarchy using linearization algorithm. One necessary and one sufficient condition for the existence of the optimal linearization are given in [MP97]. They are:

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

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

In the above conditions, first condition is sufficient condition where as second one is only necessary condition. That means, we can always find an optimal linearization in case of single inheritance and we can never find an optimal linearization if degree of multiple inheritance is greater than two. But if the degree of multiple inheritance is two, it may or may not be possible to obtain an optimal linearization. Example for these two cases are shown in figure 3.2(a) and 3.2(b) respectively. Detailed description of the linearization algorithm, necessary and sufficient conditions for optimal linearization, etc., can be found in [MP97].

3.4 Execution of Queries

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

3.5 Advantages of MT-index

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

- Using two way partitioning of data space, MT-index can answer both single and class hierarchy queries with equal efficiency.
- By extending the number of dimensions, multi-attribute queries can be easily supported with a single index structure. This not only reduces the space overhead, but also improves the retrieval time.
- The linearization algorithm assures good performance by minimizing the query space on which the query is being evaluated.

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

Chapter 4

Aggregation Hierarchy Indexing Methods

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

4.1 Basic Problems

The queries that can be asked on aggregation hierarchies can be classified into the following categories [LL98]:

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

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

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

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

4.2 Related Work

Many of the indexing structures proposed are based on precomputing traversals along aggregation hierarchies. Figure 4.1 illustrates well-known aggregation hierarchy indexing schemes by using the notion of *indexing graph* [BO99]. An indexing graph contains a node for each class in the indexed path; it moreover contains an edge from the node representing a class C to the node representing a class C' if there is an indexing relationship from class C to class C'. An indexing relationship from class C to C' means that an entry in the index contains as key values the identifiers of the instances of class C and it associates with each such key value the identifiers of the instances of class C' that reference, directly or indirectly, the key value. The multi-index [MS86] approach maintains an inverted index for each edge. As can be seen from figure 4.1(a), only backward traversal is supported. The second approach is that based on the well known join index [Val87], where two B^+-trees are maintained for join pairs between two classes. Both forward and backward

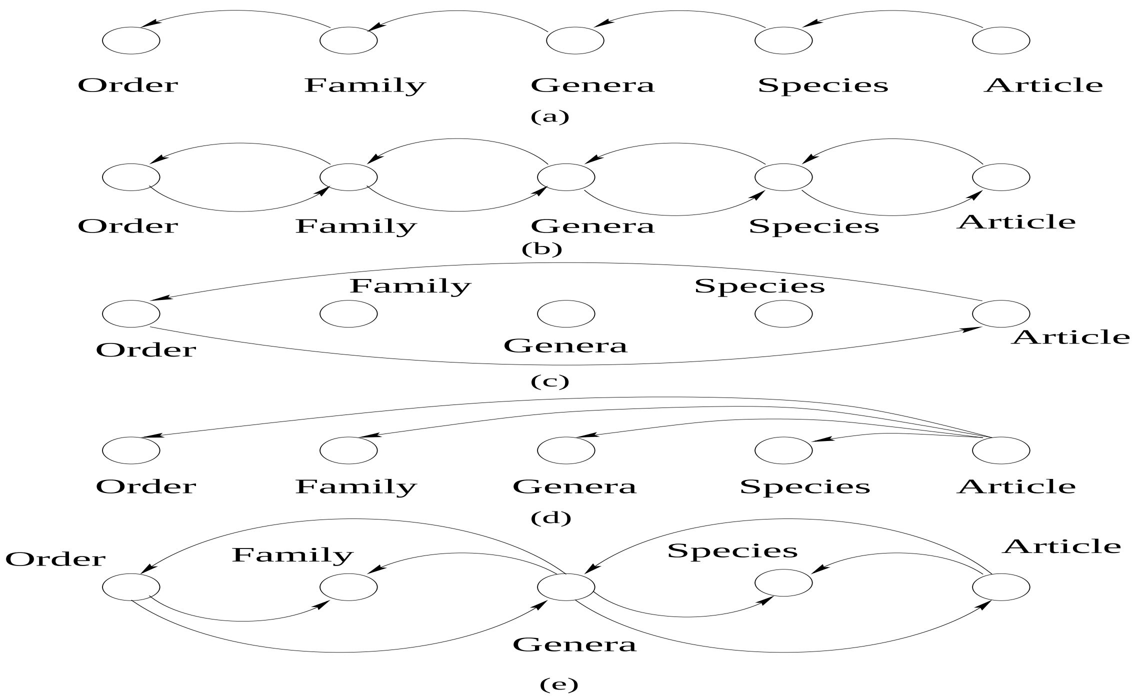


Figure 4.1: Indexing graphs: (a) multi-index; (b) join index; (c) nested index; (d) path-index; (e) path splitting.

traversals are supported, but the update cost and storage cost is much higher than before (see figure 4.1(b)). For both methods, the number of indexes that need to be accessed for navigational access is proportional to path length. To alleviate this problem, direct association between objects of the first class and last class along the path are maintained in a single nested index [EK89]. The major problem of such an index is update operations that require access to several objects in order to determine the entries that need update (see figure 4.1(c)). The nested index can be further generalized to support access from the objects to all parent objects [EK89], as shown in figure 4.1(d). To reduce update overhead and yet maintain the efficiency of path indexing structures, paths can be broken into sub-paths, which are then indexed separately [Ber94]. Figure 4.1(e) shows example where a path is split into several sub-paths and different indexing techniques are allocated on each sub-path.

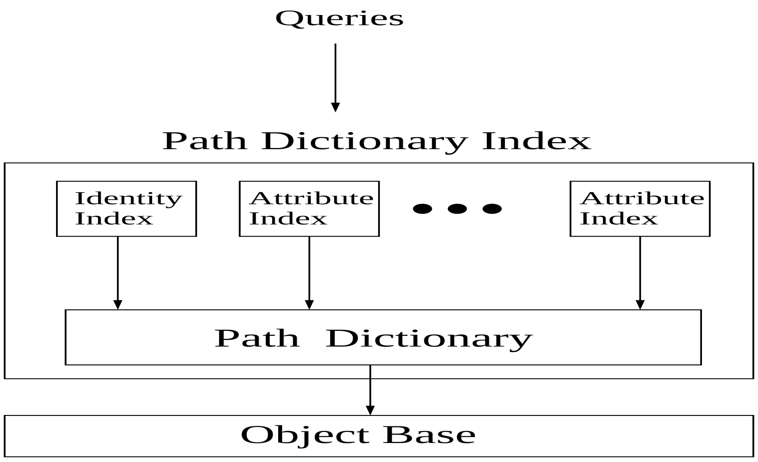


Figure 4.2: Path Dictionary Index

4.3 Our Choice: Path Dictionary Index

Path Dictionary Index (PDI) is recently proposed as an aggregation hierarchy indexing method in [LL98]. The PDI consists of three parts: the *path dictionary* supports efficient object traversal and the *identity index* and *attribute index*es support associative search. The identity and attribute indexes are built on top of the path dictionary. Figure 4.2 illustrates the overall architecture of the path dictionary index.

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

4.3.1 Path Dictionary

An object-oriented database may be viewed as a space of objects connected with links through complex attributes. Figure 4.3(a) shows some object instances corresponding to an aggregation hierarchy. Figure 4.3(b) is a *conceptual* path dictionary storing the connections among the objects in the database. General speaking, the Path dictionary

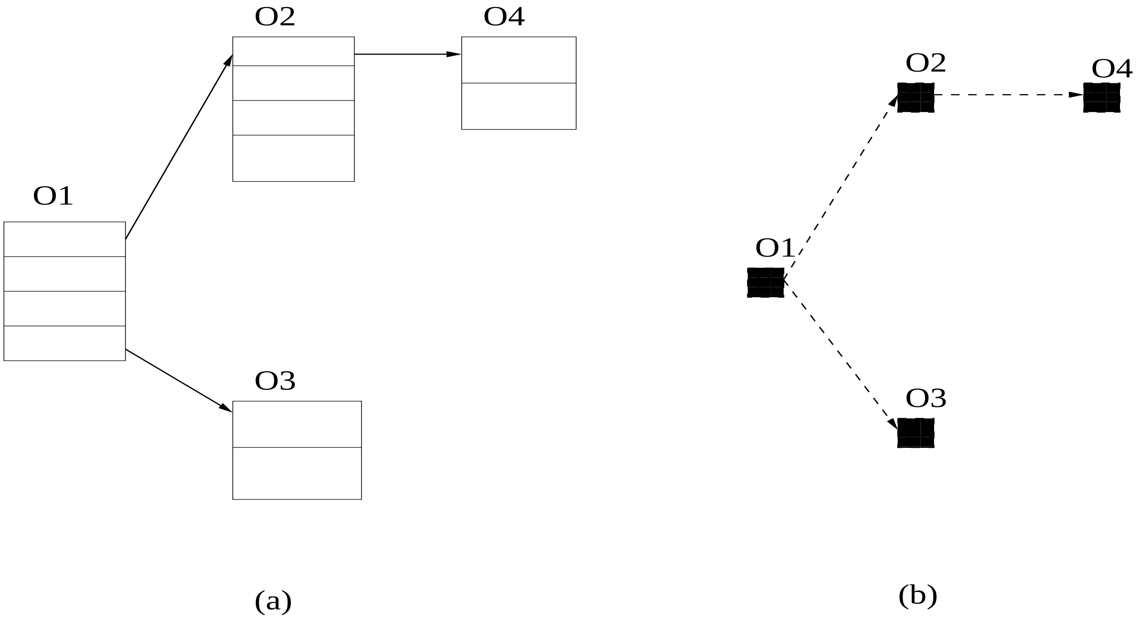


Figure 4.3: (a) A database instance; (b) Path information.

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

4.3.2 Attribute Index

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

4.3.3 Identity Index

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

4.3.4 s-Expression Scheme

Implementation of path dictionary using s-expression schemes is suggested in [LL98]. The s-expression scheme encodes into an expression all paths emanating from the same object. The s-expression for the path $C_1, C_2, ..C_n$ is defined as follows:

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

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

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

4.4 Extensions to Path Dictionary Index

The implementation of path dictionary given in [LL98] works only for either 1:1 relationships or 1:N relationships. But in OSHADHI, the possibility of N:M relationships can not be overlooked. This can be observed from the relationship between the species and their available geographical locations. Note that a species can be found in any number of geographical locations and a single geographical location may contain number of different species. This forms an N:M relationship between species and their geographical location.

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

We extended the implementation given in [LL98] to support each of the above discussed additional requirements with some modifications at the data structure level. The main idea behind the modification is to break the graph of N:M relationship into multiple 1:N relationships. By doing this, we can continue using the s-expression scheme to represent each of the 1:N relationships. But this straight forward modification introduces lot of problems, which are described below in detail with relevant examples. Their corresponding solutions are also proposed.

4.4.1 Extensions to support N:M relationships with sets and references

Look at the example N:M relationship graph shown in figure 4.4(a). If we break this into multiple 1:N relationships, the graphs and the corresponding s-expressions looks like in figure 4.4(b). Note the redundancy in these s-expressions. The children of B_1 are replicated in both of the s-expressions of A_1 and A_2. This problem can be solved by using a flag in the entries of s-expression. The flag denotes whether the entry is a direct reference or an indirect reference. All the descendant entries of an OID will be stored only in the entry which contains direct reference to that OID. This modification is shown in form of a graph in figure 4.4(c) with their corresponding s-expressions. Note the suffix for each entry which denotes whether it is a direct reference or an indirect reference.

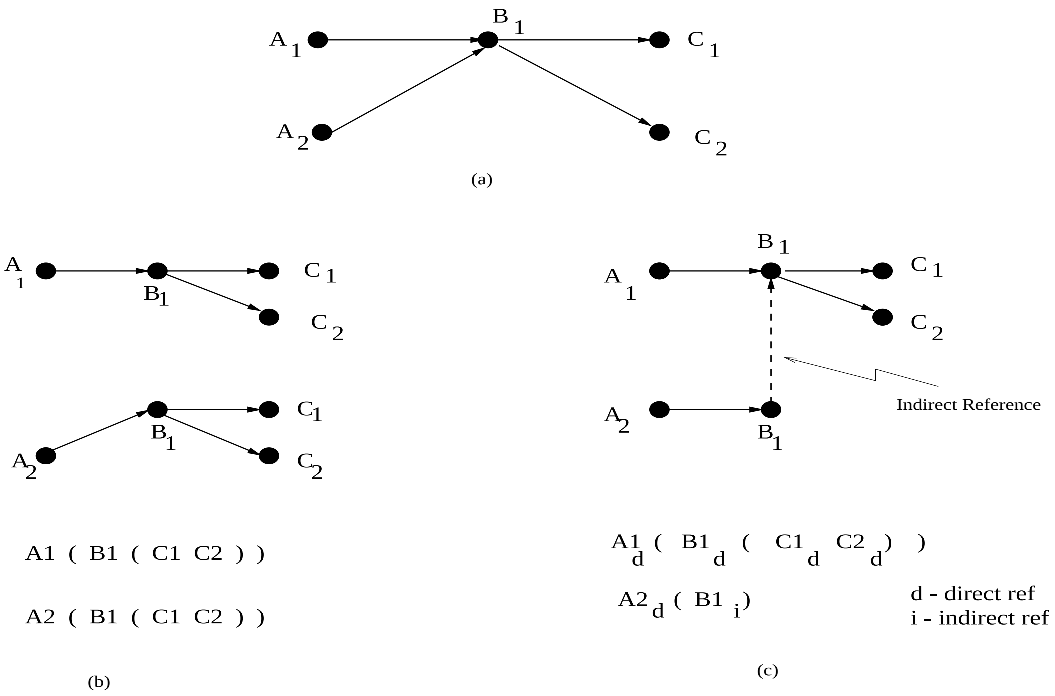


Figure 4.4: Representing N:M relationships and Problems: (a) N:M relationship (b) Equivalent 1:N relationships (c) Equivalent 1:N relationships with indirect references.

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

4.4.2 Extensions to support N:M relationships with bags

The above modification works fine for storing ordinary references and sets. But still redundancy is possible if bags are represented with the above implementation. The example for this is shown in figure 4.5. The number on edge from a to b denotes the number

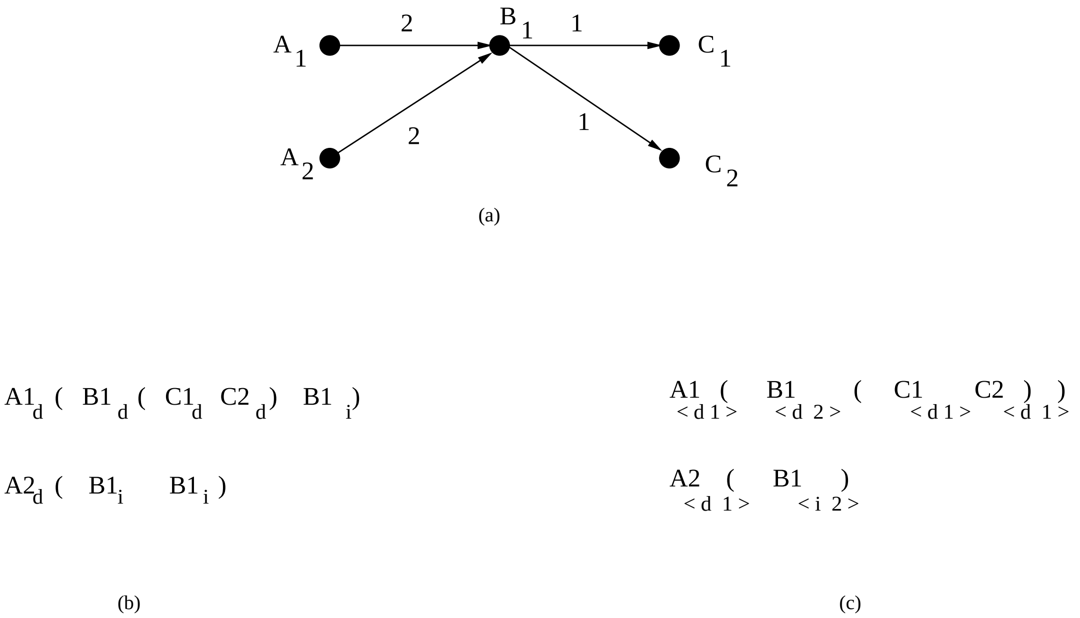


Figure 4.5: Representing N:M relationships and Problems: (a) N:M relationship (b) Equivalent 1:N relationships with just indirect references (c) Equivalent 1:N relationships with indirect references as well as replication count

of times b appeared as reference in the bag of a. The corresponding s-expressions for this graph using above implementation are given in figure 4.5(b). Note that the entry of B_1 is repeated n number of times in each expression, where n denotes the number of times reference to B_1 is contained in parent object. This replication can be eliminated by introducing one more field in the entry of s-expression which stores this replication count. This reduces the storage overhead for storing bags since OIDs are not duplicated. The s-expressions with this modification are shown in figure 4.5(c).

4.4.3 Extensions to support N:M relationships with sequences

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

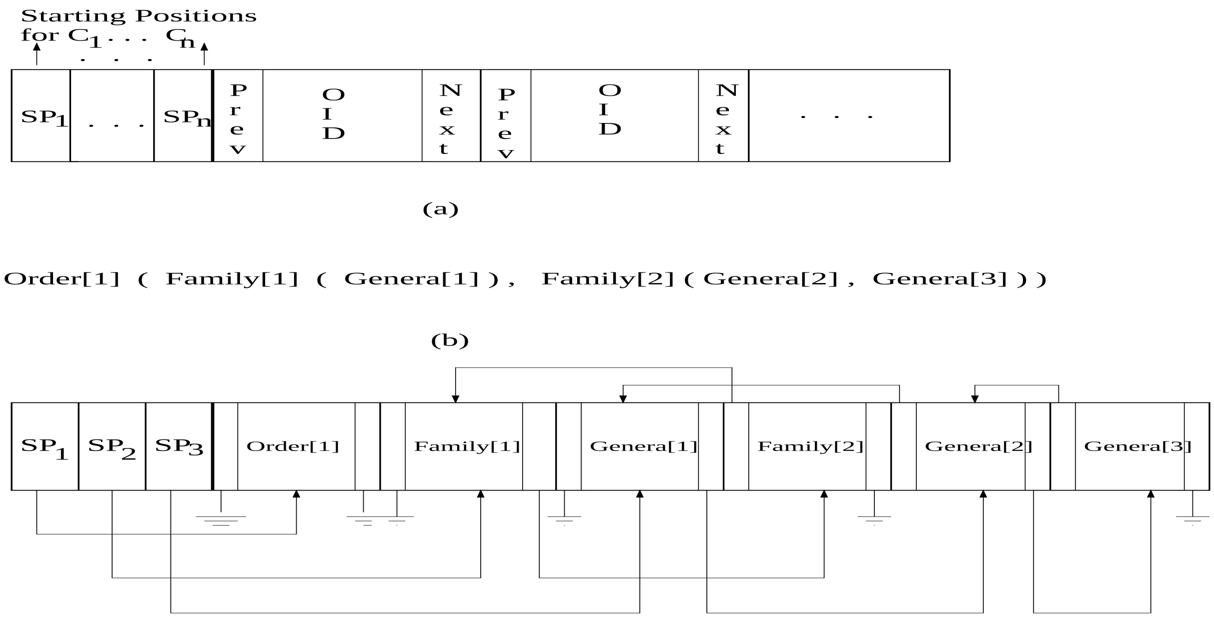


Figure 4.6: Data structure of an s-expression.

4.5 Implementation of the Data Structures

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

4.5.1 s-Expression

Figure 4.6(a) illustrates the data structure of an s-expression for $C_1 C_2 \ldots C_n$. SP_i in the header points to the first occurrence of θ_i in the s-expression. Following the SP_i fields is a series of $< Prev, OID, Next >$ entries. The data structure mimics the nesting structure in the s-expression. The OIDs in the data structure are in the same order as the OIDs in the unwrapped s-expression. The $Prev$ and $Next$ values associated with θ_i points to the previous and next occurrence of θ_i in the s-expression. Using the SP_i, $Previous$ and $Next$ values, we can easily trace the nested relationship among the objects in an s-expression. For example, to obtain the descendant list associated with θ_i, we simply collect the OIDs stored after θ_i until we reach the OID pointed to by θ_is $Next$ value. An s-expression and its representation are shown in Figure 4.6(b) and 4.6(c) respectively.

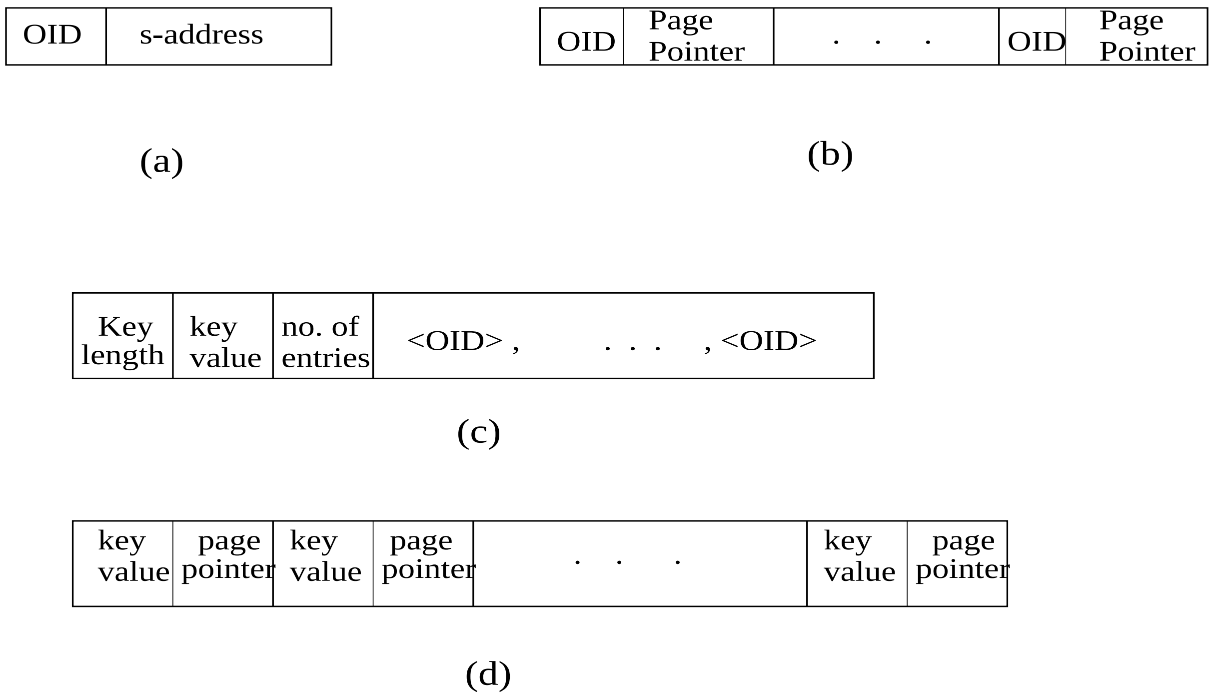


Figure 4.7: Path Dictionary Index: (a) leaf node record of the identity index; (b) non-leaf node of the identity index; (c) leaf node of an attribute index; (d) non-leaf node of an attribute index.

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

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

4.5.2 Identity Index

Figure 4.7(a) and Figure 4.7(b) show the structures of a leaf node record and non-leaf node page for B^+-tree implementation of the identity index, respectively. In the identity index, OIDs are used as the key value. The $s - address$es in the leaf node record are the addresses of the s-expressions corresponding to the OID in the same leaf node record.

The page pointers in a non-leaf node are pointing to the next level of non-leaf nodes or to the leaf nodes. Additional information can be stored in the leaf nodes depending on the requirements.

4.5.3 Attribute Index

Figure 4.7(c) and Figure 4.7(d) show the structures of an attribute index's leaf node record and non leaf node page. In the attribute index, attribute value is used as the key. The OIDs in the leaf node record are the object identifiers of the objects which are having the attribute value in the same leaf node record. The page pointers in a non-leaf node are pointing to the next level of non-leaf nodes or to the leaf nodes.

4.6 Execution of Queries

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

4.6.1 TP Queries

Given the name of the species, find its order.

Assuming an attribute index is available on attribute *species.name* , first the species name is used as key to find all the OIDs of class *species* which satisfies the predicate. These OIDs are then used as keys in the identity index to find the corresponding s-expression addresses in the path dictionary. These s-expressions are then retrieved and scanned to find the OIDs of parent class *order*. Since we are eliminating the need for traversal of intermediate objects belonging to classes *family* and *genera*, the time for evaluating query will get reduced. Due to the structure of path dictionary, the backward traversal of the aggregation hierarchy also becomes easy.

4.6.2 PT Queries

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

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

4.6.3 MX Queries

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

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

4.7 Advantages of Path Dictionary Index

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

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

- Since both forward and backward traversals are supported by the path dictionary, it is possible to evaluate both PT and TP queries with equal efficiency.
- Since intermediate objects need not be traversed, the path dictionary improves the retrieval time.
- Identity index reduces the update overhead by directly pointing the s-expressions where update is to be done.
- Path Dictionary Index can be used to store N:M relationships with the modifications suggested.

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

Chapter 5

SHORE

The SHORE (Scalable Heterogeneous Object REpository) is a persistent object system that represents a merger of object-oriented database (OODB) and file system technologies. In this chapter, we describe features of the SHORE (or Shore) system, which forms the back-end database server of the **OSHADHI** database system.

Most of the material in this Chapter is taken from [C+94] and SHORE manual pages.

5.1 Introduction

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

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

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

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

5.2 Drawbacks with EXODUS

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

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

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

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

5.3 How SHORE differs from EXODUS

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

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

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

5.4 The SHORE Architecture

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

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

5.5 SHORE Software Components

The main software components of SHORE (Figure 5.1) constitute the SHORE server and the Language Independent Library. We next discuss these components in greater detail.

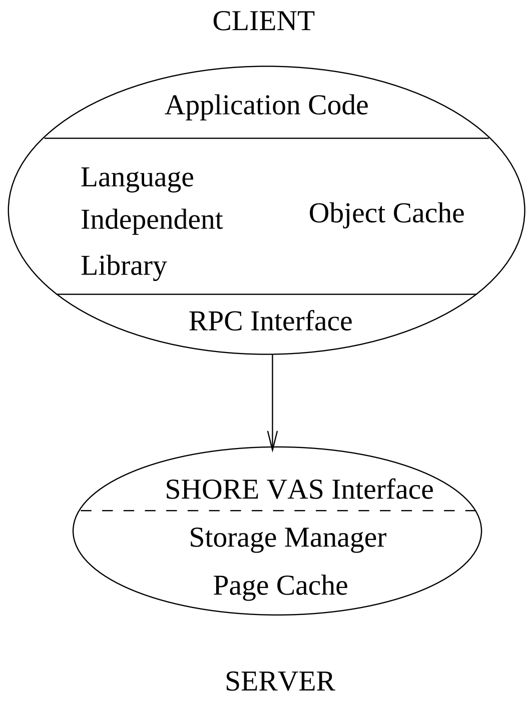


Figure 5.1: Application - Server Interface

5.5.1 The Language Independent Library

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

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

5.5.2 The SHORE server

The SHORE server (Figure 5.2) is divided into two main components: a *Server Interface*, which communicates with applications, and the *Storage Manager* (SM), which manages the persistent object store.

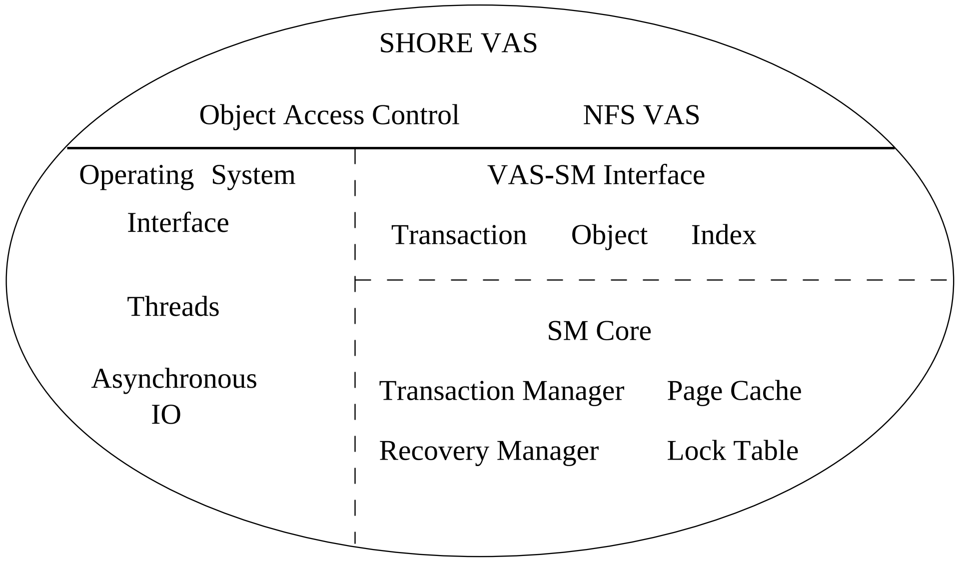


Figure 5.2: SHORE System Architecture

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

5.6 Value Added Server

The SHORE server code is modularly constructed so that users can build application-specific servers, thus supporting the notion of "value-added" servers (VAS). The Server Interface is an example of one such VAS. Another example for VAS is the NFS file server which is used to *mount* the entire subtree of the SHORE name space on an existing Unix file system. When applications attempt to access files in this portion of the name space, the Unix kernel generates NFS protocol requests that are handled by the SHORE NFS value added server.

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

5.7 The VAS-SM Programming Interface

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

5.7.1 Storage Facilities

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

Devices

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

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

Volumes

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

Files of Records

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

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

B^+-tree Indexes

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

R^*-tree Indexes

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

5.7.2 Transaction Facilities

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

Transactions

A *transaction* is an atomic set of operations on records, files, and indexes. The interface provides methods for beginning, committing and aborting transactions. Updates made by committed transactions are guaranteed to be reflected on stable storage, even in the event of software or processor failure. Updates made by aborted transactions are rolled back and are not reflected on stable storage.

Although nested transactions are not provided , the notion of $save - points$ is there. Save-points delineate a set of operations that can be rolled back without rolling back the entire transaction.

Concurrency Control

Transactions are also a unit of isolation. Locking is provided by the SSM as a way to keep a transaction from seeing the effect of another, uncommitted transaction. Normally, locks are implicitly acquired by operations that access or modify persistent data structures, but the SSM interface also provides methods for locks to be acquired explicitly.

The SSM uses a standard hierarchical, two-phase locking protocol [GR93]. For a file, the hierarchy is volume, file, page, record ; for an index, it is volume, index, key-value.

Chained transactions are also provided. Chaining involves committing a transaction, retaining its locks, starting a new transaction and giving the locks to the new transaction.

5.7.3 Crash Recovery Facilities

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

Logging

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

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

Checkpointing

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

Recovery

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

5.7.4 Thread Management

Providing the facilities to implement a multi-threaded server capable of managing multiple transactions is one of the distinguishing features of the SSM. Any program using the thread package automatically has one thread. In addition, the SSM starts one thread to do background flushing of the buffer pool and another to take periodic checkpoints. SSM also provides *latches* which are a read/write synchronization mechanism for threads, as opposed to locks which are used for synchronizing transactions. Latches are much lighter weight than locks, have no symbolic names, and have no deadlock detection.

5.7.5 Communication and RPC Facilities

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

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

5.8 Conclusion

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

Chapter 6

Implementation Issues

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

6.1 Data Model

The data definition language for OSHADHI is based on Object Definition Language (ODL) [Cat93]. Any data model to be represented in OSHADHI should be written in this language. The ODL representation of the taxonomy data model detailed in the chapter 2 will be converted into internal representation by the Query Processor module of OSHADHI. This information is then used by the Query Processor to answer the queries written in OQL (Object Query Language) as shown in figure 6.1. The detailed diagram and discussion on these conversions are beyond the scope of this paper. More details on query processing in OSHADHI can be found in [Kon00].

6.2 Path Dictionary Index

As explained in section 1.2, OSHADHI is being implemented on top of SHORE micro kernel. SHORE supports the notion of ”value-added” server (VAS). The server code is

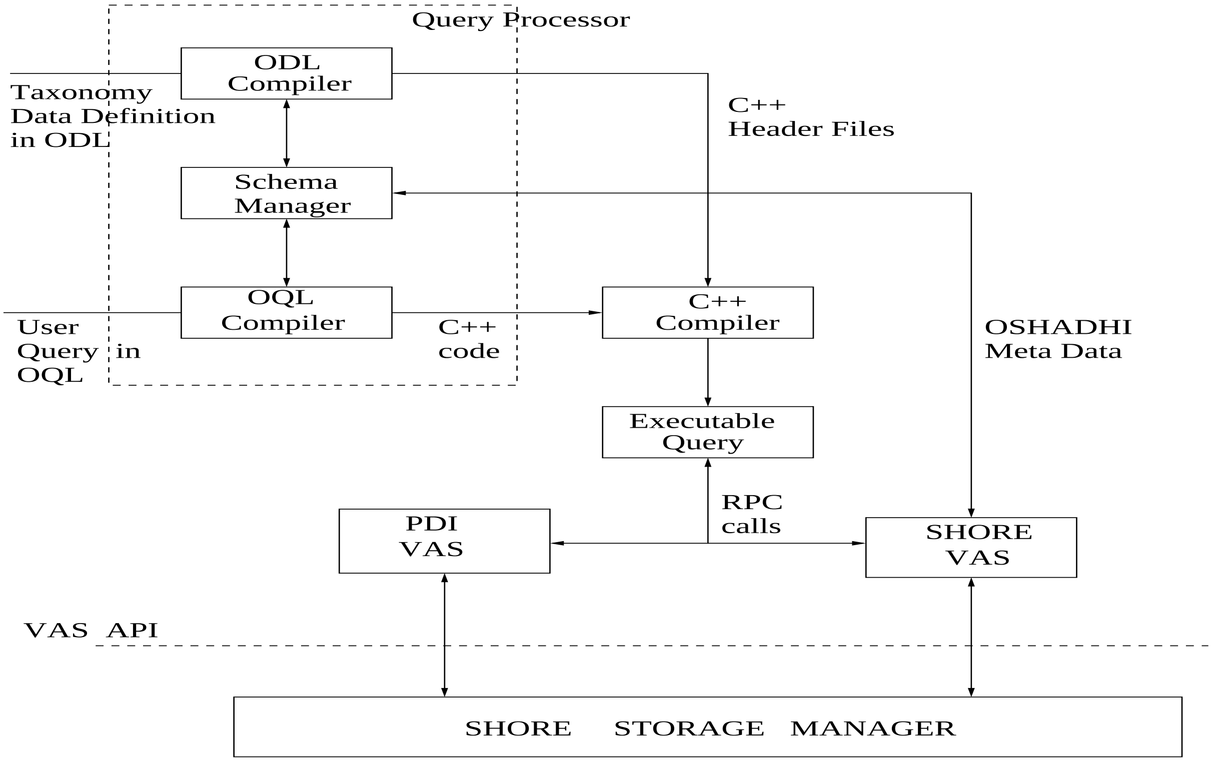


Figure 6.1: The high level diagram of data flow in query processing.

modularly constructed to make it relatively simple for users to build application-specific servers. Path Dictionary Index (PDI) is implemented using this VAS programming interface (see figure 6.1). More details on VAS and SHORE can be found in [C+94].

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

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

6.3 MT-index

SHORE supports a multi-dimensional index data structure called R^*-tree [B+90]. As discussed in section 3.3, any multi-dimensional data structure can be used along with linearization algorithm. To simplify the coding effort, we decided to use already available R^*-tree as multi-dimensional data structure in the implementation of MT-index.

Chapter 7

Current Status And Future Work

The taxonomy data model is written in ODL. Path dictionary index is implemented and extended to support N:M relationships. RPC communication is also implemented between the executable query and the PDI server. As part of the MT-index implementation, the code for linearization algorithm is written and still to be integrated with the SHORE built-in R^*-tree. The implementation, which is on a SPARC/Solaris-2.5 system, consists of about 12,000 lines of C++ code. Some preliminary testing is done on PDI implementation, which also indicated that the implementation with discussed extensions performs reasonably well. But, extensive testing and performance evaluation is still needs to be done.

In future, different applications such as OODB mining can be implemented to find the interesting relations between various species.

Chapter 8

Conclusions

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

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

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

[LL98] Wang-Chien Lee and Dik Lun Lee. Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):371–388, May 1998.

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

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

[MP97] Thomas A. Mück and Martin L. Polaschek. A Configurable Type Hierarchy Index for OODB. *VLDB Journal*, 6(4):312–332, 1997.

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

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

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

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

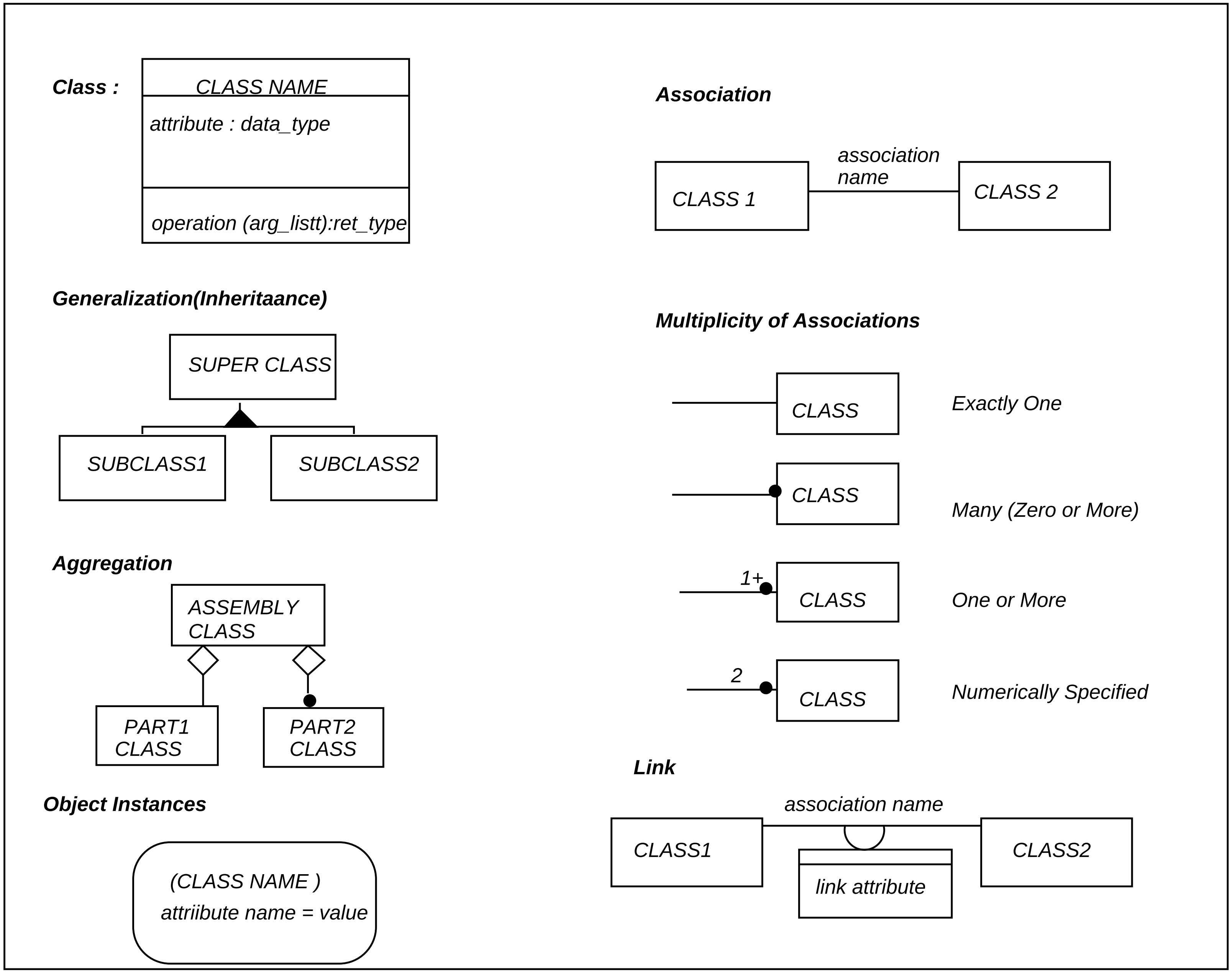
[Val87] Patrick Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[Vas74] P.C Vasishta. *Taxonomy of Angiosperms*. S. Chand and Co., Delhi, 1974.

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

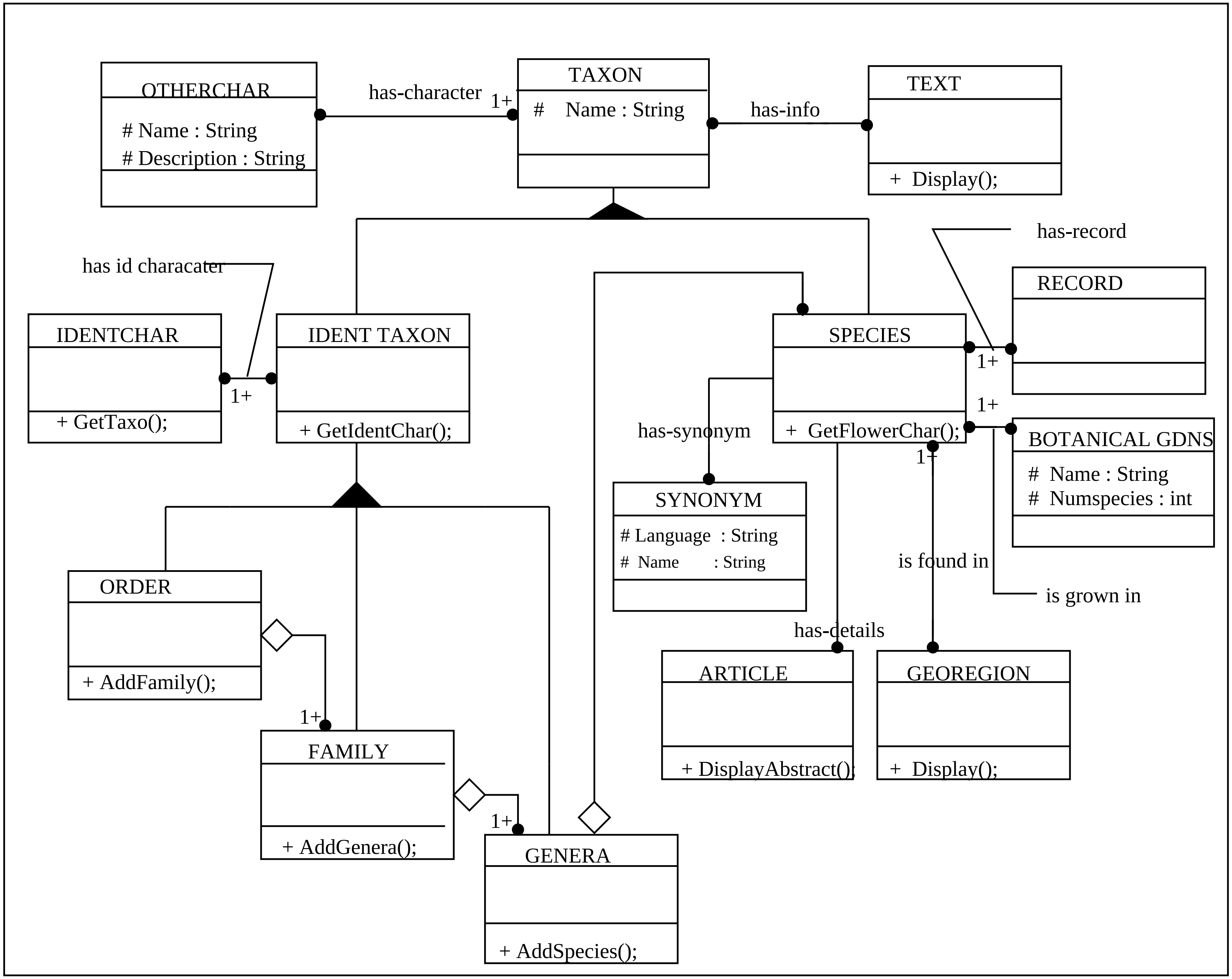
Appendix A

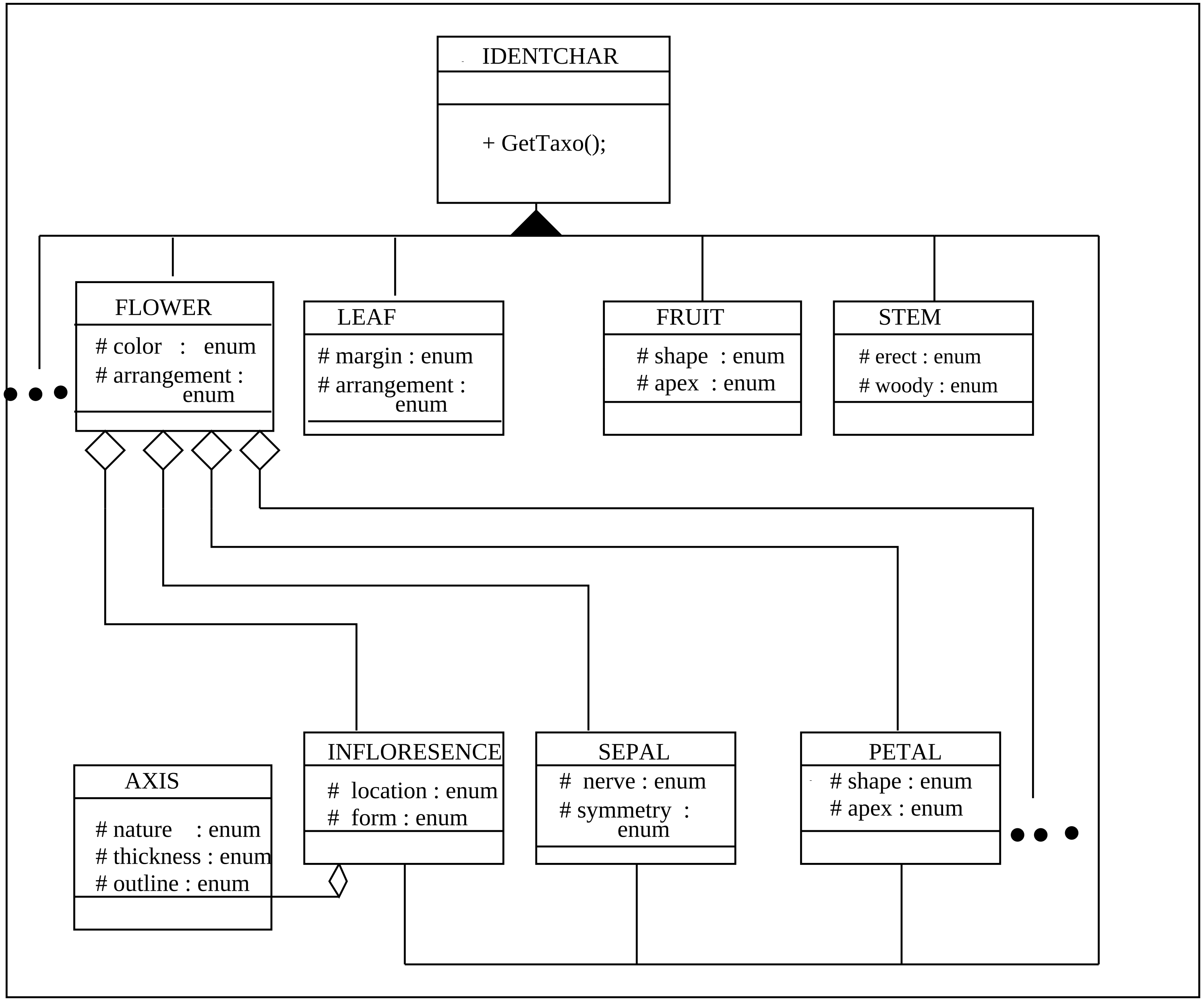
The Rumbaugh Notation

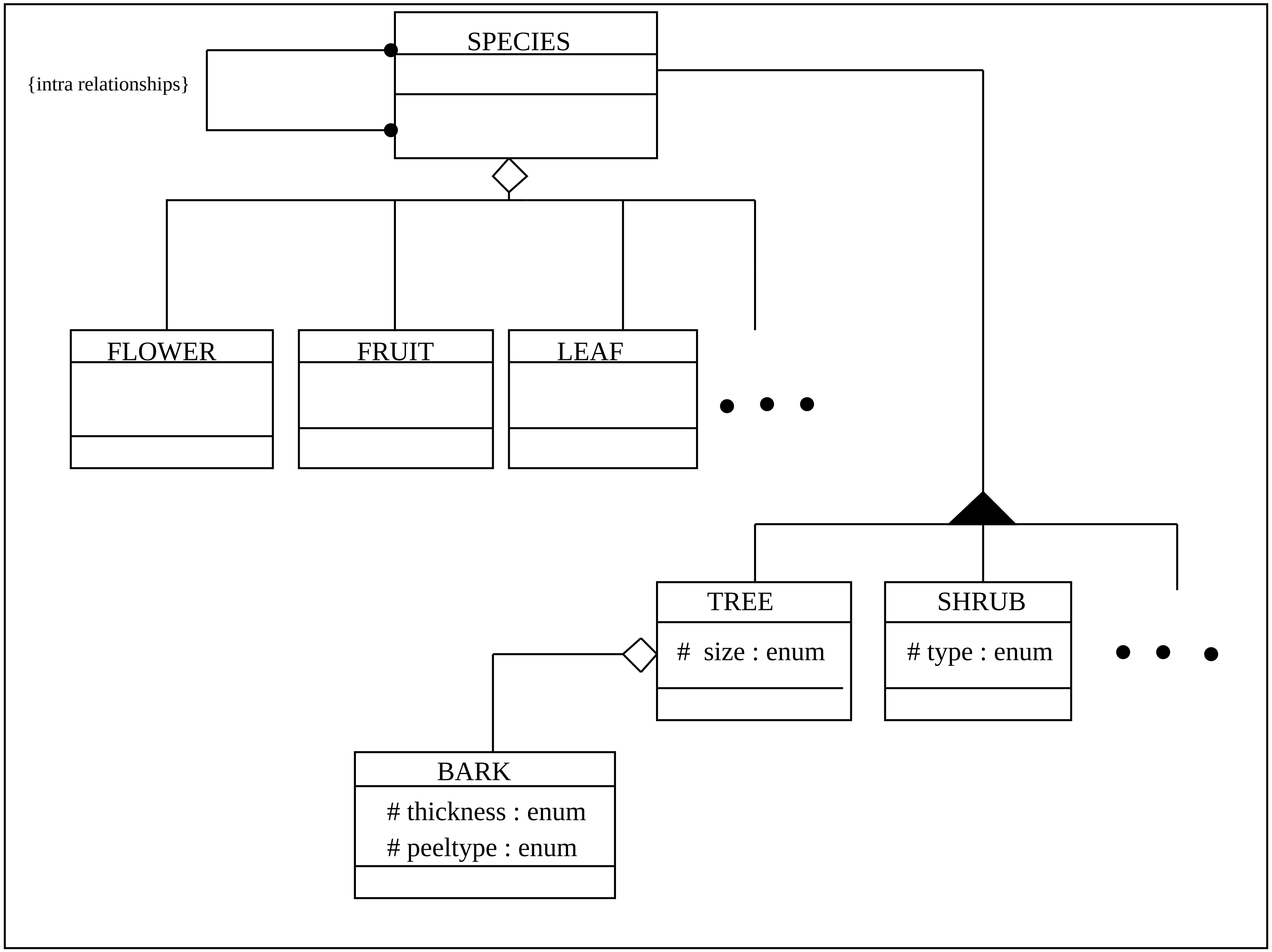

Class :
CLASS NAME
attribute : data_type
operation (arg_listt):ret_type
Association
CLASS 1
association
name
CLASS 2
Generalization(Inheritaance)
SUPER CLASS
SUBCLASS1
SUBCLASS2
Multiplicity of Associations
CLASS
Exactly One
CLASS
Many (Zero or More)
1+
CLASS
One or More
2
CLASS
Numerically Specified
Aggregation
ASSEMBLY
CLASS
PART1
CLASS
PART2
CLASS
Link
association name
CLASS1
link attribute
CLASS2
Object Instances
(CLASS NAME)
attriibute name = value

Appendix B

Taxonomy Data Model


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


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


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

Appendix C

Taxonomy Data Model in ODL

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

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

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

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

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

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

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

    const  Genera getGenera() ;

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

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

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

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


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


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

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

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

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

SeedChar setSeedChar( in SeedChar irefSeedChar );
const   SeedChar getSeedChar( ) ;
```

```
    void deleteSeedChar( in SeedChar irefSeedChar );

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

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

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

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

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

    relationship Order  refParentOrder
                        inverse Order::stCollFamily;
```

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

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

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

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


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

```
  public:

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

class SepalChar : IdentChar{
  protected:
    relationship FlowerChar refFlower
                            inverse FlowerChar::refSepal;

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

// Other identifying characteristics are not listed here
```

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

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

class Word{
  protected:
    attribute string strWord;

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

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

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

```
  };

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

 public:
    const   Order searchOrder( in string istrOrderName ) ;
    const   Family searchFamily( in string istrFamilyName ) ;
    const   Genera searchGenera( in string istrGeneraName ) ;
    const   Species searchSpecies( in string istrSpeciesName ) ;
    Order addOrder( in string istrOrderName );
    Family addFamily( in string istrOrderName,
                      in string istrFamilyName);
    Genera addGenera( in string istrFamilyName,
                      in string istrGeneraName );
    Species addSpecies( in string istrGeneraName,
                        in string istrSpeciesName );
    void deleteOrder( in string istrOrderName );
    void deleteFamily( in string istrFamilyName );
    void deleteGenera( in string istrGeneraName );
    void deleteSpecies( in string istrSpeciesName );
    const   Order getOrderElement( in long iiElementIndex) ;
    void initialize( in TaxonIndex irefTaxonIndex);
  };
```