Query Processing and Optimization in Oshadhi

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

by

Satheesh Konidala

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

Jan 2000

Dedicated to my Secondary $HOME

"Database Systems Laboratory"

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my project guides, *Dr. Jayant Haritsa* and *Dr.Ramesh Hariharan* , for their excellent guidance throughout the course of this project.

I am grateful to *Dr. Jayant Haritsa* for his constant support and encouragement, without which this project would not have materialized. I take this opportunity to express my deepest gratitude to him for the moral support that he gave me and for providing me a vast pool of literature and computing resources.

I take this opportunity to express my deepest gratitude to *Dr. Ramesh Hariharan*, a wonderful guide to work with, who has constantly helped me in the project.

I would like to thank my advisor *Dr. R. C. Hansdah*, who helped me in choosing proper courses, for his guidance during my M.E course.

I am very much indebted to my Oshadhi Colleagues *Srikanta, Rajarao, Madhav and Amitabh* who helped me throughout the project, by giving me their valuable suggestions and helped me in getting an insight of the project by their constructive criticism in the discussions which brought me in right track when ever I'm heading happily(?) the other way, from time to time.

I thank *Vikram and Maya* who made *DSL* a pleasant place to work and for sacrificing their time and terminals whenever we are in need of terminals.

Abstract

Oshadhi is an object-oriented biological information system being developed at Indian Institute of Science. In this project we concentrated on the query processing requirements for typical biological applications, like Oshadhi, and some query optimizations that could be done.

The data definition language and the query language was chosen to be such that any general purpose application can be written on top of Oshadhi. The optimizations that we focus on, in this project, are making use of the presence of indices on aggregation hierarchy and spatial data and extending the use of indices to the parameterless member functions. Since different applications may need different ways of visualizing the results, the query processor is designed such that query processor is independent of the format of displaying the results.

Chapter 1

Introduction

Oshadhi is a biological information system being developed at Indian Institute of Science, India. This project aims at building the query processor and optimizer for Oshadhi. This chapter discusses about the motivation of the project and the issues in the query processing/optimization and then gives the organization of the whole thesis.

1.1 Motivation

The genetic base of India is represented by a large number of plant species of varying characteristics. The rich economic importance and the medicinal use of these plant species make it essential for their bio-diversity to be conserved. Bio-diversity data includes taxonomic data (like details of all the species, genera to which this species belongs, their classification and characteristics) ,spatial data (like details of the geographic locations of the places where they live, their environmental conditions) and genomic data (like protein sequences of the species). Due to the presence of hierarchical relationships in biological data, an object-oriented approach was chosen to represent the Oshadhi data model and an OODBMS was chosen to implement this data model[Vid95].

Since, even within a single domain such as biological applications, the query requirements vary significantly from application to application, a general purpose query language is chosen. Since Oshadhi is a general purpose OODBMS tuned to biological applications, various indexing mechanisms like path dictionary for aggregation hierarchy, MT-index for class hierarchy and Hilbert R-tree for spatial data are supported(see [Che00, Tad00] for more details). So, optimizer has to make use of these indices in the process of query optimization. For the generic object-oriented query optimizations(which are application independent) like query unnesting, reordering of the order of computing joins etc., Oshadhi uses Lambda-DB [Feg97], a rule-based optimizer. Oshadhi's optimizer is built by extending Lambda-DB's optimizer to support the above mentioned indices provided in Oshadhi. Oshadhi provides results in a generic form consisting of both data and meta-data so that any front-end can interpret the results and display them according to its requirements.

1.2 Query Processing And Optimization: Issues

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

1.2.1 OODBMS Query Optimization Issues

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

1.2.1.1 Path Expression

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

1.2.1.2 Derived Attributes

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

1.2.2 Spatial Query Optimization Issues

The characteristics of spatial data is that data can be present in a multi-dimensional space and needs special way to represent them and special operators like overlap, contain etc. to operate on them [Güt94]. Some of the issues in spatial query processing/optimization are

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

- Conventional join algorithms may not do better in the presence of spatial data because the join predicate will be an operation like overlap, contain etc. [Güt94].

There are several spatial join algorithms which do join efficiently on a particular spatial access structure. So, the query processor has to make use of the presence of these spatial join algorithms.

1.3 Organization

The rest of this thesis is organized as follows: Oshadhi's system architecture is briefly explained in Chapter 2. Chapter 3 discusses the SHORE on which Oshadhi is built. Chapter 4 discusses the Data Definition Language and query language of Oshadhi. Chapter 5 introduces Lambda-DB on which optimizer of Oshadhi is built. Chapter 6 discusses about query processing and Chapter 7 discusses the optimizations done in Oshadhi. Chapter 8 discusses about the future work and Chapter 9 presents the conclusions.

Chapter 2

System Architecture of Oshadhi

Oshadhi is built on top of SHORE (Scalable Heterogeneous Object REpository) [C$^+$94], a database micro kernel which provides the facilities like transactions, recovery, persistence and a DDL (Data Definition Language) called SDL (SHORE Definition Language). SHORE in described in detail in Chapter 3.

Oshadhi is mainly divided into 5 components. The System Architecture of Oshadhi is as shown in Figure 2.1. Each of the Spatial, Taxonomy and Genome components have special mechanisms to handle specific data types.

1. Spatial component provides the query processor with
    - Spatial data types like point, polygon etc.
    - Spatial indexing mechanisms like Hilbert-Rtree etc.
2. Taxonomy component provides the query processor with
    - Path dictionary index used to index the aggregation hierarchy.
    - MT-index to index the class hierarchy.
3. Genome component provides
    - Data types like protein sequences, DNA sequences etc.

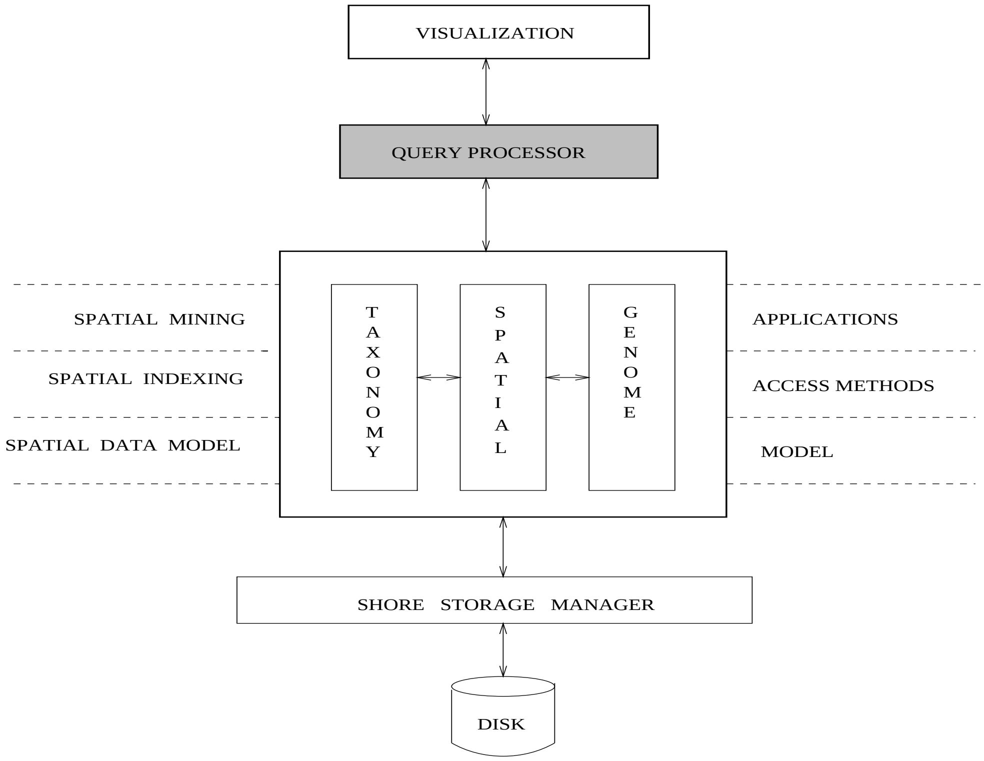

Figure 2.1: **Oshadhi System Architecture**

- Algorithms like BLAST, FASTA for finding the similarity measures between any two sequences.

4. Visualization is a web-enabled front-end with a GUI-server and a GUI-client and the GUI-server interacts with the query processor. There will be a protocol for information exchange between the GUI-server and the GUI-client and another protocol of information exchange between GUI-server and query processor. So, GUI-server converts the results from the query processor to the format necessary by the GUI-client.

Query processor gets a query from the Visualization component and uses the services of Taxonomy, Genome, Spatial components and SHORE to compute the queries. Query processor supports a general purpose DDL (Data Definition Language) and query language. DDL definitions are converted into SDL definitions and OQL queries are converted into C++ code. Oshadhi's optimizer is build on top of Lambda-DB (described in more detail in Chapter 5).

Chapter 3

SHORE

The Shore (Scalable Heterogeneous Object REpository) is a new persistent object system that represents a merger of object-oriented database (OODB) and file system technologies. In this chapter, we describe features of the SHORE (or Shore) system, which forms the backend database server of Oshadhi for storing all the objects.

Most of the material in this Chapter is taken from [C$^+$94] and SHORE Manual Pages.

3.1 Introduction

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

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

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

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

3.2 Basic Shore Concepts

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

3.3 Shore Object Basics

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

The *core* of an object is described by its type. To support the flexibility of dynamic structures with the efficiency of logically contiguous blocks on secondary storage, Shore allows each object to be extended with a variable-sized *heap*(Figure 3.1). The heap is used by the system to store variable-sized components of its value such as strings, variable arrays and sets.

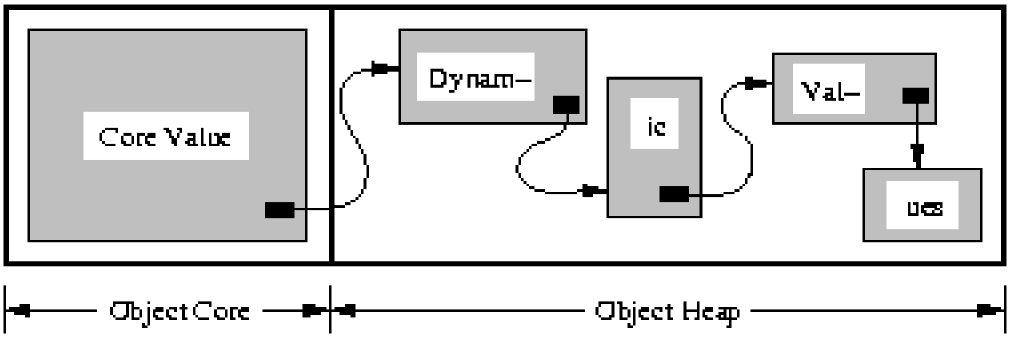

Figure 3.1: **Shore Object Structure**

3.4 File System Features

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

3.4.1 Shore Object namespace

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

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

3.4.2 Legacy Unix tool support

While Shore provides a richer environment than traditional file systems, there are many situations where tools designed to be used on files need to be invoked on database objects. The solution followed in Shore is to provide a special *Unix compatibility* feature. The Unix compatibility feature may be utilized by designating a range of bytes as its *text* field. A compatibility library provides versions of Unix file system calls.

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

3.5 Object Oriented Database System features

3.5.1 The Shore Type System

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

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

3.5.2 Shore Language Bindings

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

3.5.3 Other OODB-Like services

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

3.6 The Shore Architecture

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

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

3.7 Shore Software Components

The main software components of Shore (Figure 3.2) constitute the Shore server and the Language Independent Library. We next discuss these components in greater detail.

3.7.1 The Language Independent Library

The process of converting object references on disk to main memory addresses is called *swizzling*. When an application attempts to dereference an "unswizzled" pointer, the

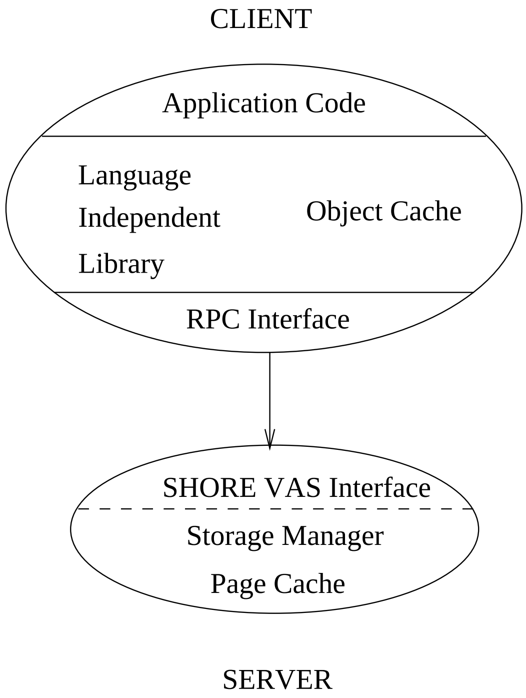

Figure 3.2: **Application - Server Interface**

language binding generates a call to the object-cache manager in the *language independent library* (LIL). If the desired object is not present, the LIL sends an RPC request to the local server, which fetches the necessary pages by reading from the local disk.

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

3.7.2 The Shore server

The Shore server (Figure 3.3) is divided into two main components: a *Server Interface*, which communicates with applications, and the *Storage Manager* (SM), which manages the persistent object store.

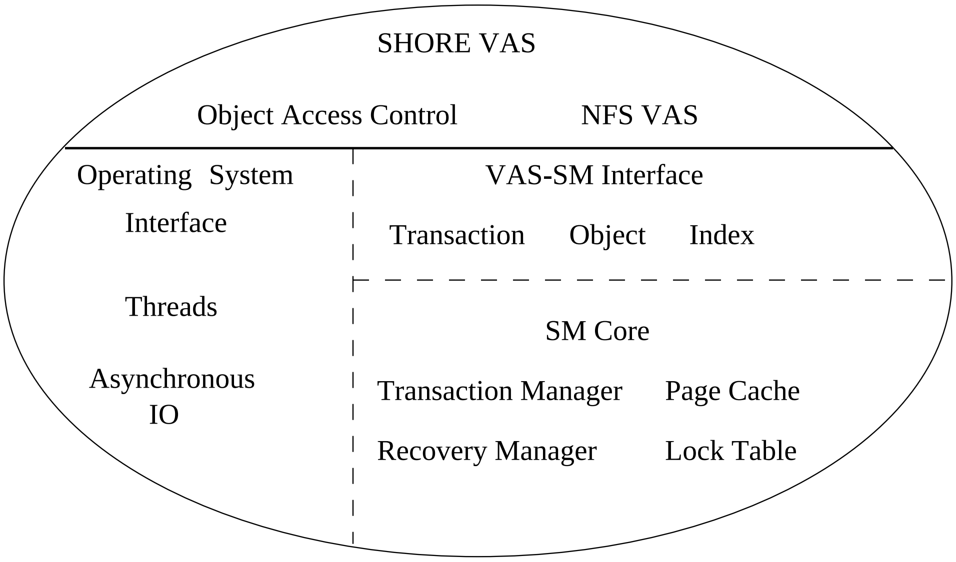

Figure 3.3: **System Architecture**

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

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

3.8 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 4

Language For Oshadhi

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

4.1 Data Definition Language

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

- In ODL the namespace is a set of modules and each module consists of a set of classes and there is no way in which one can refer one module from another. We have extended this concept to allow a module to be able to use any other module as it greatly will enhance the reuse of code.

- In ODL there is a concept of nested modules wherein the module definitions can be nested. Since, ODL declarations are converted into SDL declarations (see figure 6.1 for the flow of an ODL declaration) and as SDL does not support nested modules, this feature has been removed from ODL.

- ODL is extended to support the indexing to be done on derived attributes (more details in 7.2.1.1).

The following example (shown in Table 4.1) is written in DDL of Oshadhi. Each of the class definition has an *extent* declaration which specifies the table where all the objects of that class are to be stored. The declaration *key name* states that attribute *name* is the primary key. The *import* statement states that all the classes defined in the imported module are accessible in this module. The *SpatialModule* is a module, containing the classes like point, polygon and polyline etc, whose definition is not shown here. Each attribute is declared by the keyword attribute followed by type and then the attribute name. The rest of the example is self-explanatory. The grammar for the DDL of Oshadhi is given in the Appendix A.

4.2 Query Language for Oshadhi

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

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

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

1. **Object Creation:**

   To create a persistent object, say of type City:

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

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

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

Table 4.1: **Example code illustrating DDL in Oshadhi**

City object. All the indexes which are present on attributes of the class will be automatically updated by the query processor.

2. **Simple query:**

   To print the details of a plant say “neem” :

   FOR EACH i IN (

   *SELECT **

   FROM p IN plants

   WHERE p→name = "neem")

   DO i→Print();

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

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

   FOR EACH i IN (

   SELECT c

   FROM p IN plants ,

   c in p.specs

   WHERE p.name = "neem"

)

   DO i→Print();.

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

4. **Query with a Join:**

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

FOR EACH i IN (

*SELECT **

FROM c IN cities, w IN waterbodies

WHERE c.name = “Bangalore”

AND c.location.overlap(w.location)

AND w.type = LAKE

)

DO cout<< i →w→name<<endl;

5. **Index Creation:**

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

   CREATE HILBERT INDEX hilID1 ON Forest(boundary);

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

6. **Object updation:**

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

   (a) **Simple attribute assignment:**

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

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

   bombay.name := "Mumbai";

(b) **Collection Attribute Insertion:**

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

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

thulasi.medicineFor += "Heart Attack";.

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

(c) **Collection Attribute Deletion:**

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

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

thulasi.medicineFor −= "Heart Attack";.

Chapter 5

Query Processing and Optimizations in Lambda-DB

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

Most of the material in this chapter is taken from [FM95], [Feg95], [Feg97], [Feg98a] , [Feg98b] and Source Code Documentation.

5.1 Introduction

Lambda-DB works with OQL and ODL as its query language and DDL respectively. A query passes through several phases before the equivalent C++ code is generated for the query. This code after execution produces the results. Each of these phases are described in more detail in later parts of this chapter. An ODL declaration is converted into a SDL declaration. As ODL and SDL are more or less similar, the conversion is straight forward and the details are not discussed. Lambda-DB provides OPTL language [Feg97] using which all the rules and actions, for the query processing and optimization, are written. OPTL is described briefly in Section 5.4.

Lambda-DB uses both a calculus and an algebra as intermediate forms because calculus closely resembles OQL and is easy to optimize by query-rewriting, while the algebra is low level and can be directly translated into the execution algorithms supported by database [Feg97].

The next section discusses about the calculus of Lambda-DB, the monoid comprehension calculus, which is used as the intermediate form of query representation. The monoid algebra, similar to the nested-relational algebra, serves as an intermediate form between comprehensions and physical plans and the algebraic operators are discussed in Section 5.3.

5.2 The Monoid Comprehension Calculus

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

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

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

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

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

5.3 The Monoid Algebra

This section discusses about the algebra used in Lambda-DB. The following are the algebraic operators used by lambda-DB and the conversion process from the calculus to these algebraic operators is discussed in the Section 5.8. An example demonstrating these algebraic operators is given in this Section.

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

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

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

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

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

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

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

Table 5.1: **An Example in OQL to Demonstrate Monoid Algebra**

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

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

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

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

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

Let us consider an example to demonstrate how an OQL query in the algebraic format looks like. The query is, say, to display all the characteristics of neem plant which satisfies the predicate some_predicate. Then the corresponding OQL query is shown in the Table 5.1 and the corresponding algebraic expression is shown in Table 5.2. x and y are temporary variables generated by query processor and are used as temporary data holders.

Each of the algebraic operations present in Table 5.2 yields values of the following types:

get → set(< e: Plant >)

unnest → bag(< e: Plant, c: Characteristic >)

nest → bag(< e: Plant, x: bag(string) >)

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

Table 5.2: **An Example Query in the form of Monoid Algebra**

reduce $\rightarrow$ set(< MyPlant: string, Spec: bag(string) >)

Notice that

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

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

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

5.4 OPTL

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

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

Term:

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

- string denoting the data or

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


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

Table 5.3: **Examples of OPTL Expressions**

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

See the Table 5.3 for an example demonstrating the OPTL expressions.

Case Statement:

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

As an example of case statement in OPTL, consider the function shown in Table 5.4.

Here, if it exp is any expression having a function with one of the arguments being a join expression, then it swaps the arguments of that join expression iff the arguments are not same. Here f is bound to the function name, r and s is bound to the list of arguments to the left and right of the argument having the join expression. Options expressed can have gaurds, i.e., after pattern matching, the action is executed only if the gaurd is satisfied. Gaurd can be any C++ predicate. In the Example shown in Table 5.4, the predicate $x \rightarrow eq(y)$ is the guard and this does the job of removing the join expression if there is a join of same tables. (Note that the order of the statements are important. There can be expressions which can match so many template options in the case statements, then in that case only the first one's action is executed.)

Expr switch_join_args(**Expr** exp) { **#case** exp $\|`f(\ldots r, join(`x, `y), \ldots s) : x \rightarrow eq(y) \Rightarrow$ **return** $\# < `f(\ldots r, `y, , \ldots s) >;$ $\|`f(\ldots r, join(`x, `y), \ldots s) \Rightarrow$ **return** $\# < `f(\ldots r, join(`y, `x), \ldots s) >;$ $\|`x \Rightarrow$ **return** x; **#end**; };

Table 5.4: **Example of OPTL Case Statement**

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

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

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

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

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

For more details on OPTL language one can refer [Feg97].

5.5 Phases of Lambda-DB Optimizer

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

1. **Parsing and type checking:** During this phase, query in OQL is translated into the comprehension calculus and it is checked for type errors.

2. **Normalization:** During this phase, some forms of nested queries are unnested using the *normalization* algorithm described [Feg97]. During *normalization* , the generator domains are reduced into simple paths.

3. **Translation into an algebraic form and query unnesting:** After comprehensions are normalized they are translated into nested-relational operators which are extended to include multiple collection types and aggregation.(These operators are described in detail in Section 5.3). At the same time, nested comprehensions are unnested, by converting joins into outer-joins and by grouping together the results.

4. **Materialization of path expressions:** During this phase, if a query contains a path then it is materialized.

5. **Operator Ordering:** During this phase, the best join processing order is chosen.

6. **Physical plan generation:** The last phase considers the available access paths (indices) of each extent and the available alternative physical algorithms to generate different plans. Finally, the best plan is selected and code is generated for that.

The following sections discusses each of these phases in detail.

5.6 Parsing And TypeChecking

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

SELECT e

FROM x1 IN e1 , ... ,xn IN en

WHERE pred;

and is translated into

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

in the monoid comprehensions form.

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

SELECT p.name

FROM p1 IN (SELECT f.plants

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

This query is expressed in comprehension calculus as follows:

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

5.7 Normalization

Another claim for monoid calculus is that it supports easy manipulation of query expressions which is amenable to pattern-based rewriting [FM95]. This monoid calculus expressions can be put into a canonical form by an efficient rewrite algorithm called the *normalization* algorithm. (The *normalization* algorithm is given in [FM95], the algorithm in the form of OPTL rules is given in [Feg97] and its proof of correctness is found in [Feg94]). It generalizes many optimization techniques, such as pushing a selection before a join. The evaluation of these generally produces fewer intermediate data structures than the initial unnormalized programs.

For the query given in Section 5.6, that finds all plants in the forest Mudhumalai which can be useful in treating Asthma, the OQL query and the calculus form is shown in the Section 5.6. The normalized form of that query is:

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

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

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

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

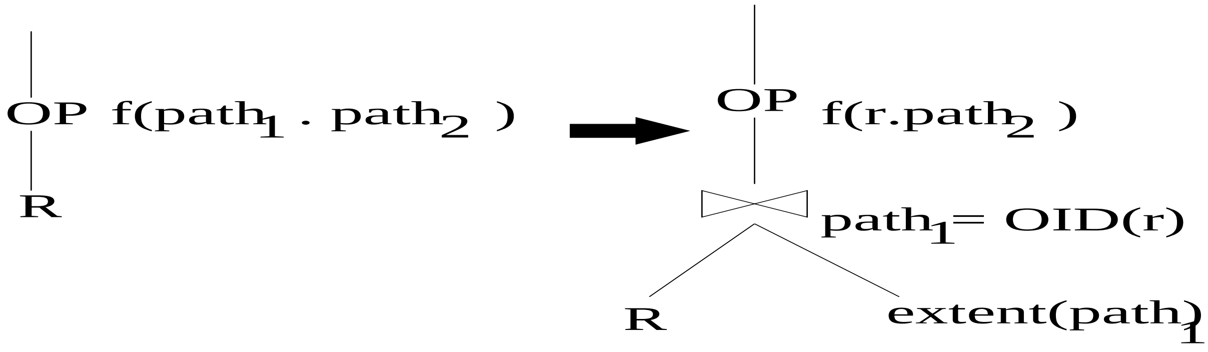

Figure 5.1: **Path Materialization**

5.8 Translation into an algebraic form and Unnesting

The next stage after normalizing is the translation of the comprehensions into the algebraic operators. The generators of a comprehension are translated from left to right, generating either a join (if the domain is an extent) or an unnest (if the domain is a path). Refer to Section 5.3 for an example demonstrating the conversion of an OQL query into algebra. If the query has any nested queries then they are unnested in this phase. (For more details on unnesting the query one can refer to [Feg98b]).

5.9 Materialization of Path Expressions

One of the most common technique for OODBMS is converting path expressions into joins [Feg97, BKG93]. It is also known as materialization of path expressions [CD92].

The Figure 5.1 shows how a path expression is converted into a join. The Figure 5.1 indicates that if op is an unary operator (such as unnest) that depends on path expression $path_1.path_2$, where $path_1$ and $path_2$ are paths and R is any input stream to the operator op, f is any predicate then this can be converted into a join of the first extent with that of the input stream R. The resultant is shown in the right part of the Figure 5.1 The detailed algorithms and rules in OPTL for this transformation is given in [Feg97].

5.10 Operator Ordering

This phase re-orders the order of the join processing so that the cost of join is minimum. Because of a number of OODB optimizations implemented in Lambda-DB, such as translation of path expressions into joins and query unnesting, it may generate a large number of implicit joins even for simple queries. So an efficient join order processing algorithm has gained importance in this context [Feg97]. The join processing ordering algorithm used in Lambda-DB is a polynomial time (of order $\mathcal{O}(n^3)$ where n is the number of relations) greedy algorithm called as GOO(Greedy Operator Ordering).

5.10.1 Greedy Operator Ordering Algorithm

The GOO algorithm is a bottom-up algorithm, similar to Kruskal's minimum spanning tree algorithm. The main data structure is the query graph where each relation r_i corresponds to a node i and each join predicate P_{ij} corresponds to an edge between i and j. Each node i is labelled by the size of r_i and each edge between i and j is labelled by the selectivity S_{ij} (the selectivity of predicate P_{ij}). At each step of the algorithm, it selects two nodes i and j that have minimum value of size $(r_i)\times$size$(r_j)\times S_{ij}$. Then these two nodes are merged into a new node ij. The size of node ij is size$(r_i)\times$size$(r_j)\times S_{ij}$ and its operator tree is $r_i \bowtie_{P_{ij}} r_j$. In addition for each node k connected to both i and j, the weight of the edge between the nodes k and ij becomes $S_{ik} \times S_{jk}$. For a full description of the algorithm see [Feg98a]. The GOO algorithm is not optimal always as it is a greedy algorithm based on heuristics. Refer to [Feg97] for an example for which GOO is sub-optimal.

5.10.2 Ordering OODB Operations

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

5.11 Physical Plan Generation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   Sorts a *plan* by the *sort_order*.

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

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

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

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

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

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

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

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

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

Chapter 6

Query Processing in Oshadhi

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

6.1 Data Flow in Oshadhi

Figure 6.1 shows different components of query processor and the flow of a query, in Oshadhi, starting from the user to the displaying of results.

ODL Declaration:

Since all objects are to be stored in SHORE, we have to use SDL as the DDL at some level. An ODL declaration is therefore parsed and converted into an equivalent SDL declaration and the SDL compiler generates an equivalent C++ binding for the SDL declarations. While Converting ODL Declarations to SDL these declarations are stored by the Schema manager for further use in optimization. ODL compiler also generates some C++ classes, in addition to SDL declarations. ODL compiler generates one class for each class and module. These are called as ClassController classes and ModuleController classes. The main use of these Controller classes is given in Section 6.3. But the use of this Controller classes can be extended to a greater extent. For example to initialize all the attributes of the objects with default values , to handle the exceptions raised by

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

Table 6.1: **Meta-data Storage Format**

the C++ member functions etc. This Controller classes design is actually a restrictive case of MVC(Model-View Controller), one of the mostly used design pattern in software architecture, where an object will be controlling a set of objects (for more details on what a design pattern is and on MVC pattern see [GHJV95]).

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

6.2 Schema Manager

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

6.2.1 Storage Format

The storage structure of meta-data is as shown in the Table 6.1. The *Name* field stores the name of the declaration (like attribute name, function name etc.), *Type* is the type of the declaration (like index-dcl, attrib-dcl, method-dcl, extent-dcl etc.), *ClassName* is the name of the class to which this declaration belongs , *Statistics* stores the statistics (like the size, cardinality etc) and *DeclarationExpression* contains the expression of the declaration in a string form as it may contain more properties which varies among the declarations (like method can have a list of parameter types and path dictionary have a path expression to be stored).

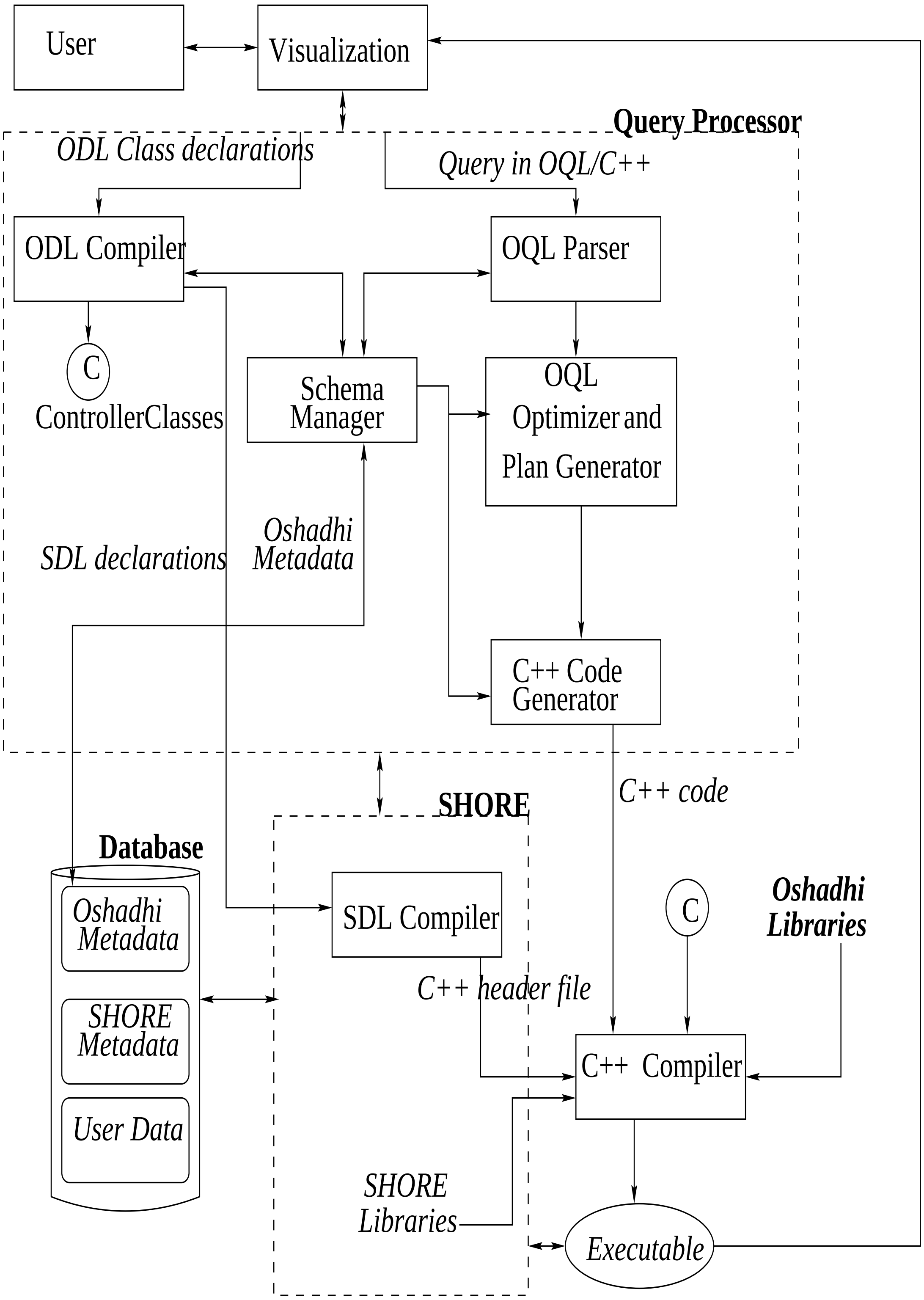

Figure 6.1: **Components of Query Processor and Query Flow in Oshadhi**

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

Table 6.2: **Meta-data Export Format Example**

6.2.2 Meta-data Export Format

The optimizer and the Visualization module needs to access the meta-data for their processing. Optimizer may need, say, information about the presence of indices to use in it's optimization. Similarly Visualization needs the type information in forming the query. So, Schema Manager exports the meta-data in the format whose grammar, in BNF, is shown in Appendix C. An example showing this export format for the class Plant shown in Table 4.1 is shown in the Table 6.2

6.3 Interface With Visualization

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

6.3.1 Issues in Interface with Visualization

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

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

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

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

1. An issue is how does the query processor traverses recursively. The main problem is how to get the attribute value at runtime given an object and attribute name. Some possible solutions to this are
   (a) While generating code for the query, generate code for functions which returns attributes given an object. But this has some disadvantages:
      i. So much of effort to generate the code from the query and this will increase the code generation time.
      ii. This code bloating will increase the compile time also.
   (b) Generate some standard functions, for each class, which returns the attributes given an object of that class type.
2. One more issue is how to pass the data from the query processor to the Visualization. There are two possible ways of doing this:
   (a) Materialize all the results in a particular format in a file and pass them to the GUI-server. GUI-server will understand the format and convert them into the format needed by the GUI-client for displaying them accordingly. But one disadvantage is that data has to be materialized in a file and it takes more disk accesses.
   (b) Another solution is pass the attributes to the GUI-server as and by when traversing the object recursively.

3. One more issue arises which is how to display the meta-data and data and in which format?. A question arises whether there is a unique way of displaying the meta-data for all applications?. The answer is unfortunately no. Each application may need a different type of displaying the data and the meta-data. So, the query processor has to be independent of the displaying format. But if results are materialized while passing the results from query processor to the GUI-server then this problem does not arise because there can be a protocol between the GUI-server and query processor and GUI-server can then convert this into the format necessary to the GUI-client and the flexibility of having more than one displaying format is possible by adding one more converter in the GUI-server. If results were not materialized but were pipelined from the query processor to the GUI-server then one possible solution is to define an interface using which the object's values are passed to the GUI-server and GUI-server sends the data and meta-data according to the needs of the GUI-client. Extensibility to have many output formats can be obtained by having different implementations of this interface.

6.3.2 Implementation Details

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

- For the problem of traversing the objects recursively we have a function generated in the ClassController class, which given an attribute name and the object returns the attribute value. The sample code of such a function is shown in Table 6.3

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

  Table 6.4 shows the ModuleController class generated for the example shown in Table 4.1

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

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

    }
};
```

Table 6.3: **ClassController Class having functions to help in recursive traversal of objects**

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

    };
};
```

Table 6.4: **ModuleController Class having functions for returning ClassController objects**

```
//file query.oql

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

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

Table 6.5: **An example demonstrating the extensibility of the formatters of results**

- For the problem of meta-data displaying Oshadhi has defined an interface, called as Formatter class, to which data will be passed as and by when attributes are traversed and GUI-server implements the interface to produce the results in the way the GUI-client needs. This leads to the independence of the query processor to the format of displaying the results. Any support for new format has be done by the GUI-server. For example, if GUI-server decides to send all its output as Latex file then a class is defined which implements the Formatter class and this object is passed to the query processor. Table 6.5 demonstrates how a typical query will be to use a new formatter in displaying the results.

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

The interactions between the query processor and the formatters is shown in the Figure 6.2. It can be observed that the architecture of interactions, for formatting the results, between the query processor and the Visualization is nothing but a Builder pattern [GHJV95]. There will be one ModuleController class for each module and whenever query processor changes to a module its moduleControllerObject is set to the current module's ModuleController object. There are several formatterObjects possible, one for each type

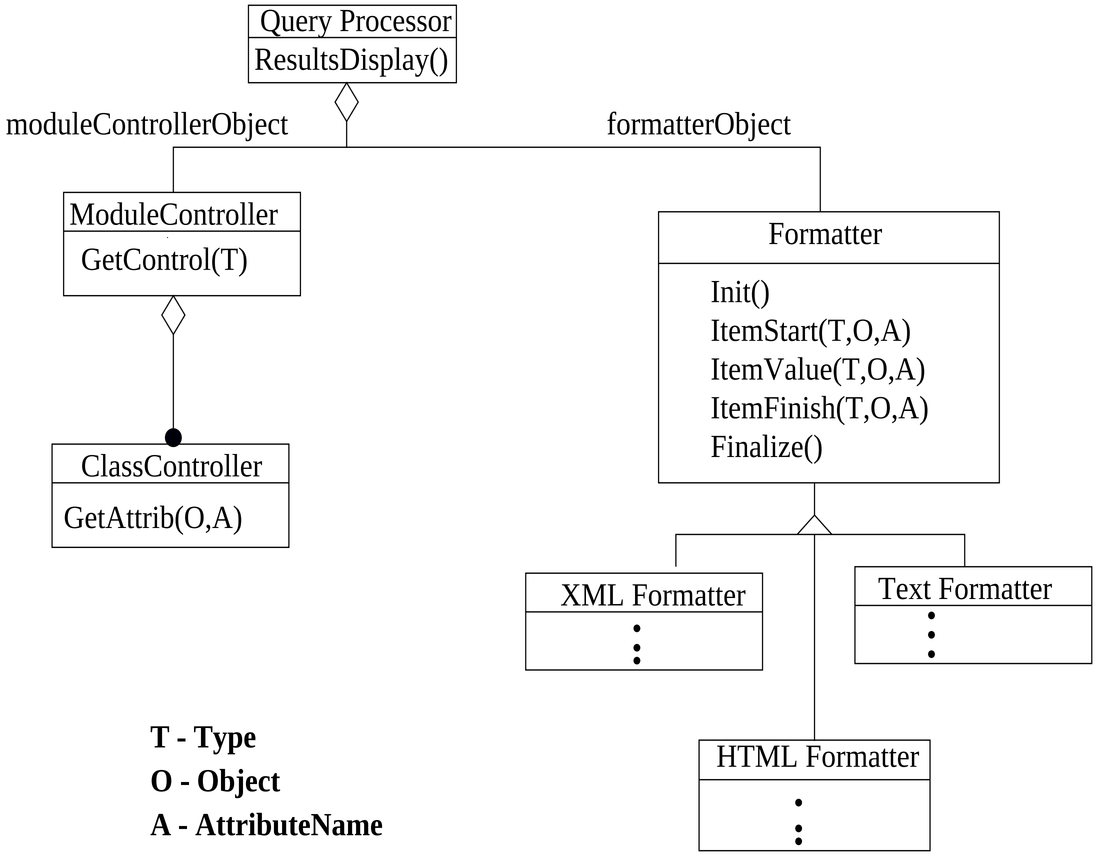

Figure 6.2: **Query Processor and Result Formatters**

of the formatting. Using the SetFormatterObject supplied by the query processor, as shown in Table 6.5, one can pass the object to the query processor which is to be used for formatting.

A high level algorithmic description of the function OshadhiDisplay, which traverses recursively and displays all the attributes, is shown in Table 6.6. This also explains how the formatters interacts with the query processor in displaying the results. Actually the code is written in OPTL. The functions ItemStart, ItemValue and ItemFinish, in Formatter class, displays the delimiters of the object and the object itself. The functions Init and Finish displays the delimiters of the whole query if needed.

To demonstrate the principle an XMLFormatter is implemented. The tags used in this implementation are:

$< STRING >$, $< LONG >$ and $< REAL >$ to represent the basic types

and $< ERROR >$ to display an error message

and $< OSHADHIDISPLAY >$ to denote the start of the results and $< CLASSTYPE = "classname" NAME = "attributeName" >$ to represent an object

and $< COLLETIONTYPE = "className" NAME = "attributeName" >$ to represent the attributes which are of type collections.

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

/*Global Variables**/

/*   formatterObject points  to the current Formatter object */

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


/*Local Variables*/

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

GenericType obj1;

 ClassController *conObj;//ClassControllerObject;


/*Algorithm*/

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

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

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

Table 6.6: **Pseudo code for OshadhiDisplay Function**

Chapter 7

Query Optimizations in Oshadhi

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

7.1 Path Expression

As explained before in Section 1.2 the presence of path expression in queries poses a serious problem in terms of time. The relationship of between the objects forming a hierarchy can be represented using a multi-level graph called as *aggregation graph*, where each level i represents objects of a type T_i. Each node n_{ij} denotes an object o_{ij} which is of type T_i. Each edge denotes the attribute relationship between two objects. So a path of type a.b is represented by an edge from object a to object b. A generalized path expression is of the form $o_0.a_1.a_2.\ldots.a_n$, where o_0 is an object of type T_0 and a_i is an attribute of type T_i in the type T_{i-1}, $1 \leq i \leq n$. A path expression $o_0.a_1.a_2.\ldots.a_n$ specifies a collection of objects o_{nj} of type T_n, where T_n is at level n deep in the aggregation hierarchy starting from type T_0 and the objects o_{nj}, $j \geq 0$, are reachable from o_0 in the *aggregation graph.*

7.1.1 Evaluation of Path Expression

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

7.1.1.1 The Naive Method

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

7.1.1.2 Path Expressions into join

The drawback of the naive method mentioned above is that it needs n lookups for each object o_{0j} if the path is of length n, Still worse if one of the attribute in the path is a collection. (By visualizing the *aggregation graph*, one can come to a valid conclusion that the set of paths in the *aggregation graph* starting at level i and ending at i+k is equivalent to a join of the extents associated with the types $T_j, i \leq j \leq i$+k). So, one of the optimization followed by some of the optimizers, including Lambda-DB, is to convert the path expression into a join [Feg97, BKG93] and then select from the resulted join table. This is also known as *materialization* of path expressions [CD92].

7.1.1.3 Path Dictionary

A data structure called *Path Dictionary*, which stores all the objects that are related by the attribute relationships, is proposed in [LL98]. Using this, evaluation of any path expression $o_{0k}.a_1.a_2.\dots.a_n$ turns out to be a lookup which returns a collection of objects which are reachable from o_{0k} and are at the depth of n in the aggregation graph. For fast retrieval it stores the OIDs of the objects involved in the attribute relationship in a generalized list structure. (for a detailed description on path dictionary see [LL98] and see [Tad00] for a detailed description of how it is implemented in Oshadhi).

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

7.1.2 Implementation Details

In Oshadhi, path dictionary is chosen as the index mechanism for aggregation hierarchy.(for more details see [Tad00]). This sub-section gives the implementation details of the regarding the optimizations using the path dictionary.

7.1.2.1 Index Creation

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

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

7.1.2.2 Transformation of a Query with Path Expression

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

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

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

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

Table 7.1: **Path Expression in Monoid Comprehension Calculus**

By the definition of monoid comprehension calculus defined in Section 5.2, a monoid comprehension calculus

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

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

For example, considering the query

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

and assuming that the index is created as shown in Section 7.1.2.1. then the monoid calculus expression is as shown below:

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

The internal representation of this query, equivalent to the above monoid comprehension, is shown in Table 7.1. Here the expression denotes that the result collection type is a bag of plants and the projection variable is p and the remaining represents the comprehension where each item is a data generator or a predicate.

- **Translating into algebraic form**

  In the phase of translating the monoid calculus expression into monoid algebra, as mentioned in Section 5.8, all data generators are either converted into a join (if the domain is an extent) or an unnest operator (if the domain is an expression). So, here the expression having pditerator expression will be converted into an unnest operator as this is a data generator and it is dependent on another variable. Refer to 5.3 for a discussion of unnest operator. The "path" argument to unnest operator, here, is the *pditerator* expression which generates the data. For the above example the query, in the algebraic form, is shown in Table 7.2.

- **Physical Plan Generation**

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

Table 7.2: **Path Expression in Monoid Algebra**

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

Table 7.3: **Path Expression in Physical Plan**

In this phase the physical operators implementing the algebraic operators will be substituted and the above example query's physical plan is as shown in Table 7.3

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

7.1.2.3 Index Updation

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

If an object of type T is created/updated then all path dictionary indices having this type T as one of its component in the path will be updated by the query processor. For example, if path dictionary index is created as shown in Section 7.1.2.1 then the query processor will generate code to update the path dictionary index pid1 whenever any of the objects of type Plant, State, City or point is created/updated.

7.2 Derived Attributes

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

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

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

7.2.1 Implementation Details

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

7.2.1.1 An Example in ODL

Suppose an application needs a Rectangle class and it need to represent the co-ordinates and Area and the database designer has chosen to represent Area as a function rather than an attribute. Then the class declaration in ODL declaration will be as shown in Table 7.4.

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

Table 7.4: **An Example in ODL For Derived Attribute**

7.2.1.2 Index Creation

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

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

where *bid1* is the index identifier.

7.2.1.3 An Example Query

Suppose the query is

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

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

Chapter 8

Future Work

In the last few chapters we have seen the the making of a query processor and optimizer for Oshadhi. In this process we have discussed about the Data Definition Language and Query Language for Oshadhi. We have also seen how query processing is done, what query optimizations are done in Oshadhi and how they are implemented. We have also discussed the salient features SHORE and Lambda-DB on which the Oshadhi's query processor and optimizer are built.

8.1 Extensions

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

- **Spatial Join:** There exists several spatial join algorithms which exploit the properties of particular spatial access methods [Güt94]. So, if Oshadhi is extended to support spatial joins then the query processor can be extended to support this spatial joins.

- **Index on Class Hierarchy:** Oshadhi supports indices to be built on class hierarchies. The index chosen for this purpose was MT-index (for more details regarding the decision to use this mechanism and the details of MT-index see [Tad00]). Query processor can be extended to support this indexing mechanism where user can optionally built an index on the class hierarchy and optimizer will take care of it's presence and use it in query processing for the queries on class hierarchies.

- **Schema Evolution:**

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

- **Optimization of Several Queries:**

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

- **Optimization of user defined functions:**

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

Chapter 9

Conclusions

Oshadhi is an object-oriented biological information system developed to cater the needs of the biological community. Bio-diversity data includes taxonomic data (like details of all the Species, Genera to which this species belongs, their classification and characteristics) , spatial data (like details of the geographic locations and their environmental conditions) and genomic data(like the protein sequences of the species). Oshadhi is built on top of SHORE, a database micro-kernel, which provides features like transactional facilities, persistence and a Data Definition Language called SDL(SHORE Definition Language).

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

The query optimizer developed was a query-rewrite rule-based optimizer, built by extending Lambda-DB's optimizer [Feg97]. Query optimizer tries to optimize the query by taking into consideration the presence of indices like path dictionary [LL98] on aggregation hierarchies and indices like Hilbert R-trees for indexing on spatial data types. Query processor supports indexing to be done based on parameterless member functions rather than just the attributes and optimizer optimizes the query by using this index if the query has a member function on which index is built. Query processor communicates with Visualization to receive queries in OQL and sends the data and the meta-data for displaying them. The interactions between the query processor and visualization are designed in such a way that Visualization can display the data and meta-data in its own format rather than query processor fixing the format of displaying the results.

Appendix A

DDL of Oshadhi in BNF

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

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

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

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


<ForwardDcl>::= INTERFACE <Id>;

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

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

<AttributeNameList>::= <Id>{,<Id>};
```

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

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

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

<ConstrTypeSpec>::=  <StructType> |<UnionType>| <EnumType>
<Declarators>::=   <Declarator> {, <Declarators> }
<Declarator>::= <Id> | <ArrayDeclarator>
<IntegerType>::=  [  UNSIGNED  ]   ( LONG | SHORT )
<StructType>::= STRUCT <Id> '{' (Member)+ '}'
<Member>::=<TypeSpec> <Declarators>;
```

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


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


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

Appendix B

Query Language of Oshadhi in BNF

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

<destination> ::= <query>.<Id>
```

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

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

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

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

//Constructor
<query> ::= <Id> '(' [<query>] ')'
        <query> ::= <Id> '(' <Id> : <query> {, <Id> : <query>} ')'
        <query> ::= STRUCT '(' <Id> : <query> {, <Id> : <query>} ')'
        <query> ::= (SET | BAG | LIST ) '(' [<query>{,<query>} ] ')'
        <query> ::= '(' <query>, <query> {,<query>} ')'
        <query> ::= [LIST] '(' query .. query')'
        <query> ::= ARRAY '(' [query {, query} ] ')'

//Accessor
<query> ::= <query> . <Id>
<query> ::= <query> . <Id>( <query> {,<query>} )
        <query> ::= * <query>
<query> ::= <query> '[' <query> ']'
```

```
        <query> ::= <query> '['<query>:<query>']'
        <query> ::= first'('<query> ')'
        <query> ::= last'('<query>')'
<query> ::= <Id>'(' [<query> {, <query>}] ')'

//Collection Expression
   <query> ::= FOR ALL <Id> IN <query> : <query>
        <query> ::= EXISTS <Id> IN <query> : <query>
     <query> ::= ( EXISTS | UNIQUE | COUNT |SUM | MIN | AVG ) '(' <query> ')'
     <query> ::= <query> IN <query>
     <query> ::= <query>
     (EQUAL | NOTEQUAL | GREATER | LESS | GREATEREQUAL | LESSEQUAL)
     ( SOME | ALL | ANY ) <query>
     <query> ::= COUNT '(' * ')'
//Select Expression
<query> ::= SELECT [ DISTINCT ] <projection_attributes>
    FROM <variable_declaration> {, variable_declaration }
    [ WHERE <query> ]
                 [ GROUP BY <partition_attributes> ]
    [ HAVING <query> ]
                 [ ORDER BY <sort_criterion> {,<sort_criterion>} ]
   <projection_attributes> ::= <projection> {, <projection>} | *
   <projection> ::= <query>
<projection> ::= <Id> : <query>
<projection> ::= <query> AS <Id>
<variable_declaration> ::= <query> [[AS] <Id>]
<partition_attributes> ::= <projection> {, <projection>}
<sort_criterion> ::= <query> [ASC | DESC ]

//Set Expression
<query> ::= <query> (INTERSECT  |UNION  |EXCEPT   )<query>

        <query> ::= ( LISTTOSET | ELEMENT | DISTINCT | FLATTEN ) '(' <query> ')'
        <query> ::= '(' <Id> ')' <query>
```

Appendix C

Meta-Data Export format in BNF

```
<Export-Format> := <className> "(" <Export-Dcl> { "," <Export-Dcl> } ")"
<Export-Dcl> ::= <attrib-Dcl> | <relationship-Dcl> | <method-Dcl>
<attrib-Dcl> ::= attribute "(" <attrName> , <attrType> ")"
<method-Dcl> ::= method "(" <methName> , <returnType> ,
                      params "(" <params-Dcl> { , <params-Dcl> } ")" ")"
<params-Dcl> ::= ( in | out | inout ) "(" <attrName> , <attrType> ")"
<attrType>   ::= <Identifier>
<methodType> ::= <Identifier>
<className>  ::= <Identifier>
<methName>   ::= <Identifier>
<Identifier> :=  [a-zA-Z][a-zA-Z0-9_]*
```

Bibliography

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

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

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

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

[Che00] Madhav Rao Cheethirala. The spatial perspective of oshadhi. Master's thesis, Dept of CSA, Indian Institute of Science, January 2000.

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

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

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

[Feg97] Leonidas Fegaras. An experimental optimizer for oql. Technical Report TR-CSE-97-007, University of Texas at Arlington, 1997. Available at http://www-cse.uta.edu/ fegaras/oqlopt.ps.gz.

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

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

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

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley Longman Inc., 1995.

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

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

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

[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.

[Tad00] Rajarao Tadimeti. Database habitats for biological species: Handling taxonomy data in oshadhi. Master's thesis, Dept of CSA, Indian Institute of Science, January 2000.

[Vid95] R. Vidya. Oshadhi a biodiversity information system. Master's thesis, Indian Institute of Science, June 1995.