System Integration, Testing and Evaluation of BODHI

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Department of Computer Science and Automation

by

Uday Sankar Sen

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

FEBRUARY 2001

Acknowledgements

I take privilege to express my deep sense of gratitude and sincere thanks to my guides Dr. Jayant R. Haritsa for his valuable guidance, genuine suggestions, encouragement and extensive cooperation during the period of project.

I would specially thank my colleague *Srikanta* for his useful suggestions in the project work and for his ever helping hand. I thank the remaining *DSLites*: *Amit*, *Maya*, *Vikram* and *Suresha* who made DSL a homely place to work. I would always cherish the time that I had spent with them.

I take the occasion to express my acknowledgement to all instructors, providing their valuable advice and encouragement throughout my post graduation. I am grateful to all my friends, who have cooperated me during the course. Finally and most importantly, I extend my gratitude to my beloved parents who have been source of inspiration and moral support to me, without whose support this work might not have been materialized.

Contents

List of Figures

List of Tables

Abstract

BODHI[14] is a biological database system, built using OO micro-kernel *SHORE*[1]. Previous work on the system focuses on various components of the system. Integrating and testing the whole system needs analyzing the system and accordingly needs some changes in existing codes of various modules of the system with minimal effect on system performance. In this paper some issues during integration of spatial module with taxonomy module in BODHI are discussed.

Unlike commercial transaction processing, the queries in biological system are more complex and compute-intensive but are not heavily loaded. So, attention is paid on supporting various functionalities rather than performance evaluation of *Bodhi*. Majority of biodiversity data is spatial data, needed to model geographic information associated with the species. On addition, the costliest operation in the system being spatial operators, evaluation of spatial module is extremely necessary for evaluating whole system. In this paper some of the complex queries are discussed and also some evaluation result is produced.

Chapter 1

Introduction

Rich biological diversity in the country can be used for various benefits, primarily drug manufacturing. This certainly leads to growing needs of proper documentation of these species. *Bodhi*, the Biodiversity Object Database arcHItectecture, caters the need along with various other needs for ecologists and biologists. *Bodhi* is built on top of an object-oriented storage manager,*SHORE*. Inherent complexity and inheritance nature of data can be handled relatively easily by object database. 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. Operation and architecture of *Shore* is described in chapter 3

The high level architecture of the system is depicted in Fig 2.1. As depicted in figure, the system comprises of five main components, viz., Object Services, Spatial Services, Sequence Services, Query Processor and Visualization Client. Biodiversity data can be broadly classified into four groups: taxonomic data, geographic data, genomic data and bibliographic data. In *Bodhi* individual modules are defined to handle these different data types. Object Services store and access taxonomic data. Spatial Services are responsible for spatial data management and access methods for those data. Genome Services provides data model for genomic data management and various algorithms to support queries based in similarity search. More details of each components are given in chapter 2.

In all the previous works over *Bodhi*, each modules are separately built and tested. Spatial data represents geographical location of a species. So, queries in various biological community is associated with spatial data access. In addition spatial queries are complex and compute-intensive. This makes spatial module one of the important modules that needs to be evaluated. In this project spatial module is integrated in the system along with taxonomy module. On course of integration several incompatibilities and bugs in the system are noticed. As shown in Fig 2.2, schema definition is done through query processor and the storage is taken care by *Shore*, at lower level than query processor. Any methods that is called for only reading underlying object needs to be defined as *constant* in SDL (*Shore* Data Language). But during schema definition, the operations of the class of objects cannot be defined as *constant*s. This incompatibility between *Shore* and the query processor leads to entry of operation in log record, even though the method is called for reading only. This unexpected log entry leads to log overflow, crippling the operation of *Shore*. In addition several other problems are found in *Shore* itself. One interesting problem noticed is index once built on any extent cannot be deleted until the extent itself is destroyed. This restricts the flexibility of user operations. Unlike SQL, **Drop Index** cannot be implemented here. In this project several such vulnerable errors are taken care of. In this report several issues, that are faced during addition of spatial module, are discussed in chapter 6. The query processor is built using *lambda-DB*, which converts all the schema definition and queries from user level to SDL, understood by *Shore* as shown in Fig 2.2. The major hurdle in this project is incompatibility of code generated by *lambda-DB* and the code that should ideally be generated. So, several extensions to *lambda-DB* is done. Some of them are mentioned in chapter 6.

As mentioned earlier in any biological information system, a large extent of queries are associated with spatial operations. Some of them are picturized in Fig 1.1. Some example spatial queries to illustrate the pictures are

- Retrieve all the forests that are in Karnataka
- Select all the states that contains himalayan vegetation and forest area more than

100 sq. km.

- Select all the cotton growing areas that are within a distance of 300 km. from Mumbai.

Some of these types of queries are described in chapter 7.2. As seen in Fig 1.1, spatial queries include several overlaps, adjacency, containments and disjointness. For example, in case of second query above, all the polygon objects that overlaps with another set of polygon objects needs to be retrieved. Mostly all the queries in spatial domain like overlaps, containments include joins. All such joins in spatial queries mostly resemble the range selection and joins in non-spatial domain which are complex and compute-intensive and hence costly. Again, unlike non-spatial database system, where number of tuples in one page is quite large, spatial data type being inherently complex and hierarchical, there can be very few objects in the type in a page. This means disk access increases, if the number of objects are same as number of objects in non-spatial system. All these constraints makes spatial data management more complex. This complexity and dominance of spatial queries in biodiversity application makes it essential to evaluate the spatial module in the system.

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

- Functional Benchmarking, where the queries cover the ranges needed for the application. System throughput, and other system performance issues are generally not measured using this benchmark.

- Performance Benchmarking, where performance evaluation of system is done by stressing the system with data. The stressing of system is done by scaling up the data.

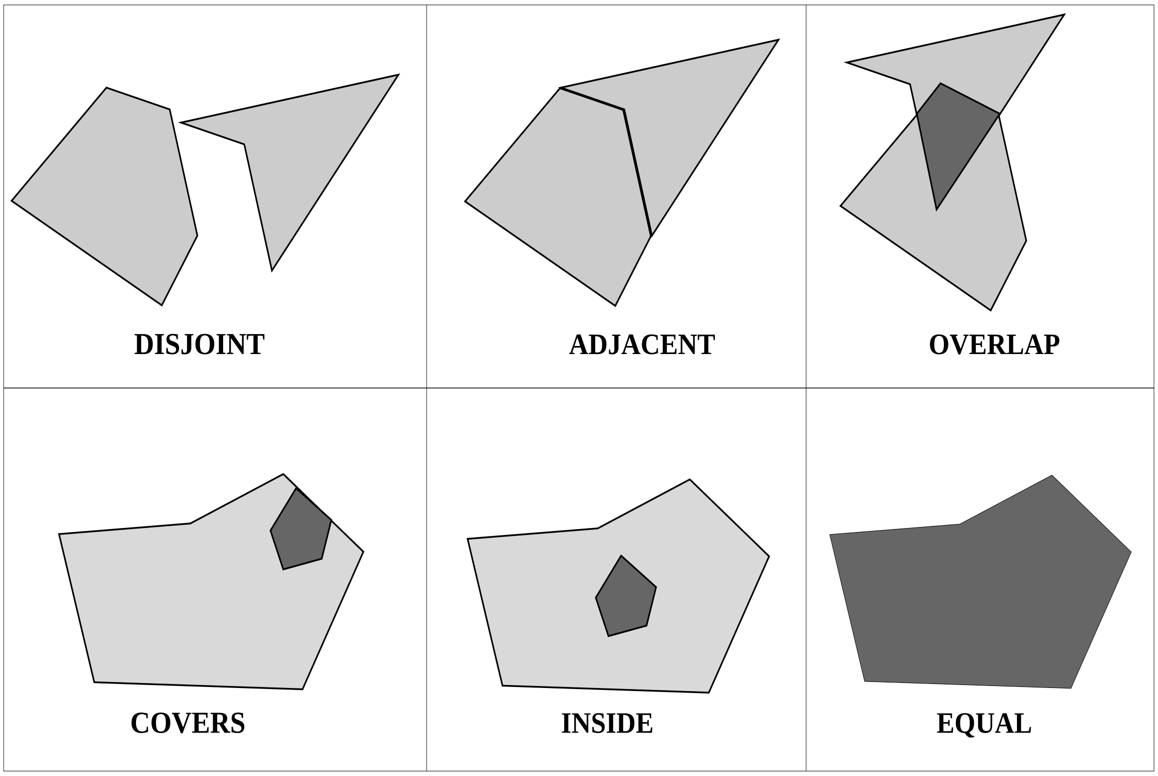

Figure 1.1: Spatial relationship figure

The choice for functional benchmarking for evaluating spatial module rather than performance benchmarking stems from the fact that the spatial queries varies to a large extent, from simple non-spatial selection to complex spatial joins. This leads to opt for *Sequoia* 2000 benchmark[15]. *Sequoia* 2000 is a benchmark that captures the data base requirements of earth scientists working in SEQUOIA 2000 project. The benchmark can also be used by Geographic Information Systems (GIS) and several other engineering and scientific DBMS users including biologists. This benchmark contains real queries that represents the tasks of biologists. Moreover *Sequoia* does not scale up data to stress system. Hence, it only looks at various functional constraints of database rather than system performance. This leads to choice of *Sequoia* for evaluating spatial module. Other than queries related to raster data, *Bodhi* supports *all* queries mentioned in *Sequoia* 2000 benchmark. Raster data is still not supported in *Bodhi*. In near future, this will be implemented in spatial module. Some evaluation results are mentioned in chapter 7.3.

1.1 Motivation

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

1.2 Related Works

There are several geo-spatial systems available from both commercial sectors and research institutes. *Paradise*[13] is a geo-spatial system built on top of *Shore*. It uses object-relational database system. Its main goal was to test scalability of various geo-spatial query processing techniques. Although the motivation of *Bodhi* is different than *Paradise*, as spatial module is essential part of bio diversity application, it needs to support various features of spatial system. *Paradise* also uses *Sequoia* benchmark for evaluating their system. They scale up their data to achieve the goal project as mentioned before. *Paradise* extend SQL to support spatial data primitives and set of spatial operators. Whereas *Bodhi* uses ODL (object Query Language) for schema definition and OQL (Object Query Language) for query. All the data types are implemented as abstract data type using C++ in both the systems. Since object-oriented system is used in *Bodhi*, it is possible to handle complex queries like recursive queries, which were not tested by *Paradise*.

1.3 Organization of Paper

The rest of the paper is organized as follows. In next chapter the architecture and some details of the system integrated will be described. In chapter 2 the system is briefly described. The implementation level problems are addressed in chapter 6. It also mentions the previously unsupported part of the system and all the extensions made to handle problems in incompatibility in interface. Section 7 presents the benchmark data and descriptions of queries comprises the benchmark, along with some performance result. Then in chapter 8 the present status and future direction still needs to enhance system performance is mentioned.

Chapter 2

Overview of *Bodhi*

In any bio-diversity database there is the a need for integration of diverse data types as mentioned later in this chapter. *Bodhi* is designed as an generic object oriented database management system and is tuned to handle complex, diversified biological data. Object oriented paradigm provides for flexible and extendible data modeling power, intuitive for a domain specialist. Moreover, accounting for inherent complexity of data in biological domain, object oriented methodology turns out to be natural choice. The system is built on top of an object-oriented storage manager, *Shore* (Scalable Heterogeneous Object REpository). *Shore* takes care of the basic database features like transaction processing, persistence and recovery. The high level architecture of the system is given in Fig 2.1. The system also provides a set of efficient access structures over these objects to speedup the query processing.

In the rest of this chapter various components of the system will be discussed in brief.

2.1 Object Services

In an object-oriented system, schema basically consists of inheritance hierarchies, which are built by deriving attributes and behaviors from existing data types and aggregation hierarchies, which are built by combining the existing data types. In such systems, queries can be based on the position of the object in inheritance hierarchy and/or in aggregation

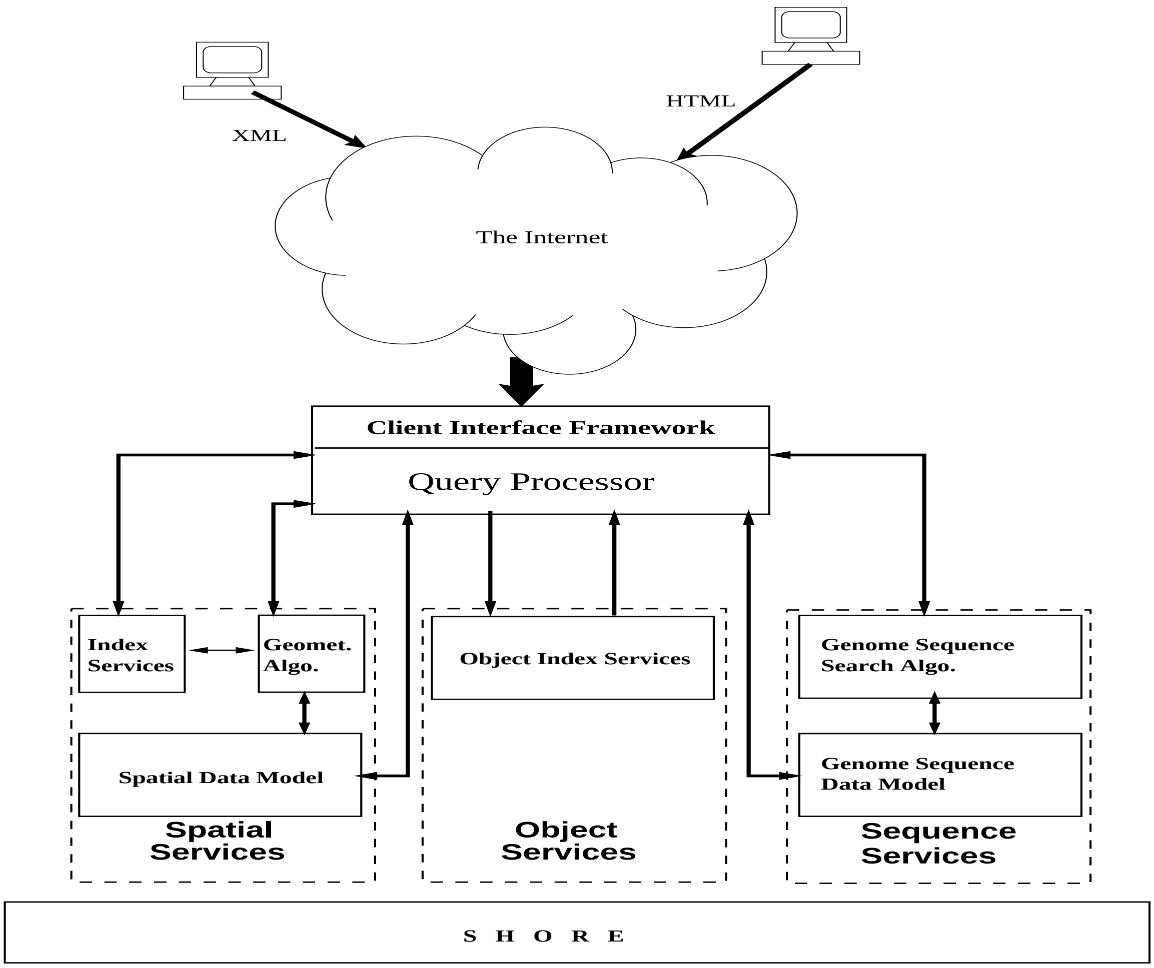

Figure 2.1: Architecture of *Bodhi*

path. In order to facilitate processing of such queries, Object Services have been designed to provide index services to the system. This module is responsible for storage and accessing the data from *Shore* storage manager. This module has two different indexing schemes :

- Aggregation Hierarchy Indexing Method : This access method can be used to efficiently answer the queries on aggregation hierarchies. Path Dictionary index[11] is implemented in the system as an aggregation hierarchy access method.

- Class Hierarchy Indexing Method : This can be used for efficient retrieval of data over queries on class hierarchies. The indexing method selected for *Bodhi* is Multi-key Type Indexing (MT indexing)[12].

The detailed description and implementation of these indexes in the system are found in [16].

2.2 Spatial Services

As mentioned earlier spatial data forms the key component of a bio-diversity data repository. Queries over spatial data is more compute-intensive than any other modules. In this module the data model is defined. Spatial data types are supported in this system. More recent index methods are also implemented in this module. Spatial data are broadly divided into two categories :

- Primitive types

- Compound types

Different data types are

- *Point*

- *Line*

- *Polygon:* This represents any type of land.

- *Layer:* It is a cluster of several polygons. This data type is supported to store several related polygons as a single object.

- *Network:* This is collection of several polylines. It is used to represent the rivers, roadways.

Primitive indexes like hash-index and B^{+}-tree handle single dimensional data well. But all the basic spatial data types mentioned above are multi-dimensional. Hence, a variant of R-tree[7], Hilbert R-tree[9] is used. Implementation details is available from [3].

2.3 Genome Services

Biological sequences are DNA strings, and genes are subsequences of such strings which are responsible for determining certain species characteristics. The sequences are extremely long strings. The similarity of such subsequences is an indicator of similarity of species characteristics. Thus, in genome databases, a common query is to locate similar sequences to a given DNA or Protein sequence. Sequence service provides similarity search algorithms over large string. In addition, this module also provide data model consisting of DNA, RNA and Protein sequence classes.

2.4 Query Processing

It optimizes the query and generates physical plan with least cost. It integrates all the features provided by the service modules in an effective manner and brings them to user. In particular, it is responsible for the followings :

- Enable user to specify database schema through ODL, extended to support the special data primitives and index structures provided in the system.

- Maintain meta data and statistics about the database for use in optimization of queries.

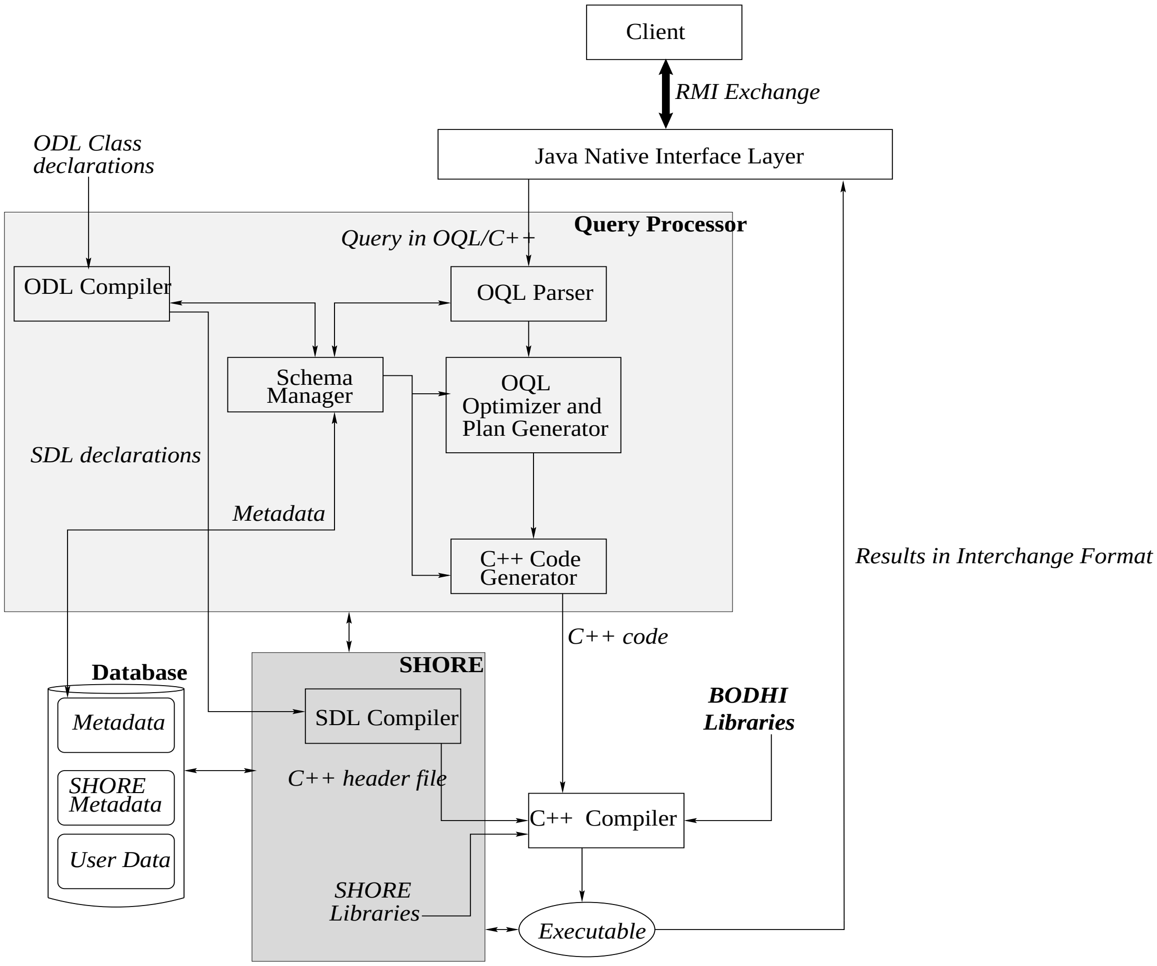

Figure 2.2: Query flow in *Bodhi*

- Allow users to pose queries, as OQL command strings.
- Parse, type check, optimize and execute the queries presented using extended OQL, after incorporating the optimization rules.
- Transform query results into an interchangeable format as requested by the user in the query.

The core of query processor module is *lambda-DB*[4], a rule-based optimizer for ODL/OQL. Fig 2.2 gives a proper view of flow of query processing. As seen in figure, schema is defined using Object Description Language (ODL) [2]. Several keywords are added to ODL, to represent data primitives provided by spatial and sequence service modules. ODL is been extended to cater to the need for supporting several modules. The ODL declarations are converted into SDL (*Shore* Data Language), a data definition language provided by shore storage manager. The *Schema Manager* of the query processor extracts the meta information and maintains it in the database. All further optimizations and type checking of queries depend on this.

Query can be posed in OQL (Object Query Language)[2]. Scheme for collection of queries across the web is still to be supported. The lambda-DB works on OQL with embedded C++. The optimizer consists of following stages:

- parsing OQL queries.
- translation of OQL queries into monoid comprehension.
- type checking the calculus. In this stage λ-DB checks the type of data requested, the type of parameter passed and all other type checking corresponding to the declaration of schema in ODL.
- query unnesting and translation of monoid comprehension calculus into monoid algebra.
- algebraic rewriting, including normalization of comprehensions and materialization of path expressions into pointer joins.
- join permutation using cost-based heuristic.
- physical plan generation using a rule-based cost-driven system.
- translation of physical plans into equivalent C++ code.

After C++ code is generated, query is evaluated in pipelined fashion. In λ-DB, all queries are optimized at compile time and corresponding executables are generated. This results in fast query execution as queries are not interpreted taking risk of suboptimal plan generation.

Chapter 3

SHORE

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

Most of the material in this Chapter is taken from [1] 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 multi-lingual data access.

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

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

3.2 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.

3.3 SHORE Software Components

The main software components of SHORE (Figure 3.1) constitute the SHORE server and the Language Independent Library. We next discuss these components in greater detail.

3.3.1 The Language Independent Library

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

To reduce paging, the object cache manager locks the cache in memory and uses LRU replacement if it grows too large. All OIDs in the cache are swizzled to point to entries in

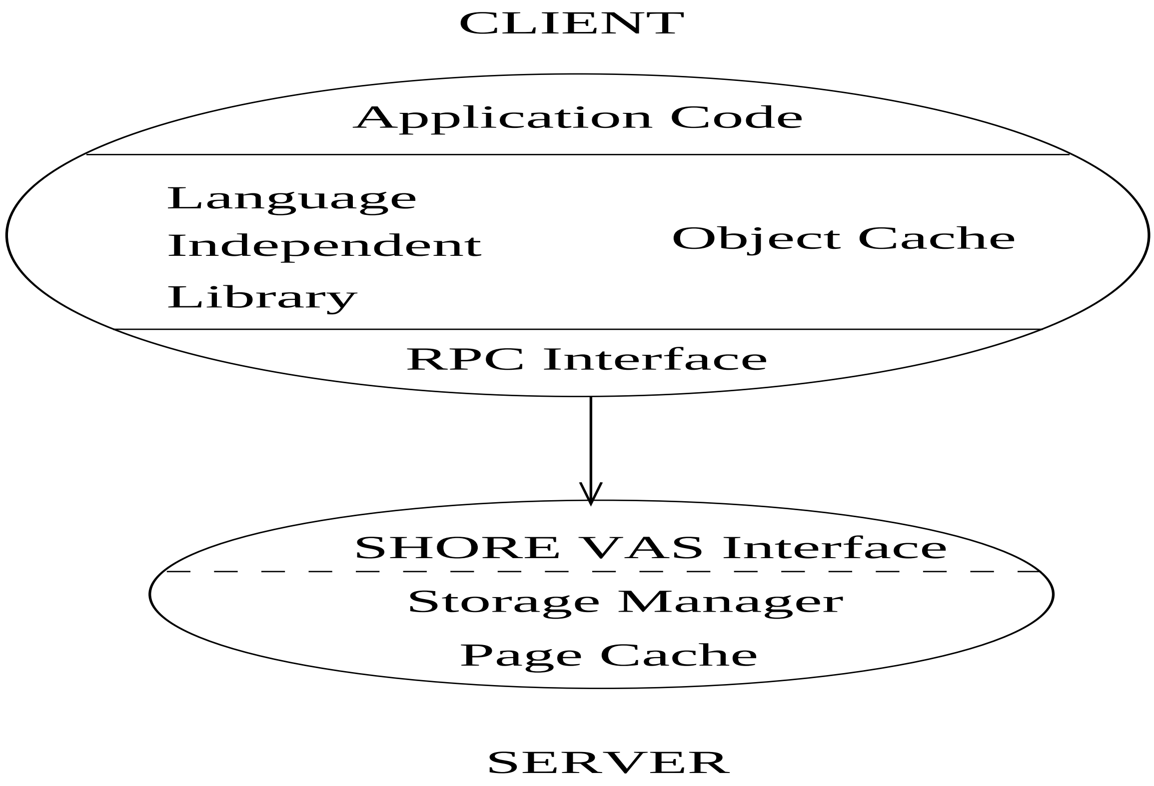

Figure 3.1: Application - Server Interface

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.

3.3.2 The SHORE server

The SHORE server (Figure 3.2) is divided into two main components: a *Server Interface*, which communicates with applications, and the *Storage Manager* (SM), which manages the persistent object store.

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

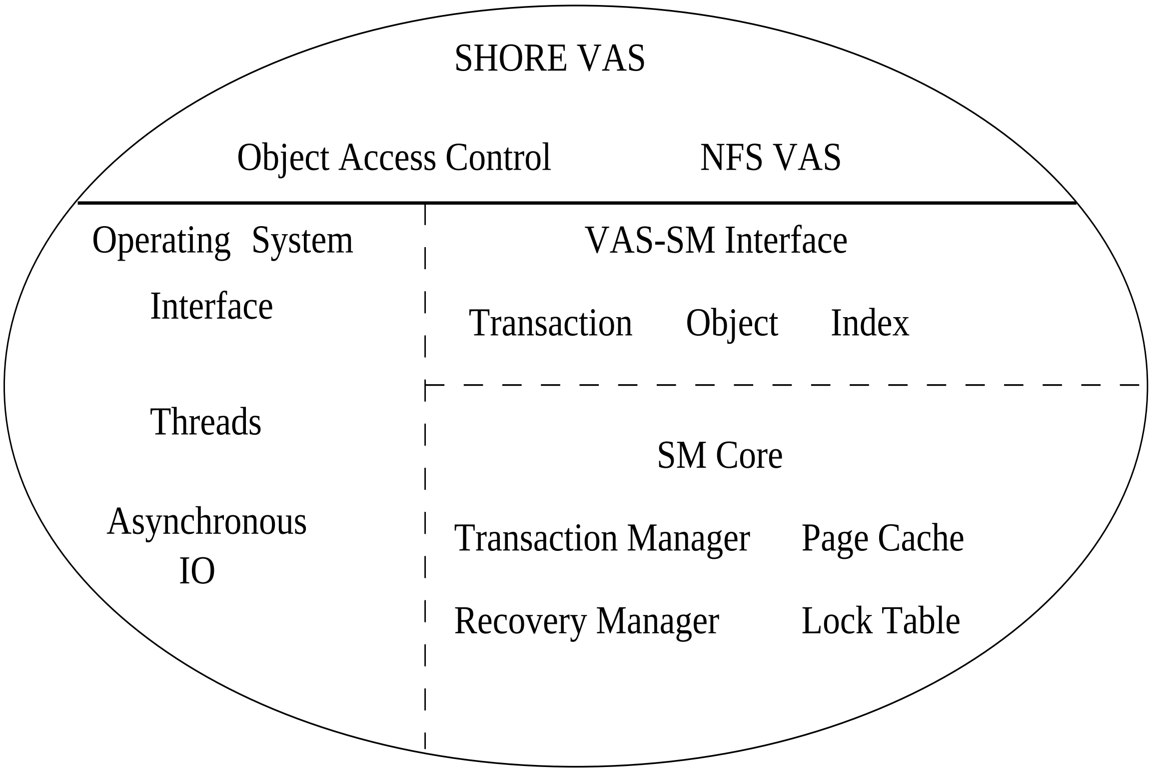

Figure 3.2: SHORE System Architecture

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.

3.4 Value Added Server

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

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

3.5 The VAS-SM Programming Interface

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

3.5.1 Storage Facilities

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

Devices

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

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

Volumes

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

Files of Records

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

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

B^+-tree Indexes

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

R^*-tree Indexes

An R-Tree is a height-balanced tree structure designed specifically for indexing multi-dimensional spatial objects. It stores the minimum bounding box (with 2 or more dimensions) of a spatial object as the key in the leaf pages. The current implementation in SHORE is a variant of R-Tree called R*-Tree [18], which improves the search performance by using a better heuristic for redistributing entries and dynamically reorganizing the tree during insertion. All the operations provided for B^+-tree implementation are also provided for R^*-tree.

3.5.2 Transaction Facilities

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

Transactions

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

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

Concurrency Control

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

The SSM uses a standard hierarchical, two-phase locking protocol [8]. For a file, the hierarchy is volume, file, page, record ; for an index, it is volume, index, key-value.

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

3.5.3 Crash Recovery Facilities

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

Logging

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

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

Checkpointing

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

Recovery

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

3.5.4 Thread Management

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

3.5.5 Communication and RPC Facilities

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

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

Chapter 4

Language For *Bodhi*

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

4.1 Data Definition Language

This section demonstrates the DDL of *Bodhi* with the help of an example which is used in describing the query language. The DDL of *Bodhi* is adopted from ODL (Object Definition Language), specified by ODMG (Object Data Management Group) [2]. As ODL is fairly standard [2], 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 Fig 2.2 for the flow of an ODL declaration) and as SDL does not support nested modules, this feature has been removed from ODL.

- ODL is extended to support the indexing to be done on derived attributes.

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

4.2 Query Language for *Bodhi*

OQL is a declarative query language standard proposed by ODMG and has been widely accepted as a query language for OODBMS [2]. This section gives a flavor of the query language designed for *Bodhi*, which is an extension of OQL.

OQL, by itself, is not computationally complete [2]. However queries can invoke methods written in the host language and any host language methods can include OQL queries. OQL supports only B+ tree index structures, which is not enough for *Bodhi*. So, OQL has been extended to support several index structures. The grammar for the query language of *Bodhi* is given in Appendix B. The following shows some example queries written in the query language adopted by *Bodhi*.

1. **Object Creation:**

   To create a persistent object, say of type City:

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

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

```
    module Taxonomy
    {
        // No import of module as everything is now in
        // single Taxonomy module
        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 *Bodhi***

2. **Simple query:**

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

   %for each v in (

   *select **

   from P in plants

   where P→name = "neem")

   do v→Print();

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

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

   %for each v in (

   select C

   from P IN plants ,

   C in P.specs

   where P.name = "neem"

)

   do v→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 v in (

   *select **

from C in cities, W in waterbodies

where C.name = "Bangalore"

and C.location.overlaps(W.location)

and W.type = LAKE

)

do cout<< v →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 hilID on Forest(boundary);

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

6. **Object creation:**

   The following example demonstrates query related to object creation.
   %forest_a:= forest(name:sundarban, boundary: polygon_x); This is constructor for simple object. To construct complex objects like set of objects or sequence of objects should be done only after creating the object. Currently object contain only collection type and nothing else cannot be consrtucted. It should explicitly have onr simple object type. And then insertion of collected objects are done.

7. **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:

tulasi.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:

tulasi.medicineFor $-=$ "Heart Attack";.

Chapter 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 in [4]. 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. 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.

5.1 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 [6]. 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:

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

5.2 Normalization

Another claim for monoid calculus is that it supports easy manipulation of query expressions which is amenable to pattern-based rewriting [6]. 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 [6], the algorithm in the form of OPTL rules is given in [4]. 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 5.1, 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 5.1. 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.

5.3 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).

If the query has any nested queries then they are unnested in this phase.

5.4 Materialization of Path Expressions

One of the most common technique for OODBMS is converting path expressions into joins [4]. It is also known as materialization of path expressions.

The Fig 5.1 shows how a path expression is converted into a join. The Fig 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 Fig 5.1 The detailed algorithms and rules in OPTL for this transformation is given in [4].

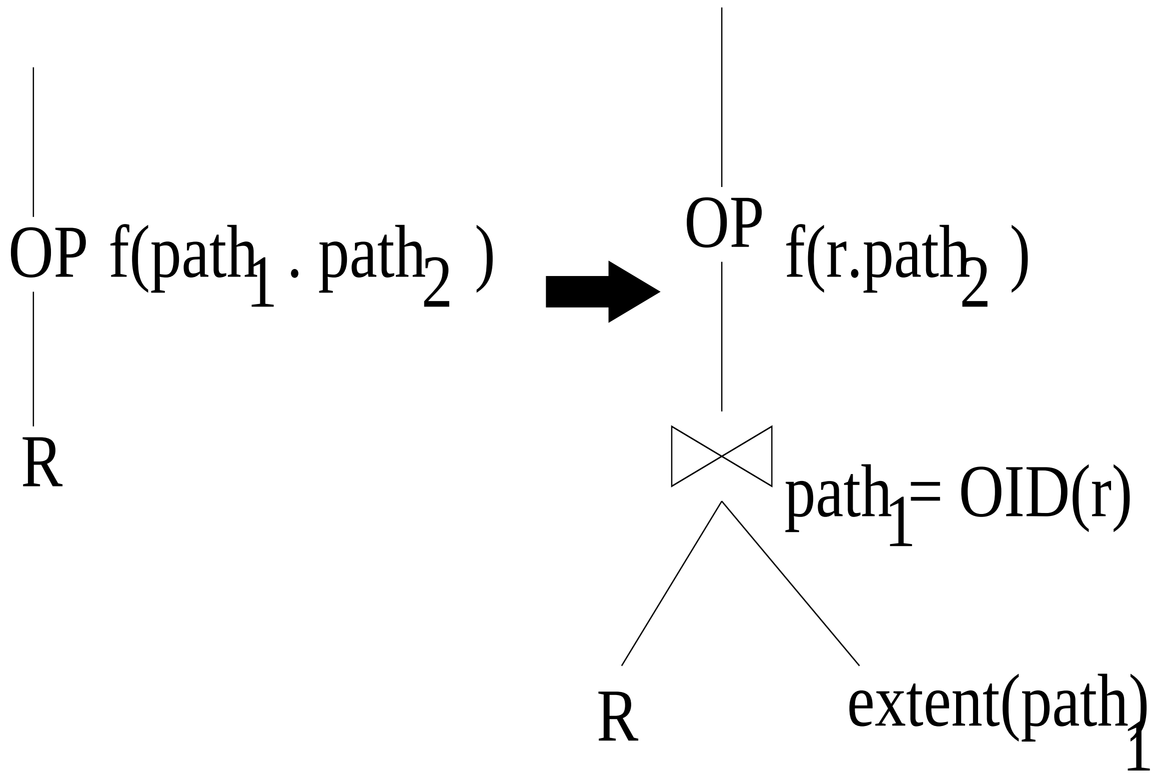

Figure 5.1: Path Materilization

5.5 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 [4]. 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).

5.5.1 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 $r_i \times_{P_{ij}} r_j$. 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 [5]. The GOO algorithm is not optimal always as it is a greedy algorithm based on heuristics. Refer to [4] for an example for which GOO is sub-optimal.

5.5.2 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.

5.6 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

Issues in Integration of Spatial Module

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

6.1 Extension of Schema Definition Language

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

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

6.2 Extension of Query Language

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

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

Almost all the benchmark queries except data loading, needs to call functions, with object as its parameter as one above. Details of such queries are found in chapter 7.2. This needs change of code on typechecking and parsing, where it fails. Plan generation however need not to be changed, because the information about the objects and schema is maintained already in database catalog.

Several *signatures* needs to be added to the primitive data types to support benchmark queries. Functions are added to support query 7 and 8 (refer chapter 7.2) of *Sequoia* which are actually query 4 and 5 in this report. Enhancement of query processor is done to support use of spatial index. **Closest** aggregate operator is added to support Paradise queries. But it is still not aggregated with the lambda-DB.

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

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

6.3 Problems in underlying *Shore*

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

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

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

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

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

Chapter 7

Benchmark Description and System Evaluation

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

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

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

- There is no specific benchmark, as mentioned earlier by which we could have evaluated the whole system using both taxonomic and spatial modules.

- Unavailability of real data set, where taxonomic data is associated with spatial data.

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

7.1 Benchmark Data

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

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

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

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

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

7.2 Query Descriptions

The queries for evaluating the system is *Sequoia* benchmark queries. As described in [15], it covers the range of queries necessary for this field. Followings are the queries of *Sequoia* that are supported by *Bodhi*

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

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

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

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

Query 4: Select all polygons that lie within a particular distance from a point and that are more than specific area. This type of query is one mentioned in example queries in chapter 1.This query requires combination of spatial and non-spatial selection

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

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

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

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

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

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

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

7.3 System Evaluation Result

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

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

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

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

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

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

1. The data loading was done across NFS, which inherently slows down disk access rate.

2. In *Shore*, pointer swizzling and unswizzling is the costly operation. Pointer swizzling of stored data to an object increases 3 folds the database access time. This also

causes overhead in queries. The problem can be solved if data is loaded from *shore* storage manager level instead of value-added server level.

3. Creation of objects has also overhead.

4. Bulk loading is not

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

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

Chapter 8

Conclusion and Future Directions

As mentioned above all the works on *Bodhi* are focussed on individual module, this project deals with inclusion of spatial modules into the system and evaluating it. In integrating the module, several interface problems are addressed. The importance of adding spatial module to the system is to evaluate the system. Since spatial operations are complex and costly, their inclusion in system may possibly slow down the system operation. The evaluation of the spatial module is done. The specification of system on which *Paradise* and *Bodhi* performed are given below. For further details one can refer [13].

- *Paradise* was tested using a cluster of 17 PCs configured with dual 133 MHz Pentium processors. Solaris 2.5 was used as operating system. Whereas *Bodhi* is evaluated using single Sparc/Solaris 2.5 workstation.

- Since the effort of *Paradise* is to parallelize the geo-spatial system, they need to look for better performance. Unlike them *Bodhi* is built to cater the need for biologists and ecologists. As mentioned earlier in this paper, transaction load in this application is not heavy. So, concentration on performance evaluation is though would have been advantageous, not addressed.

The formatter class is to be implemented properly which is necessary for generic visualization. Spatial joins using index is still to be implemented. Implementation of spatial tools like mining can enhance the performance of the system.

Appendix A

DDL of *Bodhi* 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 *Bodhi* 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>
```

Bibliography

[1] M. J. Carey et al. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, May 1994.

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

[3] M.R. Cheethirala. The spatial perspective of Oshadhi. Master's thesis, Department of Computer Science & Automation, Indian Institute of Science, Jan. 2000.

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

[5] L. Fegaras. A New Heuristic for Optimizing Large Queries. DEXA, 1998. Available at *http://lambda.uta.edu/order.ps.gz*

[6] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1995.

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

[8] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques.* Morgan Kaufmann, San Matego, CA, 1993

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

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

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

[12] T.A. Mück and M.L. Polaschek. A configurable type hierarchy index for OODB. *VLDB Journal*, 6(4):312-332, 1997.

[13] J. Patel et al. Building a scalable geo-spatial DBMS: Technology, implementation, and evaluation. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, June 1997.

[14] B.J. Srikanta and J. Haritsa. Design and implementation of a biodiversity information management system. In *Proc. of the 10th International Conference on Management of Data*, 2000.

[15] M. Stonebraker et al. The Sequoia 2000 storage benchmark. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, May 1993.

[16] R. Tadimeti. Database habitats for biological species: Handling taxonomy data in Oshadhi. Master's thesis, Department of Computer Science & Automation, Indian Institute of Science, Jan. 2000.

[17] R. Vidya and J. Haritsa. Oshadhi : A biodiversity information system. In *Proc. of the 7th International Conference on Management of Data*, 1995.

[18] N. Beckmann et al. The R^*-Tree: An efficient and robust access method for Points and Rectangles. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, May 1990.