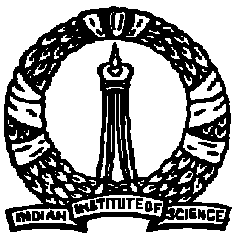
Implementation of SPINE Genomic Index and Graphical User Interface in BODHI

A Project Report
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in
Computer Science and Engineering

by

Abhijit Kadlag



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

JULY 2004

Acknowledgments

I sincerely thank Prof. Jayant Haritsa for providing me the opportunity of being a part of the highly spirited Database Systems Lab. I thank him also for providing all the encouragement, guidance and support during my work.

Database Systems Lab was a wonderful place to work in and I am thankful to all the members of DSL: Amol, Anoop, Kumaran, Maya, Parag, Shipra, Srikanta and Suresha for making the lab a lively place. My special thanks to Srikanta for all the helpful suggestions, timely help and all the cooperation as a project partner.

Finally with a deep sense of belonging I must mention that my association with the students, faculty and staff members of the Department of Computer Science and Automation for the last two years has been an extremely memorable and enriching period of my life.

Abstract

Modern bio-diversity research involves systematic and simultaneous study of macro- and micro-level relationship between various biological entities. BODHI[1] [3, 4] is a native object oriented database system which seamlessly integrates the *taxonomy, spatial and sequence* types of data occurring in bio-diversity studies.

BODHI implements the heuristics based *BLAST* algorithm to handle sequence similarity queries. Performance evaluation of BODHI indicated that these BLAST operations are far costlier than other operators supported in BODHI, indicating need for efficient sequence index structures. Popular sequence alignment packages like *Mummer* use *suffix trie* indexing techniques to get accurate sequence alignments efficiently, without using BLAST like heuristics. SPINE is a recently proposed, horizontally compressed suffix trie index which was reported to perform better than Mummer. The first part of this project was to integrate the SPINE indexing technique with BLAST implementation in BODHI to speed up sequence similarity queries.

BODHI fully supports OQL/ODL querying interface on the server side. However for bio-diversity researchers the system is more convenient if BODHI comes with a user friendly interface that allows them to specify queries without having to know OQL. The interface should take care of converting the user specified queries into OQL, run the OQL query on server and display the answers to users in an elegant manner. The second part of this project aimed at building a graphical query interface for plant bio-diversity database currently being hosted with BODHI. We have developed a web-based query interface which achieves this aim.

[1] BODHI is the tree under which Budhha gained enlightenment

Contents

List of Figures

Chapter 1

Introduction

Modern bio-diversity research involves systematic and simultaneous study of macro- and micro-level relationships between various biological entities. Multi-domain queries of the following kind are increasingly common among the researchers in this field:

> *Retrieve the names of all plant species that have common inflorescence location characteristics, share a part of their habitats, and have a high chromosomal DNA sequence similarity with Actinodaphne-bourneae.*

Answering this query requires the ability to process data across: (a) taxonomy hierarchies (*common inflorescence location*), (b) recorded spatial distribution of species (*common habitat*), and (c) genomic sequences (*chromosomal DNA sequence similarity*). Unfortunately, due to the lack of holistic database systems, biologists are often forced to split the query into component queries, each of which can be processed separately over specialized independent tools and services. Further, the individual results have to be combined either manually or through the use of a customized tool.

Motivated by this lacuna of an information management system that can support complex queries common to bio-diversity research, **BODHI** (Biodiversity Object Database arcHItecture) [3, 4], a native object-oriented database system that seamlessly integrates multiple types of data occurring in biodiversity studies, has been developed in Indian Institute of Science over last few years. BODHI expresses the sample multi-domain query presented above using an OQL(Object Query Language) [6] syntax as shown in Figure 1.1.

To the best of our knowledge, BODHI is the first system to provide such an integrated view of diverse biological domains ranging from molecular to organism-level information.

```
SELECT species2.name FROM
        species1 IN PlantSpecies, species2 IN PlantSpecies,
        dna1 IN species1.DNAEntries, dna2 IN species2.DNAEntries
WHERE
        species1.name = ”Actinodaphne-bourneae” AND
        species1.flowerchar.inflochar = species2.flowerchar.inflochar AND
        species1.georegion OVERLAPS species2.georegion AND
        dna1 BLAST dna2 WITHIN 70;
```

Figure 1.1: **Expressing a Multi-domain Query in BODHI**

1.1 Part I: The SPINE Genomic Index

BODHI achieves high performance by employing a variety of specialized access structures reported in the research literature for handling predicates over taxonomy hierarchies and spatial data. However performance evaluation of BODHI [4] indicated that sequence similarity operations are far costlier than other operators supported in BODHI, indicating need for efficient sequence index structures.

Suffix trees [10] (i.e.*suffix trie indexes*) are popular choice for indexing sequences in the database community. Widely used sequence alignment packages like *Mummer* use *suffix tree* technique to get accurate alignments efficiently. Recently Neelapala et.al. [21] suggested a novel horizontally compacted suffix trie index structure which was found to consume much lesser space as compared to the traditional suffix trees and more amenable for a disk-based version. It was also found to perform better on disk when compared to the *Mummer* [7] package popularly used for sequence similarity queries.

As the first part of this project we have integrated the SPINE [21] index structure with BLAST implementation in BODHI and studied the performance of the modified algorithm as opposed to the original BLAST implementation.

1.2 Part II: Graphical User Interface Framework

The BODHI database server supports full ODL (Object Definition Language) [6] interface for schema definition and OQL interface for querying the database. However for BODHI to be useful in a convenient manner to the bio-diversity researchers, it should provide a user friendly graphical query interface. The bio-diversity researchers should be able to express their queries using this interface and the interface should take care of converting the input information to OQL queries automatically. This user interface is required to be intuitive, easily learnable and should provide the maximum functionality possible. It should also be possible to query the database from anywhere in the world.

To provide the above mentioned functionality we have implemented a web-based graphical user interface for querying the database as well as for visualization of results. Easy dissemination of information is supported through the use of XML in publishing the results of queries, providing added semantics for future data exchange requirements.

1.3 Organization of the Report

Chapter 2 describes the architecture of BODHI in brief. Chapter 3 describes the SHORE storage manager which has been used as a backend for implementing the SPINE index. In Chapter 4 we describe the structure of SPINE index in brief and the relevant implementation details from the work of Neelapala et.al. Chapter 5 describes our work of re-implementation and integration of the SPINE index structure in BODHI. The issues involved in the design of the user interface, the technology choices, and implementation as a web based interface are explained in Chapter 6. Finally Chapter 7 summarizes the achievements of this project and the future work.

Chapter 2

Architecture of BODHI

The architecture of BODHI is shown in Figure 2.1. The SHORE [5] storage manager, available from University of Wisconsin, at the base provides the fundamental needs of a database server such as device and storage management, transaction processing, logging and recovery management. The application specific modules, which supply the object, spatial and genomic services, are built over this storage manager and form the functional core of the system. The λ-DB [11] extensible rule-based query processor and optimizer interfaces with these functional modules and performs query processing and produces efficient execution plans using the metadata exported by the modules. BODHI supports full OQL/ODL query and data modeling interface for creation of new database schemas, data manipulation and querying.

Finally, the user interface and XML publishing engine form the external interface to BODHI. The second part of this project was to implement this part.

The BODHI server is partitioned into three service modules: *Object*, *Spatial*, and *Sequence*, each handling the associated data domain. The service modules provide appropriate storage, a modeling interface, and evaluation algorithms for predicates over the corresponding data types.

2.1 Object Services

In querying over bio-diversity data, it is common to specify predicates over long relationship paths, or over an inheritance hierarchy rooted at a chosen base type. To efficiently handle these predicates, access methods for both inheritance (*multi-key type Index* [18]) and aggregation hierarchies (*path-dictionary index* [17]) are included in this module.

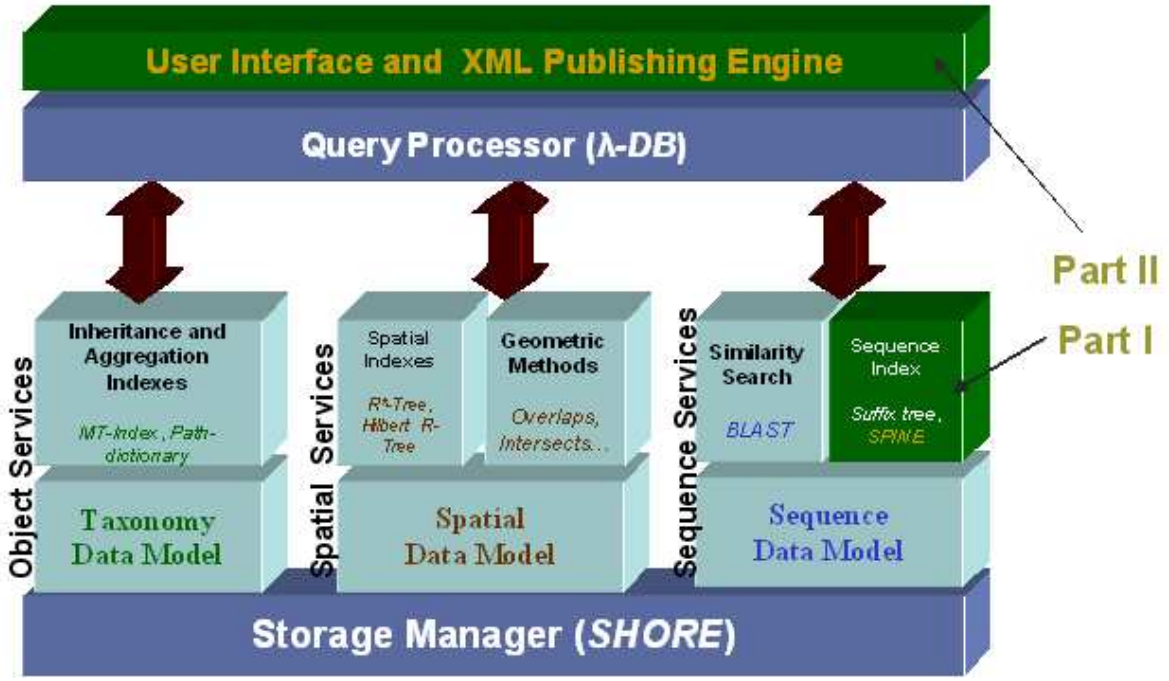


Figure 2.1: **Architecture of BODHI**

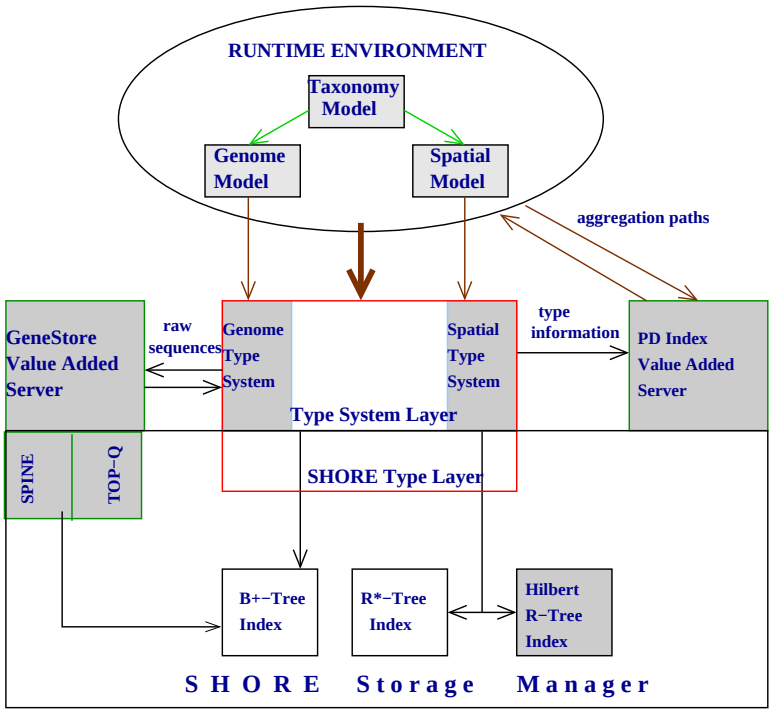


Figure 2.2: **Implementation of BODHI**

2.2 Spatial Services

This module provides a spatial type system for modeling of spatial data associated with biological information. Various geometric operators such as *overlap, adjacent, area*, etc., are implemented over this type system. The module incorporates *R*-Tree* [2] and *Hilbert R-Tree* [16] indexing to speed up these otherwise expensive operators.

2.3 Sequence Services

This module provides efficient storage and operations over genome sequence data of species. It implements the de-facto standard *alignment-based* sequence similarity algorithm of BLAST [1] A Value Added Server (explained in Chapter 3), GeneStore, handles the querying of sequence data for this algorithm. We have implemented the SPINE index in this module.

Figure 2.1 indicates the positions of the SPINE and user interface modules in the overall BODHI architecture. Figure 2.2 shows the implementation details of BODHI and the interactions between the various functional modules of the system. For more details on the implementation we refer the reader to [3, 4].

Chapter 3

The SHORE Storage Manager

Please Note: Most of the text in this Chapter is taken from [5] and SHORE manual pages [22].

SHORE (Scalable Heterogeneous Object REpository) [5] 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 of the BODHI database system. The SHORE system has also been used for the implementation of SPINE genomic index.

3.1 Architecture of SHORE

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.

3.1.1 SHORE Software Components

The main software components of SHORE (Figure 3.1) constitute the SHORE server and the Language Independent Library.

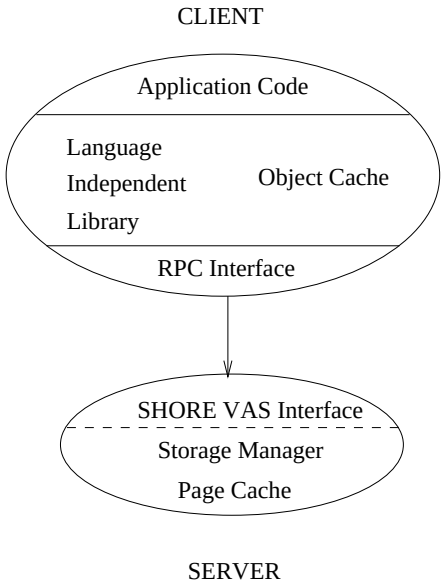


Figure 3.1: **Application - Server Interface**

3.1.2 The Language Independent Library

The Language Independent Library contains the object-cache manager which takes care of converting object references on disk to main memory addresses.

3.1.3 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.

3.1.4 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

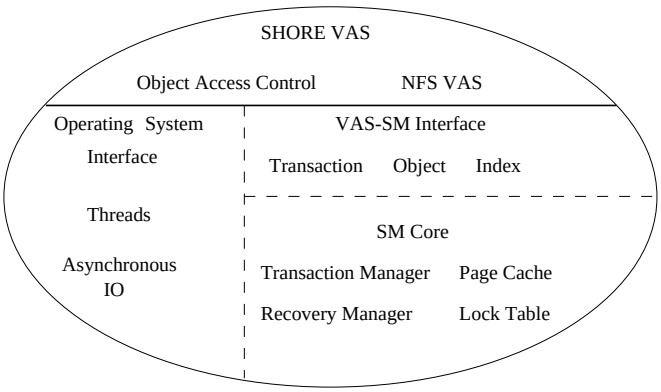


Figure 3.2: **SHORE System Architecture**

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.

3.2 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

the following Sections we describe some of the facilities that can be accessed through the VAS-SM programming interface.

3.3 VAS API : Storage Facilities

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

3.3.1 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.

3.3.2 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.

3.3.3 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 is 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/appending/truncating records and pinning records for reading are also provided as part of the interface.

3.3.4 B^+-tree Indexes

B^+ tree is the most popular indexing method in database community. The B^+-tree index facility in SHORE 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.

3.3.5 R^*-tree Indexes

An R-Tree [14] 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 [2], 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.

3.4 VAS API : 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.

3.4.1 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.

3.4.2 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 [13]. 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 VAS API : Crash Recovery Facilities

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

3.5.1 Logging

Updates performed by transactions are logged so that they 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.

3.5.2 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. However no interface is provided for a VAS programmer to control checkpointing.

3.5.3 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.

3.6 VAS API : 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.

3.7 VAS API : 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 [22].

Chapter 4

The SPINE Genomic Index

Please Note: Some part of the text in this Chapter is taken from [21].

Performance evaluation of BODHI indicated that it was the sequence similarity queries which were the costliest and affected the performance of cross domain queries. Thus the need for an index structure to improve the performance of sequence similarity queries was felt.

In this Chapter we describe the various methods used by biologists for sequence similarity queries and the structure of the SPINE genomic index, which has been proposed recently as an alternative to traditional suffix trie indexes.

4.1 State-of-the-Art

The sequence search tools that are currently used by biologists can be broadly classified under two heads [19]: *Seed-based*, exemplified by BLAST, the classical sequence alignment package [1], and *Suffix Tree-based*, exemplified by MUMmer [7], the recently-developed alignment software from Celera Genomics and TIGR (The Institute for Genomics Research) .

In the seed-based approach, the data sequence is first searched for exact-matches of short *seed sequences* from the query sequence. These seed sequences are stored in a keyword tree that is usually implemented as a hash table. The successful exact matches then form the candidates that are extended into better alignments [1].

In the Suffix Tree-based approach, on the other hand, an explicit index called the *suffix tree* [10] is created for the entire data sequence – this index stores all suffixes of the data sequence in a vertically-compacted trie structure. The popularity of suffix trees can be ascribed to their having linear (in the size

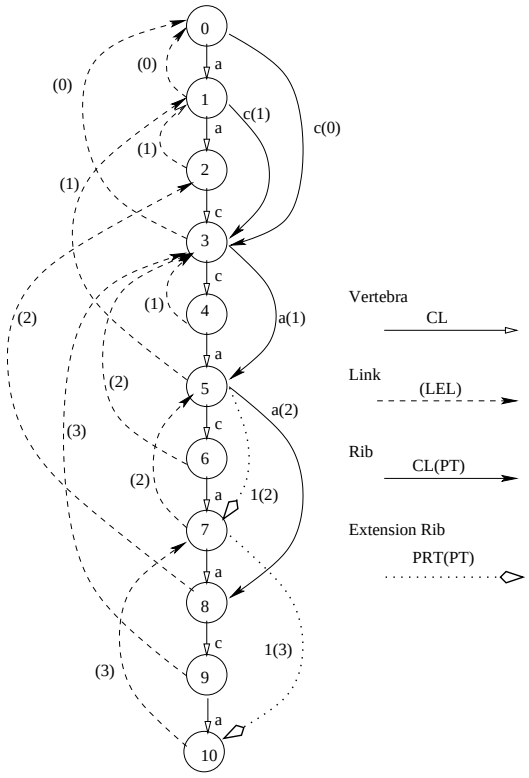


Figure 4.1: **Example SPINE Index (for *aaccacaaca*)**

of the data) construction time and space complexity as well as linear (in the size of the query) searching times. This tree is then used for finding all subsequence[1] matches of a given length of more (i.e. *seed sequences*) with the query sequence. These matches are then extended in both directions to get local alignments [7].

4.2 Structure of SPINE

Neelapala et.al. [21] recently presented a new index structure, called SPINE (Sequence Processing INdexing Engine), which they found to have a variety of advantages with regard to the suffix tree in terms of its search performance.

We introduce here, the SPINE index structure in brief. A sample picture of a SPINE index is shown in Figure 4.1 for the data sequence `aaccacaaca`. At its core, the SPINE index consists of a *backbone* formed by a linear chain of nodes connected by *vertebra* edges, representing the underlying genome sequence. The nodes are additionally connected by forward *ribs* and *extension ribs*, and backward *links*

[1] a substring is called subsequence by the biologists

for facilitating fast traversals over the backbone during the index construction and query search processes. All the edges have associated labels that are assigned during the construction process and are used to determine which paths are valid for traversal in the SPINE structure.

In particular, each vertebra corresponds to a character in the input data sequence, and this character is used to provide a *character label* (CL) for the vertebra. The ribs and extension ribs represent (in conjunction with the backbone) all possible suffixes of the data sequence. Each rib is also labeled with a character label, corresponding to the character that it represents in the associated suffix. The coalescing of all the paths into a single path leads to the possibility of introducing false positives during search on SPINE. To avoid this the ribs and extension ribs are labeled with a Pathlength Threshold (PT) which indicates the maximum path length that can be traversed before traversing that particular rib. The extension ribs have an additional label called Parent Rib Threshold (PRT) associated to identify them with a particular rib. The links have an integer label called Longest Early terminating suffix Length (LEL) which is also assigned during index creation and helps avoid false positives. Due to lack of space, for more details on the structure we refer the reader to [21].

From an abstract viewpoint, SPINE can be viewed as a **horizontal compaction** of the trie of the data sequence, in marked contrast to suffix trees which represent, as mentioned earlier, a *vertical* trie compaction. The motivation behind this horizontal compaction is to avoid the duplication of common segments among the various paths in the trie, thus reducing the number of nodes and thereby the space required to index a sequence. In fact, it carries this to the logical extreme of representing each character of the original data sequence only *once* in the index structure. This is in contrast to the suffix trees, where the number of nodes may go upto *double* the number of characters in the sequence.

Further, the improvement is not restricted to just the number of nodes, but the *size* of SPINE nodes, with the implementation of Neelapala et.al., is also smaller than their suffix-tree counterparts. The average index overhead per sequence character is about 12 bytes as mentioned in [21], whereas MUMmer requires 17.4 bytes per character indexed.

4.3 Original Spine Implementation

We enlist here few properties of the SPINE index structure, which are important for understanding the implementation aspects. These properties are discussed in more details in [21].

- Each node has a *link* associated with it.

- The *vertebras* need not be explicitly represented, due to sequential allocation of nodes.
- Each node can have from 0 to 4 rib entries.
- A node can have atmost one *extension rib* emanating from it.

Due to the above properties the original implementation of SPINE as mentioned in [21] uses five different tables. It consists of a **Link Table** (LT) and four **RibTables** (RTs), entries of which are shown in Figure 4.2. The LT contains one entry for each node (character) in the string. It stores LEL of the node's *link* as one of its columns while the other column represents either the destination node of that *link* (the LD field) or a pointer to an entry in one of the RTs (the PTR field). In particular, the LT stores the link destinations only for the nodes that don't have any *ribs/extension rib*. For the remaining nodes, they are stored in the RT entries only.

Each node features in at most one RT table. A RT entry for a node stores the destination node of the *link* of that node and also the destination nodes (the *RD* fields) and the threshold values (the *PT* fields) of all the *ribs/extension ribs* emanating from the node. And, lastly, the *PRT* field denotes the PRT value of the *extension rib*.

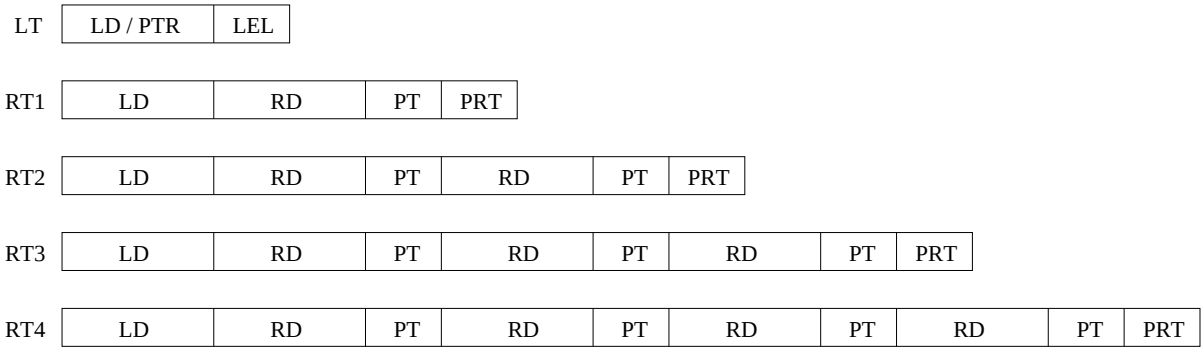


Figure 4.2: **Optimized SPINE Implementation**

By implementing all the above optimizations, the net effect is that the average node size in SPINE is *less than 12 bytes*, that is, the index takes upto 12 bytes per indexed character. The advantage of smaller node sizes is reflected not only in space occupancy but also in improved construction and searching times.

Chapter 5

Integration of SPINE with BODHI

We had at our hands the source code of implementation of SPINE from Neelapala et.al. The code written by them was required to be re-engineered in order to integrate it with BODHI.

In this Chapter we first describe the features of original SPINE implementation that required changes. We then discuss in Chapter 5.1, how we have changed these features and a few issues which are relevant to our implementation using SSM. Finally in Chapter 5.2 we discuss the integration of SPINE with BODHI which required SPINE to be merged with the implementation of BLAST in BODHI.

The features of original SPINE code which largely required re-engineering are:

1. Original implementation used operating system files for storing the entries in the LT and RT$[1..4]$ tables. Each table was stored in a single file. This needed to be changed to using a database storage manager, in particular the SSM.

2. Use of SSM required the storage technique to be changed. Original SPINE program stored records of each table in a separate file in a sequential manner. The records were accessed back using the record number as a key. Due to the limited programming interface provided by SSM, this storage scheme of one file per table could not be implemented and alternative storage scheme was needed to be devised.

3. BODHI is a client-server database system, which required us to support SPINE index creation and querying using a client-server interface. This needed significant code restructuring, modifications and adding RPC based client-server functionality.

4. BODHI allows for multiple users to access, read and modify the database simultaneously. When using an index like SPINE with BODHI, the ACID (Atomicity, Consistency, Isolation, Durability)

properties of all transactions need to be preserved, which was not required with the *stand alone* file-based implementation of Neelapala et.al. The SSM provides transaction facilities to preserve ACID properties. To use these facilities, the interface for adding, modifying, deleting and reading records was needed to be modified.

5. The original source code of SPINE was written in C, whereas BODHI has been implemented using C++. Thus the code was required to be rewritten by adding a class interface, making syntactic changes and providing proper interfaces for interacting with other BODHI modules.

5.1 Re-Implementation of SPINE

In order to support the above requirements we made the following changes to the implementation of SPINE:

5.1.1 Implementation as a VAS

We chose to implement SPINE as a Value Added Server (VAS). The advantages of implementing it as a VAS can be easily seen to be:

1. The code of SHORE was not required to be changed. Changes in the code of SHORE would have had system-wide implications on BODHI and every module might have required to undergo changes.

2. This easily satisfies our requirement of providing a client-server interface.

3. The index works as a separate module which interacts with other modules of BODHI, thus facilitating modular code and easier development.

4. It could further help the performance by running the SPINE server in parallel with SHORE and other VASs in BODHI.

5. The handling of index becomes independent of handling of sequences. Sequences are still being handled by the Genestore VAS already existing in BODHI. The SPINE VAS interacts with Genestore for fetching sequences to build indexes. (As mentioned in [21]: once the index is built, for subsequent querying one does not need to remember/know the original sequence.)

5.1.2 B^+ Trees for storing records

As mentioned in Chapter 3, the SSM interface provides both files of records and B^+ tree interfaces.

The simple solution of using a separate file for each of the five tables of SPINE was not found to be viable due to following reason: The interface for storing and retrieving records from files uses a structure called *serial_t* [22] as a key on the file. An *integer* key based storage and retrieval interface is not provided. However, all the records in the LT or RT$[14]$ tables need to be accessed by simple array-indexing style technique. Thus using files of records would require a mapping between the *integer indices* and *serial_t* structure. Such a mapping can be provided by using the B^+ tree structure. The index would store the *serial_t* structure as the data and use the *integer index* as the key. However, now the access to a record requires accessing both the index and file of records. To avoid this double access, we chose to store the record itself in the B^+ tree as data and use the *integer index* as the key.

Storing records in B^+ tree has the disadvantage of reducing fan-out of the tree and increasing sequential scan overheads but the advantage of avoiding access to a file as well as B^+ tree scores over the disadvantages.

Thus our implementation of SPINE uses five B^+ trees for storing each of the five tables explained in Chapter 4. This implementation has the advantage that after creation of the index, access to records can be done in logarithmic time while the earlier implementation in [21] had a linear time access.

5.1.3 Multiple SPINE indexes on the same volume

BODHI hosts a large number of species and their sequences. (One specie may have more than one sequence stored for it, e.g., all the chromosomes of that specie). It should be possible to create a SPINE index on each of the sequences on the same *volume* and access it back for querying.

To support this feature, our implementation uses a *master B^+ tree index* (Note that this is not the same as *root index* discussed in Chapter 3). The *master B^+ tree index* uses the *Logical Object IDentifier (LOID)* of the sequence as a key. The LOID which is assigned to each data item is guaranteed to be unique by SSM. Also, all the storage structures managed by SSM (e.g. records, indexes, files, etc.) have associated with them a *serial_t* structure guaranteed to be unique. The data indexed by the *master B^+ tree index* is a record containing the *serial_t* ids of the five B^+ trees (one for LT and four for RTs) of the SPINE index. Thus we can locate the LT and RT[1-4] tables for any of the species and any of its sequences on the *volume* after the index has been built. Figure 5.1 shows the design of storage structure.

Please note that the access to *master B^+ tree index* is made only once(per sequence) while creating

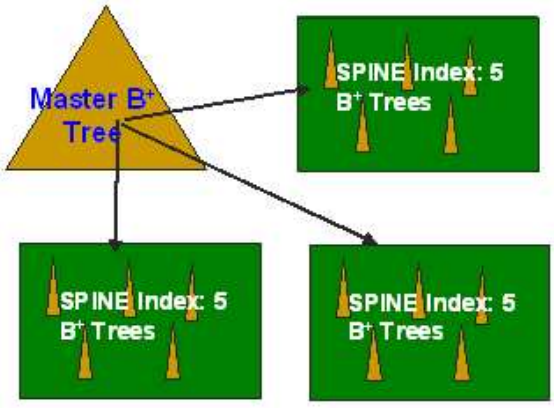


Figure 5.1: **Storage Structure Organization for SPINE**

the index or while using the index for querying.

5.1.4 Handling multiple transactions simultaneously

Using the VAS interface we have implemented SPINE as a multithreaded server. Thus multiple clients can connect to it simultaneously. As discussed in Chapter 3 the ACID properties of transactions are taken care by the SSM.

We have implemented client interface for SPINE in two versions: (a) a command line driven interface (b) a library interface. The command line driven interface can be used by users to access the SPINE server independent of BODHI while the library interface has been built for integrating the SPINE VAS with other softwares like BODHI. The server itself also supports the command line driven interface.

5.1.5 Record size optimizations

The implementation of SPINE as explained in [21] uses various optimizations to reduce the sizes of records. However in a C/C++ program these optimizations can not hold due to the alignment restrictions that the compiler imposes on the sizes of structures. Thus a record which holds total 6 bytes of data elements could be stored as a 8 byte record by the compiler. Neelapala et.al. treated the `C structures` as `arrays of characters` and stored the exact number of bytes on disk, in their implementation. Thus even the access to any of the elements inside the LT or RT table records was done by manipulating arrays of characters.

This could not be done with SSM because the interface provided by it accepts a structure for storage and not an array of characters. Thus the tricks employed in [21] could not be used anymore. Thus,

the optimizations suggested in [21] for the sizes of the records could not be implemented with SHORE. This resulted in the sizes of records with our implementation to be bigger than the implementation of Neelapala et.al.

5.1.6 Workarounds for log size limitations

SSM logs both the old and new values of data for all the updates(see Chapter 3). This makes log records of size twice the data modified. Thus the log requirements become very huge if we are trying to create SPINE index for very big sequences (size in millions). However even with the maximum possible log size allowed by SSM, we could create index for sequences of length upto 800,000 only. To improve on this we tried turning off the logging for the records. However the page-level logs were still big enough to allow us build indexes upto length 5 million only.

To handle this problem we have implemented a workaround which makes use the *chained* transactions facility provided by SSM. We commit the transaction after each 5 million characters have been seen for building index and restart the transaction. Since committing and starting a new transaction requires acquiring all the locks again we use a chained transaction instead. The effect of committing transaction is to make SSM reuse the log space used so far and significant amount of log space is regained. In our experiments, we observed that the linear time complexity of building SPINE was not affected due to this.

5.2 Integration with BODHI

Though SPINE is implemented as an independent client-server module, for a user of BODHI the SPINE VAS should appear as an integral part of BODHI. As described earlier, we have implemented the client-side interface of SPINE VAS as a library also. Using this RPC client interface, we have changed the existing implementation of BLAST algorithm in BODHI as explained ahead in details.

5.2.1 Query flow in BODHI

Figure 5.2 shows query flow in BODHI. The ODL and OQL queries are compiled by the corresponding compilers into equivalent C++ code. This C++ code is then compiled into an executable by the C++ compiler. The executable is also linked with the RPC (Remote Procedure Call) libraries of the SHORE server, the GeneStore VAS, the path-dictionary index server (not shown in figure) and the SPINE VAS. When the executable is run, the desired query gets executed.

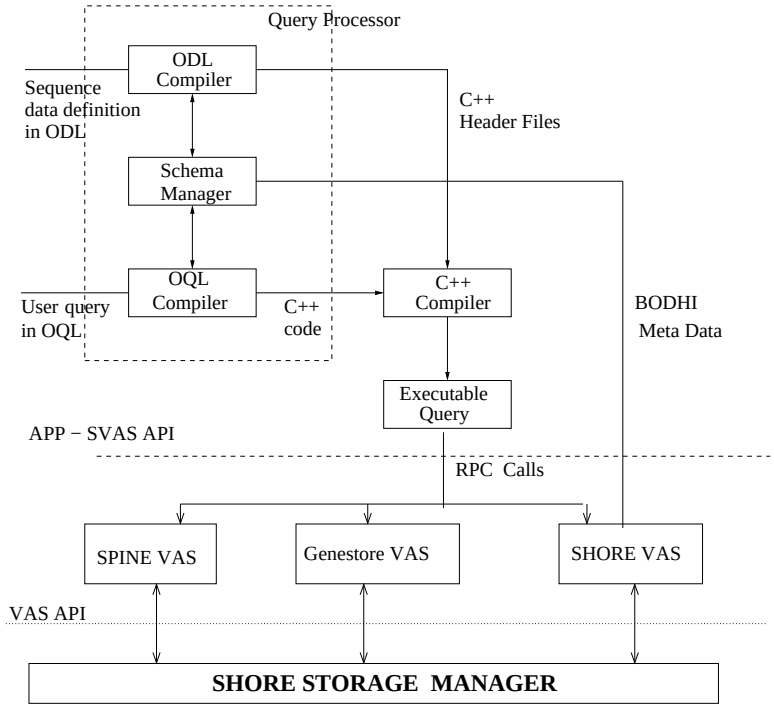


Figure 5.2: **Query Flow in BODHI with SPINE**

5.2.2 Functional Enhancements to the Object Model

Figure 5.3 shows the object model of the plant bio-diversity database currently being used with BODHI. The DNA entity represents a DNA entry for each species.

The current implementation of the GeneStore VAS uses functions defined for each DNA object for its operation. Thus a BLAST query on a DNA sequence is actually executed as a function of that DNA object.

On lines similar to the GeneStore VAS, we have added two more functions to the DNA entity on the object model. **First:** *createIndex*, for creating the SPINE index and **Second:** *findMatches*, a function which finds all maximal subsequence matches of the the DNA sequence with another sequence passed as an argument.

We explain the implementation of these two functions in more details ahead. Please note that the DNA entity would remain the same for any other object model of BODHI, hence our integration of SPINE with BODHI will work for all object models dealing with DNAs.

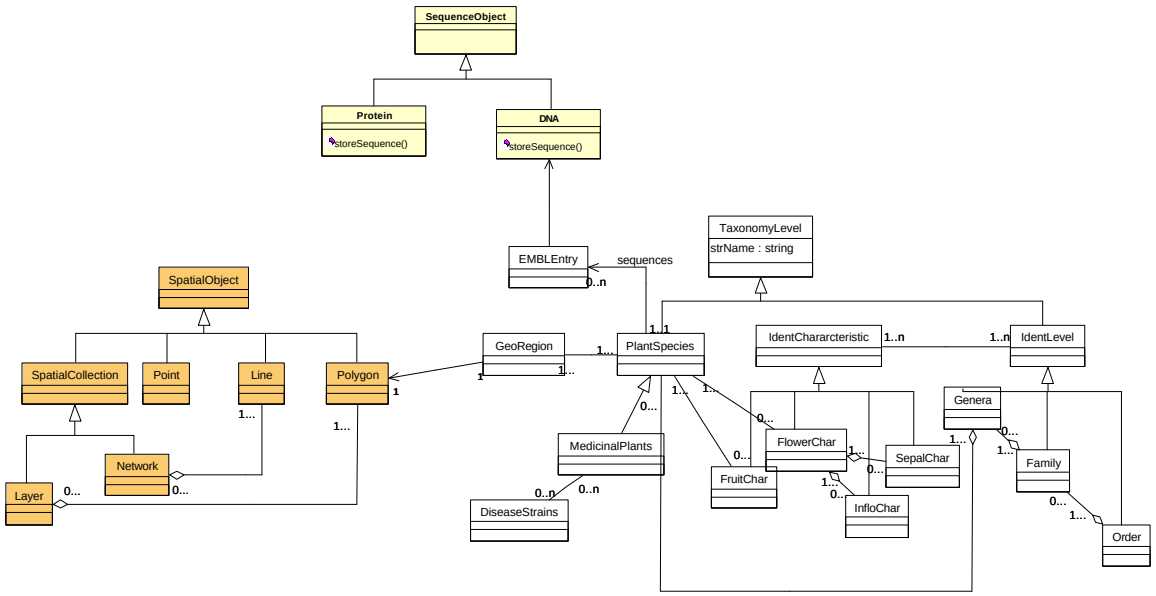


Figure 5.3: **Object model of plant bio-diversity database**

5.2.3 Index Creation

The *createIndex* function builds SPINE index on that DNA's sequence. Using the SPINE client library (explained in Chapter 5.1) it creates a SPINE client. The SPINE client makes a RPC call to the method for creating index on the SPINE server, passing the DNA sequence and its LOID as arguments. The SPINE VAS then builds the index and an entry is made into the *Master* B^+ *tree index* using the DNA's LOID as the key.

5.2.4 Querying the Index

As explained in Chapter 4, suffix trees are used for sequence similarity queries. The difference between the *seed-based* and *index-based* approaches towards sequence similarity query processing lies mainly in the first stage, where two given sequences are matched against each other for finding *seed sequences*. After this first stage the second stage of extending the seed sequences is the same for both approaches.

To integrate SPINE with sequence similarity algorithm of BLAST in BODHI, we have modified the existing implementation of BLAST to use the SPINE index during the first stage of sequence similarity query computation.

The new implementation of BLAST calls the *findMatches* function during its first stage. This function takes the query string as an argument. It creates a SPINE client object which in turn makes a RPC call to the SPINE server with the DNA's LOID and the query string as arguments. The SPINE server

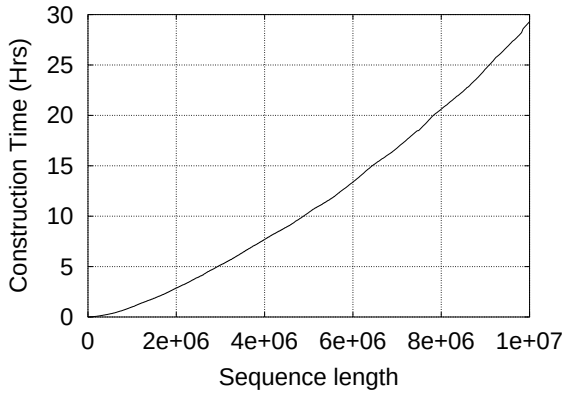


Figure 5.4: **Index Construction Time**

checks first if the SPINE index already exists for that particular DNA (using the LOID key). If the index exists then it is used for finding all the maximal subsequence matches of a particular length of more. All the matches found are returned through the SPINE client to the BLAST function. If the SPINE index for the DNA does not exist then an error is returned and the BLAST function runs the *seed-based* first stage.

RPC implementations impose certain system specific limits on the sizes of data that can be transferred (around a few thousand bytes). Hence our integration of SPINE with BLAST has the limitation to handle sequences of only a few thousands of length.

It should be noted that the SPINE VAS does not need to contact the GeneStore VAS once given the query. This is because SPINE does not require the original sequence for querying.

5.3 Performance Study

The platform used for our experiments was a Pentium-IV 2.4 GHz, 1 GB memory machine running Redhat Linux.

5.3.1 Index construction

Figure 5.4 shows the index construction time for our implementation of SPINE using SHORE. It can be seen that the construction time is almost linear in size of sequence. This is expected since the construction algorithm is linear in time complexity. The construction time in absolute numbers is high, but since index creation cost is occurred only once, it is not much a concern.

5.3.2 Index size

We observed the size occupied by index on the *volume* to be very high at around 85 bytes per character indexed. The reasons that we can site for this are: (a) The sizes of records in LT and RT[1-4] tables are bigger than 12 bytes since we can not manipulate them as arrays of characters (b) SSM adds its own headers to records, increasing their sizes further (c) The occupancy of B^+ trees is not guaranteed to be 100% and may be as low as 50% also. The non-occupied space in each page of B^+ tree is an overhead.

5.3.3 Subsequence match queries

We have compared the performance of our implementation of SPINE using SSM with (a) BLAST implementation in BODHI and (b) BLAST implementation done by NCBI [20] which is a tool commonly used by biologists.

Comparison with BLAST in BODHI

Figure 5.5 shows the performance of sequence similarity queries in BODHI using original BLAST implementation in BODHI and using modified BLAST implementation (using SPINE).

The data used consisted of 100 species with one sequence per specie. The sequences were obtained by taking random substrings from the sequences [ECO], [VIB], [CLS], [CEL] as explained subsequently. The query sequences used were also obtained similarly and were of same length as each data sequence. The memory allocated to SPINE VAS and Genestore VAS was 30MB each.

Figure 5.5 shows the results of the comparison. It is a stack graph and upper layer in each stack shows the extra time taken by original BLAST implementation. It can be clearly seen that use of SPINE index has improved the performance. The performance improvement varies from half to almost one order of magnitude. Also, with increasing sequence length and/or decreasing seed length we get more and more gains by using index. We could not obtain query performance for bigger sequences due to the RPC limitations mentioned earlier.

Comparison with NCBI BLAST:

NCBI [20] BLAST is a sequence similarity query tool commonly used by biologists. The tool runs totally in memory. We have done a comparative study of subsequence match query performance of SPINE and NCBI-BLAST. The sequences used were

- **ECO**: E.Coli earthworm genome of length 3.5 million characters;

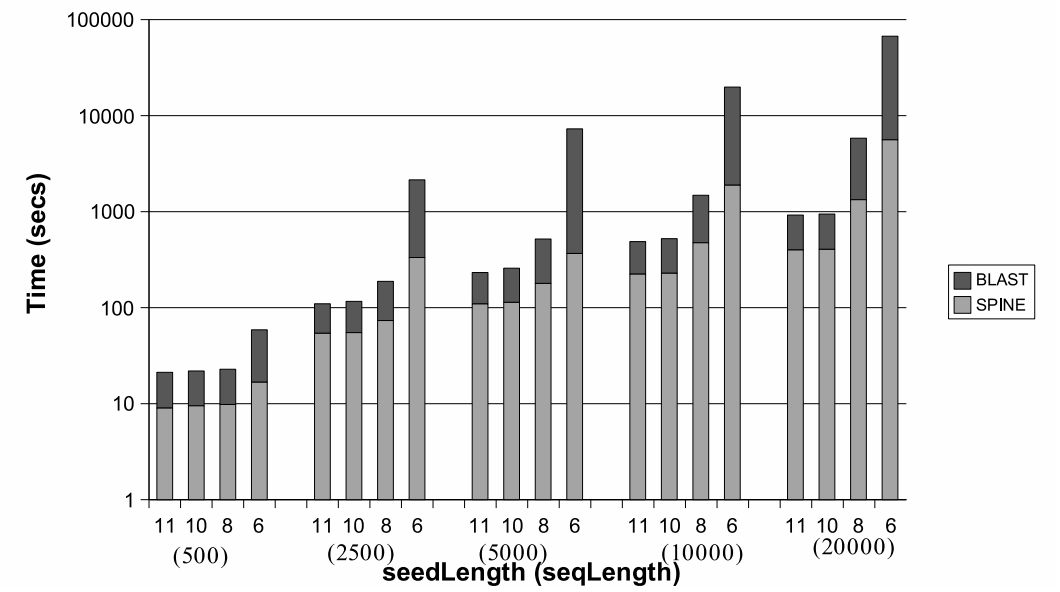


Figure 5.5: **Sequence similarity query performance in BODHI**

- **VIB**: Vibrio bacterial genome of length 1.0 million
- **CLS**: Clostridium-tetani bacterial genome of length 2.7 million
- **CEL**: 4.0 million length prefix of C.Elegans bacterial genome of length 15.5 million characters;

The experiments were conducted using the following methods:

- **BLAST-11**: NCBI-BLAST with seed length of 11
- **BLAST-10**: NCBI-BLAST with seed length of 10
- **MEM-SPINE**: Neelapala et.al's implementation of SPINE in memory
- **SHORE-SPINE**: Our implementation of SPINE using SHORE with 30 MB memory and 300 MB memory. The size of 300 MB was chosen to ensure that any of the indexes fits in memory.

Figure 5.6 shows the results of our experiments. The timings reported for MEM-SPINE and SHORE-SPINE are only for subsequence matching and do not include the time for extending matched subsequences (for finding alignment). This is because (a) MEM-SPINE's implementation from Neelapala et.al. does not have the code for doing extension and (b) SHORE-SPINE's performance was studied using the command-line interface since the client-interface integrated with BODHI suffers from sequence length limitations discussed earlier.

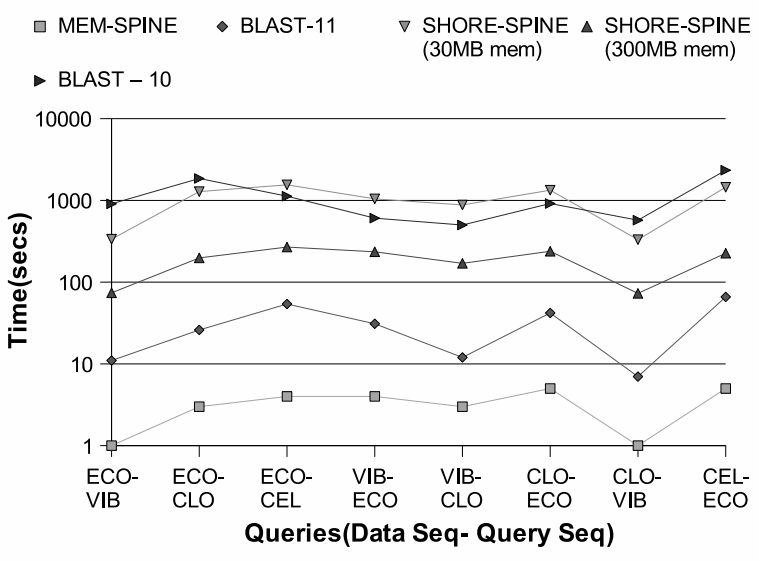


Figure 5.6: **Subsequence Match Query Times**

We observe that BLAST with seed length of 11, which is the heuristic value, performs much better compared to SHORE-SPINE. Though MEM-SPINE appears to perform better than BLAST, it actually is performing as good as BLAST-11 considering the time BLAST-11 takes for extending matches.

BLAST is a heuristic based algorithm and performs best for seed length 11, however while doing so it may miss out some matches of smaller length which when extended may result in good alignments. A solution to this is to run BLAST with smaller value of seed sequence length, e.g. 10. However Figure 5.6 shows that, BLAST-10 performs much worse than BLAST-11, worse than SHORE-SPINE with 300 MB memory and almost same as SHORE-SPINE with 30 MB memory. We have observed that BLAST performs even worse when the seed lengths are decreased further. On the contrary either MEM-SPINE's or SHORE-SPINE's performance does not depend on the seed length. The seed length only determines which matches are returned. Thus SPINE offers accurate results while performing as good as BLAST, when seed lengths are small.

It may appear strange that SHORE-SPINE with 300 MB of memory performs much slower than MEM-SPINE, when 300 MB is enough for any of the indexes to fit in memory. We attribute this to more memory accesses made by SHORE-SPINE due to the following reasons: (a) The size of an index with SHORE-SPINE is almost one order of magnitude more than with MEM-SPINE. MEM-SPINE requires 12 bytes per character while SHORE-SPINE requires around 85 bytes. Thus we access more memory with SHORE-SPINE. (b) Access to a record with MEM-SPINE is directly made by using array indexing, whereas with SHORE-SPINE each record access requires traversing internal nodes of a B^+

tree. (c) The page tables maintained by storage manager of SHORE are accessed frequently and with 300 MB memory, page table size is also big enough. (d) Cache misses would be more with 300 MB memory than with 30 MB memory. (e) The program size is much bigger implying more running times also. We have verified these reasons by doing profiling of our program using the *gprof* tool available in Linux. Thus the performance of SPINE is found to be affected due to the environment (SHORE) in which it is implemented.

Chapter 6

Implementation of Graphical User Interface Framework

BODHI supports full OQL/ODL [6] querying interface on the server side [3, 4]. However for the bio-diversity researchers, who are the end users of the BODHI system, learning a querying language is a cumbersome task. Typically they do not have any experience of using such languages and are reluctant to learn it. Thus, in order to be really useful to its end users, BODHI requires a graphical user interface for expressing the queries.

BODHI currently hosts a plant bio-diversity database, the schema of which has been shown in Figure 5.3. Our aim was to build a user interface to allow querying this particular schema comprehensively.

We enlist below the properties and functionality desired from the user interface.

1. **Intuitive and simple:** The user interface should be intuitively clear to the end user. The layout and the fields provided for querying should be simple enough to be understood by just looking at them. The semantics associated with each querying facility should be intuitively clear from their appearance.

2. **Automated and correct:** The queries expressed by users should be automatically and correctly converted into OQL. The OQL queries should be run on the server, and the results returned to the user in a transparent manner.

3. **Compact:** A screenful of information [23] is what the human eye can grasp easily. By an intelligent placement of query fields and conscious attempt at reducing the number of querying fields,

the interface should be kept compact. Also, one should be able to express multi-domain as well as uni-domain queries using the same interface.

4. **Utility:** While being compact the interface should also allow the user to form as many queries as possible with as less effort as possible. The OQL allows users to form any number of queries, given the database schema. The query interface should aim at going as close as possible to this ideal situation.

5. **Accessibility:** Accessibility over the internet is a highly desired feature. This allows the researchers all over the world to use and extract information from the database. Also it makes the user interface system platform independent, as most of the popular browsers run on most of the platforms.

6. **Customizable output :** The output of user queries should be presented in a customizable manner. The users should be able to apply their own formatting and view the results the way they want.

7. **Data exchange:** The output should also be in a format that can satisfy future data exchange requirements that may arise.

8. **Convenient spatial querying:** Expressing the spatial queries is a pain for the users if they are required to enter the latitude-longitude parameters by hand. To simplify this kind of querying, the querying interface should provide graphical ways of specifying the spatial components of queries. Facilities like zooming, panning and rubber-band selection would facilitate the task greatly.

9. **Easy navigation:** It should be possible to navigate easily throughout the query interface. Quicker ways of reaching the desired part of the interface should be provided.

10. **Help:** Apart from being intuitive and simple, an online help facility would greatly enhance the understandability of the interface for the end users.

We have implemented a graphical user interface for BODHI which satisfies all of the above requirements to a large extent. We now explain the architecture of the user interface framework,the technologies involved in the implementation, the capabilities of the interface and provide few snapshots of the interface.

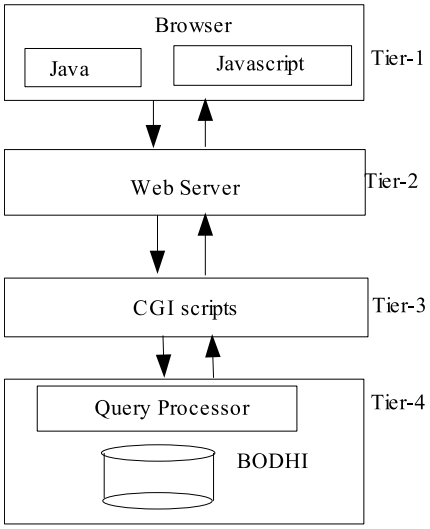


Figure 6.1: **Graphical User Interface Framework Architecture**

6.1 Architecture of the Graphical User Interface Framework

Figure 6.1 shows the architecture of the Graphical User Interface Framework of BODHI. It is a typical 4-tier architecture which consist of a backend database system (BODHI), a webserver serving HTML webpages (apache), CGI-scripts for the application logic (perl) and the web browser at the user end.

6.2 Technology Choices

We chose the following technologies for implementing the user interface of BODHI:

- **HTML form for query input:** The requirement of having BODHI accessible over the internet translates into having web based graphical user interface. We have implemented the query input interface as an HTML form as shown in Figure 6.2.

- **Javascript for query conversion to OQL:** Javascript is the most popular language for client side programming with webpages. It is an object-based programming language with C like syntax and supported by most of the browsers. All the popular web browsers like Internet Explorer, Netscape, Opera support it.

- **Java Applet for spatial querying:** We have used the *GIS4* applet available from [12] for spatial data viewing and querying. The original applet taken from [12] allowed only for viewing of spatial

data. We have modified the applet code to also allow rubber-band selection of queries, as shown in Figure 6.3.

- **Java-Javascript interaction:** Interaction between Java and Javascript is required for exchange of the spatial query information. The Java applet does the job of rubber-band selection of query and passes on the information to Javascript code for converting it into OQL. To perform this we used the LiveConnect Java package available from Netscape [15].

- **CGI-Perl scripts for server side processing:** CGI (Common Gateway Interface) is a standard for external gateway programs to interface with information servers such as HTTP servers. Perl has became a popular language for doing CGI programming due to the large set of libraries available with it. The queries from the user interface are processed by the CGI-perl scripts on the server side. They do the job of submitting the queries to the translator and returning the results from BODHI system back to users.

- **XML for output:** XML [8] has become the de-facto standard for data exchange over the web. Hence it is highly desirable for the results to be presented using XML. This allows the users to apply their own formatting to the results (using XSL stylesheets) for viewing, thus allowing customization of result-viewing. Use of XML in publishing results also makes provisions for future data exchanges if required.

6.3 Capabilities of the Query Interface

Figure 6.2 shows the query input form. The current implementation of the querying interface is capable of specifying multi-domain as well as uni-domain queries involving upto four species.

In particular for each of the three domains, viz. taxonomy, spatial and sequence, it has the following capabilities:

6.3.1 Taxonomy:

The left half of the form allows us to query the taxonomy and phenotype (concerning the characteristics of species) information. One can specify:

- Equality predicates on species name, genera, family and order.

- Display predicates on all the attributes.

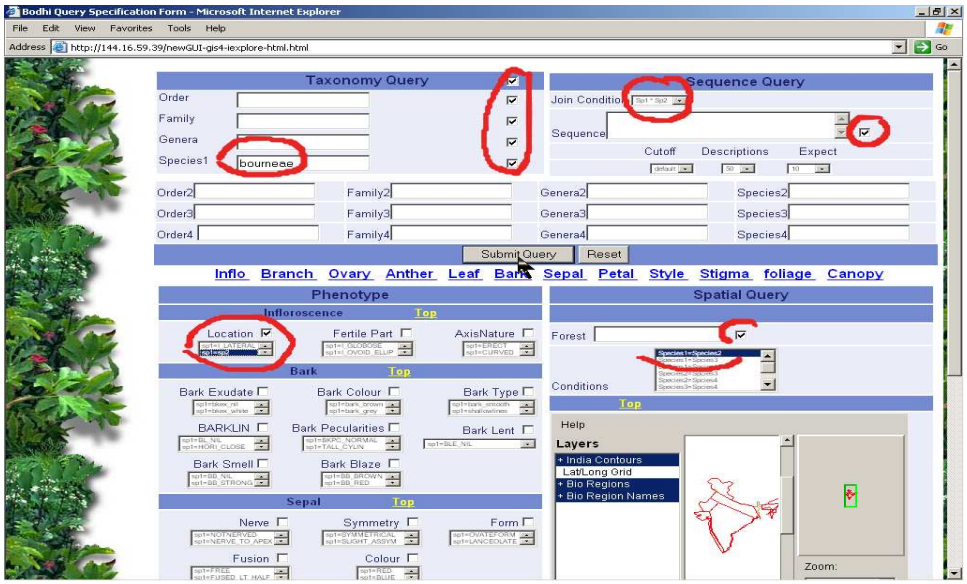


Figure 6.2: **Query Input Form (Markings highlight the sample query in Chapter** 1**)**

- Equality predicates on each of the characteristics of the species.
- Join predicates on each of the characteristics. Multiple join predicates can also be specified on the same characteristic.

6.3.2 Sequence:

The upper right part of the form allows us to query the sequence information. It can specify:

- Display predicate on the sequence.
- Blast query with *specie vs. specie* or *specie vs. sequence* join predicate.
- Predicates for blast query like *cutoff score, number of outputs*.

6.3.3 Spatial:

The lower right part of the form allows us to query the spatial information. It can specify:

- Equality predicates on the names of spatial attributes, viz. *forests, georegions and rivers*. Only the *forest* part can be seen in Figure 6.2.
- Display predicates on all the spatial attributes.
- Join predicates like *overlaps, intersects, contains, equals* across the spatial attributes of species.

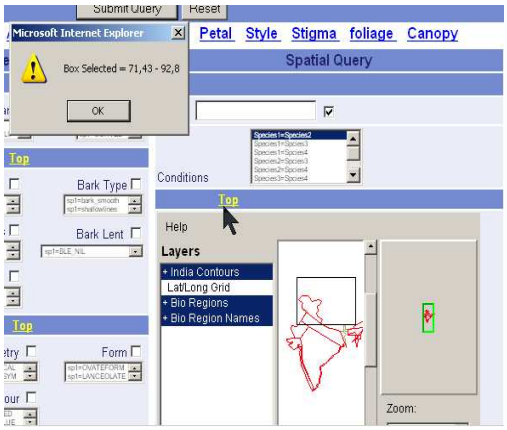


Figure 6.3: **Rubber-band selection of Spatial Query**

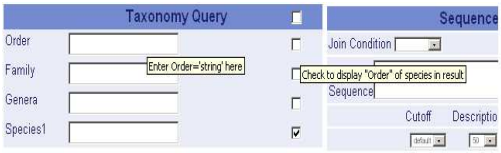


Figure 6.4: **Help Tags**

- The spatial applet (GIS4 [12]) allows zooming, panning, switching on-off the information layers for better visualization of the query map. One can specify a rubber-band rectangle-selection query using this applet.

6.4 Snapshots of the Interface

We already have discussed the query input form shown in Figure 6.2. It can be seen that it is as compact as one and half screens. It also shows how one can specify the sample query specified in Chapter 1 using the query form.

A comparison of Figure 5.3 and Figure 6.2 shows that the query interface comprehensively covers the object model for the plant database currently being hosted with BODHI. Figure 6.3 shows specification of a region by using rubber-band selection on the map seen on applet. The *help tags* associated with each query input field appear as shown in Figure 6.4. Figure 6.5 shows the zooming operation being performed using the GIS applet.

The results returned from server are in XML which can be seen in Figure 6.6. Finally the results are displayed in HTML as shown in Figure 6.7. We have used a XSL [9] style sheet to specify the conversion from the XML results to HTML. The hyperlink shown in Figure 6.7, when clicked, displays the spatial

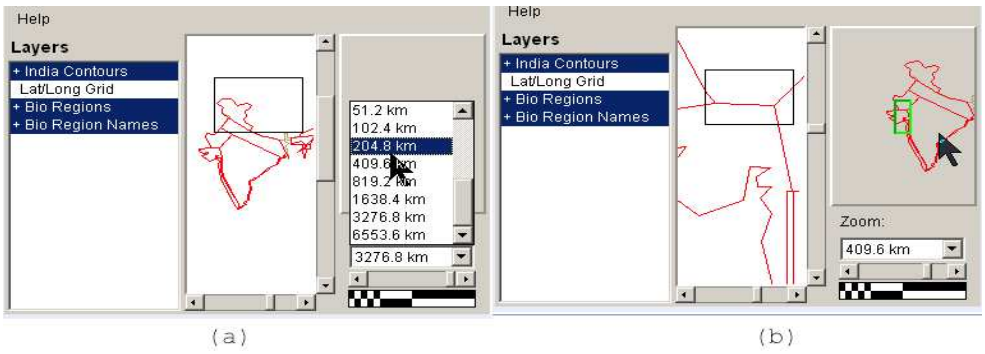


Figure 6.5: **Zooming: (a) Operation (b) Result**

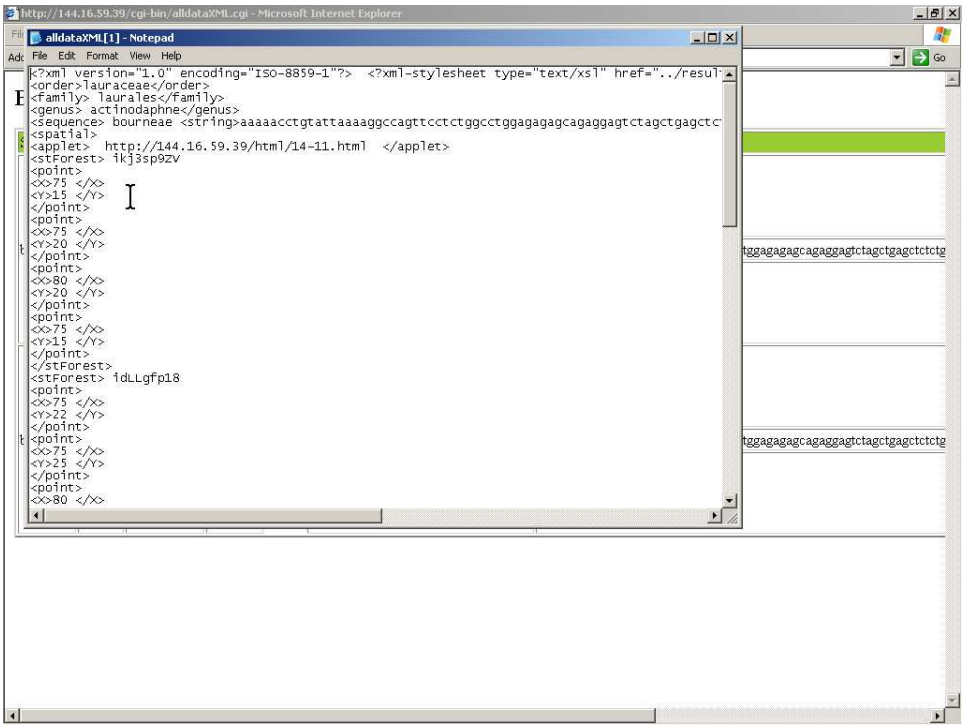


Figure 6.6: **Results are in XML(Displayed in *Notepad* as the source of result) (Using View-Source menu in browser)**

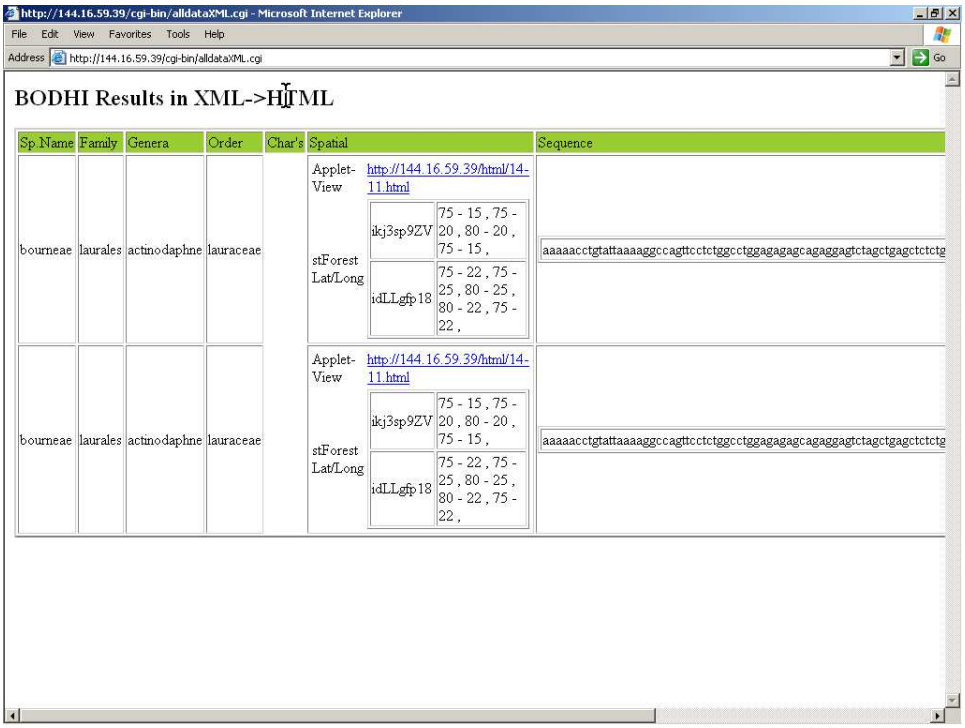


Figure 6.7: **XML Results shown as HTML**

results using the GIS4 applet (not shown in Figure).

Chapter 7

Conclusions and Future Work

BODHI now comes equipped with the SPINE genomic index, which has helped improve the performance of sequence similarity queries. The improvements are more for smaller seed lengths and larger sequences.

The web based query interface has simplified the task of specifying queries over the plant biodiversity database to a great extent and the system is useful in a convenient fashion to the plant biodiversity researchers. We have demonstrated the user interface to researches at Center for Ecological Sciences at Indian Institute of Science and they have opined the interface to be useful.

Future work can concentrate on defining cost model for SPINE and integrating it with query optimizer. Also, efforts can be directed towards using the SPINE index to support various sequence operations like finding all occurrences of a string, maximal matching substrings, etc. independently.

Bibliography

[1] S. Altschul, W. Gish, W. Miller, E.W. Myers, and D. Lipman. A Basic Local Alignment Search Tool. *Journal of Molecular Biology*, (215), 1990.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R^*-Tree : An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[3] S. Bedathur, A. Kadlag, and J. Haritsa. BODHI: A database habitat for bio-diversity information (demo). In *ACM SIGMOD Intl. Conf. on Management of Data, Paris, France*, 2004.

[4] S. Bedathur, J. Haritsa, and U. Sen. The Building of BODHI, a Bio-diversity Database System. *Information Systems*, 28(4), 2003.

[5] M. J. Carey et.al. Shoring up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 1994.

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

[7] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27:2369–2376, 1999.

[8] Extensible Markup Language. http://www.w3.org/TR/REC-xml.

[9] The Extensible Stylesheet Language Family. www.w3.org/Style/XSL/.

[10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[11] L. Fegaras. An Experimental Optimizer for OQL. Technical Report TR-CSE-97-007, University of Texas at Arlington, 1997.

[12] Global Information Systems. http://elib.cs.berkeley.edu/gis.

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

[14] A. Gutteman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Intl. Conf. on Management of Data*, 1984.

[15] Java-Javascript Interaction. http://wp.netscape.com/ eng/mozilla/3.0/handbook/javascript/packages.htm.

[16] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Databases*, 1994.

[17] W. Lee and D. L. Lee. Path Dictionary: A New Access Method for Query Processing in Object-oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(3), May 1998.

[18] T. A. Mück and M. L. Polaschek. A Configurable Type Hierarchy Index for OODB. *VLDB Journal*, 6(4), 1997.

[19] B. Ma, J. Tromp, and M. Li. Pattern hunter: Faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[20] National Center for Biotechnology Information. http://www.ncbi.nih.gov/BLAST/.

[21] N. Neelapala, R. Mittal, and J. Haritsa. SPINE: Putting Backbone into String Indexing. In *Proceedings of the IEEE Conference on Data Engineering*, 2004.

[22] Shore documentation. http://www.cs.wisc.edu/shore.

[23] B. Shneiderman. *Designing the User Interface*. Addison-Wesley Publishing Company, 1998.

[24] T.A. Smith and M.S. Waterman. Identification of Common Molecular Subsequences *Journal of Molecular Biology*, 284, 1981.