BODHI: A Database Engine for Biological Applications

A Thesis
Submitted for the Degree of
Doctor of Philosophy
in the Faculty of Engineering

By
Srikanta B. J.

Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

April 2006

“In fond memory of my Mother – my first teacher”

Abstract

Biodiversity research generates and uses a variety of data spanning across diverse domains, including taxonomy, geo-spatial and genetic domains, which vary greatly in their structural features and complexities, query processing costs and storage volumes. In this thesis, we present BODHI, a database engine that seamlessly integrates these diverse types of data, spanning the range from molecular to organism-level information. BODHI is a native object-oriented database system built around a publically available micro-kernel and extensible query processor, and offers a functionally comprehensive query interface. The server is partitioned into three service modules: object, spatial and sequence, each handling the associated data domain and providing appropriate storage, modeling interfaces, and evaluation algorithms for predicates over the corresponding data types. To accelerate query response times, a variety of specialized access structures are included for each domain. Our experiments with complex cross-domain queries over a representative biodiversity dataset indicate efficient evaluation even on off-the-shelf standard hardware.

BODHI features suffix-tree indexes for expeditious processing of sequence similarity predicates. These indexes are well-known to be not easily amenable to persistent implementation and usage, since their traversal patterns induce severe disk thrashing. To minimize the impact of this problem, we present a suite of optimizations, including: TOP-Q, a novel low-overhead buffer management policy that takes into account the probabilistic behavior of traversals during suffix-tree construction; and STELLAR, a layout-reorganization algorithm that minimizes the cost of suffix-link traversals. Through experimentation on a variety of real genomic and proteomic sequences, we show that the combined effect of these optimizations results in substantially improved index construction and search times.

In summary, this thesis presents the architecture and implementation of a holistic and efficient database engine targeted towards helping biodiversity scientists swiftly advance the state-of-the-art in their research.

Publications

1. "Design and Implementation of a Biodiversity Information Management System"
   Srikanta Bedathur and Jayant Haritsa
   Proc. of 10th Intl. Conf. on Management of Data (COMAD)
   Pune, India, December 2000, pgs. 121-134.
   (Received the Best Paper award)
2. "The Building of BODHI, a Biodiversity Database System"
   Srikanta Bedathur, Jayant Haritsa and Uday Sen
   Information Systems, Elsevier Science Publishers,
   vol. 28, no. 4, June 2003, pgs. 347-367.
   (Special issue on *Bioinformatics and Biological Data Management*, Editors: M. Zaki and J. Wang)
3. "Engineering a Fast Online Persistent Suffix Tree Construction"
   Srikanta Bedathur and Jayant Haritsa
   Proc. of 20th IEEE Intl. Conf. on Data Engineering (ICDE),
   Boston, USA, March 2004, pgs. 720-731.
4. "BODHI: A Database Habitat for Bio-diversity Information"
   Srikanta Bedathur, Abhijit Kadlag and Jayant Haritsa
   Proc. of ACM SIGMOD Intl. Conf. on Management of Data,
   Paris, France, June 2004, pgs. 953-954.
5. "Search Optimized Persistent Suffix-tree Storage"
   Srikanta Bedathur and Jayant Haritsa
   Proc. of 12th IEEE Intl. Conf. on High Performance Computing (HiPC),
   Goa, India, December 2005.
   published as
   High Performance Computing - HiPC 2005, Springer, Lecture Notes in Computer Science (LNCS) 3769,
   eds. D. Bader, M. Parashar, V. Sridhar and V. Prasanna, pgs. 29-39.

Acknowledgements

In the long journey that led to the creation of the work presented in this thesis, I have been guided, helped and supported by many. This is my opportunity to thank them for everything they have done for me.

First of all, I am deeply indebted to my advisor Prof. Jayant Haritsa, for teaching me the principles of good research. His dedication and enthusiasm towards research have been tremendously inspiring (and in some cases very daunting!!).

Special thanks are due to Prof. Ramesh Hariharan, who helped me in the initial stages of my research with immense patience and provided valuable insights which have helped me throughout. I will be forever grateful for his help.

I would like to thank Prof. N. Balakrishnan, Prof. S.K. Nandy, Prof. R. Govindarajan and Prof. Matthew Jacob of SERC for their timely advice and kind help in many occasions. I am also grateful to all the administrative staff of SERC – Shekhar, Sashi, Sarala, Govindaswamy, Triveni, Gopakumar, Mallika, Shivanna and Manjari. I am indebted to CMC staff led by Raju and Ananth who were always there to help me deal with hardware and software administration issues, and enabled me to do my work in time.

In addition to my research, one thing that I really enjoyed in the lab was the company of my lab-mates – Vikram, Kumaran, Suresha, and, of course, the "BODHI gang" of Rajarao, Satheesh, Madhav, Uday and Abhijit. Together, we managed to turn the lab into a "home"!

During my stay at Indian Institute of Science, I have made a number of friends who have made my life at IISc extremely enjoyable as well as intellectually stimulating. I thank Pramod, KVS, GVSK, KVM, MBK, Anurag, Manjusha, Prithu, Siddhartha, Aditya and

Ravindra for their company at the “famous” Coffee Board & Tea Kiosks of IISc, IISc Gymkhana’s Hockey field and Badminton court, and long walks into the Jubilee Park to watch snakes, ants and for just lazing about!

There is one special person I met at IISc who deserves special thanks for being a dependable friend, an enjoyable companion, a fierce critique, and my life partner, Maya. Without her I am not sure I would have come out of my PhD with as much happiness as I have now. She celebrated my successes with more thrill than me, encouraged me not to buckle down when I thought the whole world was against me, and helped me refine many of my ideas, despite having her own research work to deal with!

I thank my Father-in-law who provided guidance in many times of need, and mother-in-law for wonderful food she used to send us!! And ofcourse, Madhava who sometimes psyched us with his love of computers!!

I cannot put into words the gratitude I owe my parents and my loving brother. Without them I wouldn’t have been here writing my PhD thesis! They encouraged me to excel in whatever I do and always had unwavering faith in me. They supported my decision to quit my job and take up my PhD, despite having a number of difficulties, and never made me regret my decision. I don’t think I could have asked for better Mom, Dad and Brother than the ones I have. It is very unfortunate that my Mom is not here to watch me write this thesis – I can only imagine how proud she would have felt to have me here. I miss you Amma. I also thank my sister-in-law, Gayathri, for taking over the responsibilities at the home front without any complaints! Arrival of Vishnu, the cute-little-boy :-) was one of the best parts of our life! I had never seen a more smiley baby than him!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Biology can be studied at a variety of scales, ranging from molecular structures in individual cells to the macro-level interactions at the ecological level. The humongous variety of the natural patterns thus formed, and the variety of life constitutes the Biodiversity of the Earth [109]. This diversity is often understood in terms of the wide variety of plants, animals and microorganisms that occur in nature, and in terms of the associated phylogenetic and ecological relationships between them. Biodiversity is the inherent wealth of a region, as most economies are directly or indirectly dependent on the products of biodiversity, and more importantly, on the life supporting conditions that it provides. Thus, understanding of biodiversity is vital from scientific, educational, commercial and medicinal viewpoints.

Table 1.1 summarizes the estimates on the conservation status of global biodiversity

Estimated number of species	3 – 100 million
Number of species identified	1.75 million
Rate of loss of biodiversity	$\approx 10^4$ species per year
Rate of discovery of new species	$\approx 10^4$ species per year

Table 1.1: Status of Biodiversity Conservation

from World Conservation Monitoring Center (WCMC) [133] – one of the leading global organizations building infrastructure for world-wide free exchange of biodiversity data. These figures indicate that the rate of loss of this precious biodiversity wealth is, at best, on par with our ability to just discover new species, let alone the ability to effectively collect, curate and disseminate their information. Hence it is extremely important to develop effective conservation policies and speed up the species discovery process through an extensive use of computerization and other advances in technology. There are a number of national-level and international efforts underway to tackle this challenge.

An urgent need in this regard is to design and deploy *information systems* that help the management of a wide range of data associated with biodiversity studies [109]. A biodiversity information management system has to enable effective and efficient ways to model, store and manage the taxonomic information of species, their phenetic characteristics, the phylogenetic and other environmental relationships with other organisms in the biota (the flora and fauna of a region), and the geographical information about the endemic habitats of species.

With recent advances in rapid genome sequencing of organisms, there is an added dimension to the study of biodiversity – at the level of micro-level genome relationships. One can try to infer associations between macro-level characteristics of species based on their genome-level information and vice versa. For example, if research is being undertaken into the DNA sequence of an organism to identify genes responsible for a specific characteristic, then one can utilize the information available for related organisms, which can be located through the associated taxonomic information. Similarly, accurate phylogenetic tree construction and verification – an important task in the understanding of biodiversity – benefits immensely from the interaction of taxonomic information and genetic data analysis.

Keeping these developments in view, the study of biodiversity has recently been redefined by WCMC to be an integrated study of species diversity, habitat diversity and genetic diversity. This integrated view has lead to extensions in biodiversity studies, which traditionally had focused on taxonomic and habitat information, to include genome level information. In the process, new branches of biological science such as *Molecular Biodiversity* [84] and *Ecological Genomics* [124] have been formed. This integrated study requires combined analyses of large collections of data from disparate domains, and managing such a wide variety of data to enable efficient discovery of inter-relationships is a serious challenge towards achieving swift advancements in biodiversity research.

1.2 Current Solutions

Storing and managing the enormous amounts of data generated from the biodiversity research efforts, and analyzing this data to extract nuggets of information, requires the extensive use of computers. There are tools for individually managing data from each of the domains involved in biodiversity studies – for instance [87], at our institution, species data is currently maintained in MS-Excel worksheets on individual computers, the ecosystem information is managed through the use of spatial data management tools such as ArcView [43], and the web-based services from global organizations such as NCBI (National Center for Biotechnology Information) and EMBL (European Molecular Biology Laboratory) are relied on for querying genetic data.

However, the presence of such a bag of independent tools is no longer adequate, from both functionality and efficiency perspectives, in the new age biodiversity research which places increasing emphasis on *simultaneously* studying the micro-level and macro-level relationships between biological entities. To illustrate this point, consider the following example query, which is of interest to modern evolutionary biologists – similar research questions are to be found in the ecological literature, e.g., in [82]:

Query 1 *Retrieve the names of all plant species that have common inflorescence characteristics, share a part of their habitats, and have a high chromosomal DNA sequence similarity with Michelia-champa*[1].

Answering the above query requires the ability to perform integrated searches over taxonomy hierarchies (*"common inflorescence characteristics"*), recorded spatial distri-

[1]A fragrant medicinal plant endemic to India and Nepal, also called Michelia champaka.

bution of species (*"share a part of their habitat"*), and the genome sequence databases (*"high chromosomal DNA sequence similarity"*). Unfortunately, however, due to the lack of holistic database systems, biologists are usually forced to split the query into component queries, each of which can be processed separately using independent tools and services. Further, the results from these individual tools have to be combined either *manually* or through the use of a *customized* tool.

For example, a typical "experience story" for answering the above query, as gathered from domain experts at our institution [87], would be:

1. Locate all plant species that have inflorescence characteristic common with that of Michelia-champa, by performing a join over the taxonomy database, stored in a PC-based relational database.

2. Access the habitat data, stored in *ArcView* [43], a popular spatial database product, to find the species that have shared habitat with Michelia-champa, by performing a spatial join. Then, compute the intersection between the set of species obtained in the earlier step, and the newly-derived set of species, in order to prune species that do not share common habitat with Michelia-champa.

3. From the output of Step 2, identify the names of the plant species of interest, and then perform repeated BLAST [1] searches over (a subset of) *NCBI GenBank* [41] DNA sequence database to identify the sequences (and, thereby the species), that have an MSP (Maximal Segment Pair) score more than a cutoff value. Note that this final score-based pruning has to be performed externally by the researcher.

Long procedures, such as the above, for answering standard queries are not only cumbersome but can also lead to delays in understanding various micro-level and macro-level biodiversity patterns. Worse, the patterns may not be found at all due to limited human capabilities – an example of this problem was reported in the molecular biology study [112], where comparison of sequences "by hand" missed out some of the significant alignments thereby leading to erroneous conclusions about the functional similarity of the proteins examined in the study.

Furthermore, the discovery of a new species involves considerable time to be spent by the biodiversity researcher on *field*, collecting characteristics and other important traits of the species under study. During such field-trips it is of immense help to be able to access a remote database, to query across multiple domains and pull out data about related species, possibly through a hand-held device. With the bag-of-tools as described above, it is extremely frustrating to perform such tasks.

1.3 Holistic Databases for Biodiversity Research

Based on the above discussion, there appears to be a clear need for building an *integrated* database system that can be productively used by the biodiversity community. However, building such a system is highly challenging because the data associated with each of the subdomains of biodiversity studies – namely, *taxonomy*, *spatial* and *sequence* – vary greatly in the following characteristics:

1. Structural complexity,
2. Query processing cost, and
3. Storage volume.

For example, while the taxonomy information of species has complex hierarchical structure, spatial data associated with ecosystems are inherently voluminous and the spatial operators are computationally expensive. On the other hand, genomic sequence processing is based on specialized pattern recognition and similarity identification algorithms over DNA or Protein sequences of the species.

As a result of the limited support offered for the resulting complex data processing requirements in biology, current database systems have been relegated to play the role of *backup stores*, with much of the processing being done outside the DBMS by Unix-based tools. This is primarily due to the lack of holistic database systems that provide a wide range of functionalities as well as the performance demanded by these applications.

Thus, in order to cater to the new but critical breed of modern biodiversity research, data management systems have to address a host of novel design and implementation challenges which were addressed in isolation in the past. The database community has also realized the exciting opportunities for novel data management techniques in this domain – in fact, biodiversity was featured as the theme topic at the 26th International Conference on Very Large Data Bases (VLDB), 2000 [73].

1.4 Design of BODHI

Biodiversity databases are typically very large, comprising objects of different types and inter-relationships to form deeply nested hierarchies. Queries that span these hierarchies need to perform multiple joins and, in many cases, these involve spatial joins or sequence similarity predicates, that are computationally more expensive to evaluate.

Motivated by the lacuna of a holistic database solution catering to the needs of biodiversity researchers, in this thesis we propose the design of **BODHI** (**B**iodiversity **O**bject **D**atabase arc**HI**tecture),[2] a native object-oriented database system that *seamlessly* integrates multiple types of data occurring in biodiversity studies. It is built around a publically available storage manager kernel, and offers a functionally comprehensive query language. The BODHI system expresses the sample query presented earlier in Query 1, which spans multiple data domains of biodiversity research, using an extended OQL 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.

A modular schematic of BODHI is shown in Figure 1.2. The SHORE storage manager [19] at the base provides the fundamental needs of a database server such as device and storage management, transaction processing, logging and recovery management. The functional core of the system is built over this storage manager, and consists of three application specific modules, which supply the object, spatial and genomic services. The

[2]Gautama Buddha gained enlightenment under the Bodhi tree.

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

Figure 1.1: Expressing Multi-domain Query in BODHI

query processor and optimizer of BODHI, which is based on the λ-DB rule-based object query processor [38], interfaces with these functional modules and performs query processing and produces efficient execution plans using the metadata exported by the modules. The base functionality provided by λ-DB has been significantly enhanced to take into account the availability of new data modeling and query language features, and a variety of access-structures in the system. BODHI supports full OQL/ODL query and data modeling interface for creation of new database schemas, data manipulation and querying. Finally, the client interface framework and XML publishing engine form the external interface to BODHI, enabling biologists to construct complex queries through a form-based interface as well as to graphically visualize the results.

Each of the service modules in BODHI provide appropriate storage, a modeling interface, and evaluation algorithms for predicates over the corresponding data types.

Object Services. In querying over biodiversity 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* [83]) and aggregation hierarchies (*Path-dictionary index* [74]) are included in this module.

Spatial Services. This module provides a spatial type system for modeling of spatial data associated with biological information. Various geometric operators such as

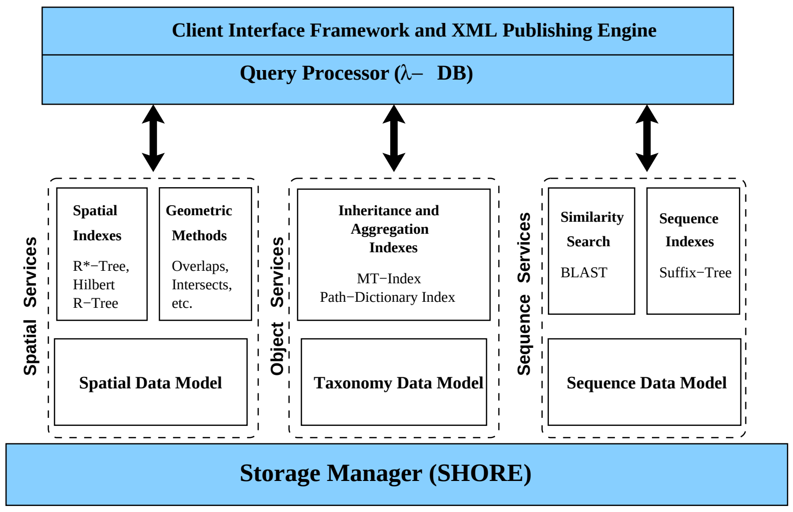


Figure 1.2: BODHI Architecture

overlap, *adjacent*, *area*, etc., are implemented over this type system. The module incorporates R*-Tree [7] and Hilbert R-Tree [67] indexing to speed up these otherwise expensive operators.

Sequence Services. This module provides efficient storage of sequence data associated with species and a suite of operations over it. It implements popular alignment-based sequence similarity algorithms of BLAST [1] and Smith-Waterman dynamic programming [113]. To alleviate the response time bottleneck due to brute force scan adopted by these algorithms, this module of BODHI provides persistent version of *suffix-trees* [8], the ubiquitous main-memory sequence indexing structure. The persistent suffix-tree index provides an accurate indexing solution for a number of biological sequence querying applications. We are not aware of any other database system that incorporates persistent suffix-trees as a first class sequence indexing strategy.

The BODHI system is fully operational and the source code is available under GNU Public License for further customization and enhancements. The system has been demonstrated at the ACM SIGMOD International Conference on Management of Data, Paris, France, 2004, and in a number of national-level ecological workshops in India, and has been uploaded by the San Diego Supercomputer Center (SDSC), USA.

1.5 Biological Sequence Indexing in BODHI

A major research challenge that we have tackled in the BODHI system is the issue of providing high-performance sequence similarity searching. Despite the extensive utility of sequence similarity searching, there has been very little direct database support for such operations. In his keynote address at SIGMOD'2001, Gene Myers pointed out that this has resulted in a missed opportunity in tighter integration of databases in the bioinformatics research.

By incorporating persistent suffix-trees in BODHI, we reduce the overheads incurred due to the brute-force scanning of sequence databases, adopted by BLAST and other popular biological sequence similarity search algorithms. The integration of suffix-trees involved addressing of two issues associated with persistent suffix-trees: (i) efficient construction of persistent suffix-trees, and (ii) their storage management to improve query throughput.

1.5.1 Efficient Construction of Persistent Suffix-Tree Indexes

The suffix-tree of a sequence is the defacto index structure used in numerous biological applications for accelerating queries over sequences [29, 53, 81]. The main attraction of suffix-trees lies in their linear time and space construction complexity, and in their use in a number of sequence querying situations, with almost any of the similarity metrics employed by biologists to compare DNA or Protein sequences.

Although the utility of suffix-trees is well known, their viability is limited to small-size datasets due to their large space requirements – the best implementation of suffix-trees imposes more than 12 bytes overhead for every symbol indexed. This is in marked contrast to traditional database index structures, which are typically a fraction of the overall database size. The obvious solution of extending the suffix-trees onto disk is seriously hampered by the *disk-unfriendly* nature of suffix-tree construction algorithms [80, 126, 130]. Hence, the popular belief that suffix-tree indexing is not practical over large datasets [89].

In this thesis, we present techniques to significantly improve the performance of traditional suffix-tree construction algorithms such as [80, 126], in the context of persistent suffix-trees. Specifically, we consider the impact of the *buffering policy* employed during construction and the *physical representation* of suffix-tree nodes and make the following contributions:

First, we present a novel buffering policy called **TOP-Q**, that takes into account the probabilistic behavior of traversals during suffix-tree construction. This strategy uses only the path length invariant (formally defined in Chapter 4) of suffix-tree nodes, and results in a computationally low-overhead policy that outperforms other popular database buffering policies.

Second, we show that the much preferred physical representation for suffix-trees that stores sibling nodes in the tree as a linked-list, is extremely expensive in terms of disk I/O costs. As an alternative, we propose the use of a simple array of pointers at every internal node, which we show to be more I/O efficient despite the increased space overhead.

A significant advantage of our proposal is that all the existing suffix-tree based bioinformatics tools can be migrated to persistent store without having to reinvent or reimplement the algorithms. This is due to the fact that unlike alternate proposals for suffix-tree building [61, 123], we completely retain *all* the structural elements of suffix-trees. In particular, the *suffix-links* between internal nodes, which play an important role in linear time construction and subsequent querying over suffix-trees are retained in our technique.

1.5.2 Storage Management of Suffix-Tree Indexes

In most applications of sequence indexing, the construction is over a relatively static data – for e.g., NCBI GenBank is released only every two months, while the curated SwissPROT database is released even more infrequently. During this period, there could be millions of queries on these databases. Enabling practical construction of persistent suffix-trees is only the first step in their wide-spread acceptance as sequence index structures within database systems. In addition to providing efficient means of constructing persistent suffix-trees, it is also important to devise techniques to ramp up their *search performance*.

The performance of persistent index structures is measured in terms of the amount of I/O incurred during searches over them. Popular database index structures such as B-Trees and R-Trees are designed specifically for use in secondary memory, and such trees are characterized by their structural balance as well as the sizing and the branch factor of their nodes designed to exploit the disk pagesize. Unlike such indexes, the structural properties of suffix-trees are the outcome of the properties of the indexed sequence. They are typically "tall and skinny" with small-sized nodes and small fanout at every node – limited by the size of the alphabet of the sequence.

In this thesis, we investigate whether it is possible to optimize the *layout* of the suffix-tree with regard to the assignment of tree nodes to disk pages, such that the search performance over the resulting layout is improved. While layout has been well-studied in the database literature for access structures such as kdb-trees [103], Quad-trees [107] etc., we are not aware of any similar work for suffix-trees.

In the above context, this thesis makes the following contributions:

First, we show through extensive experimental results that standard layouts of persistent suffix-trees optimize the locality of only either the tree-edges or the suffix-links, resulting in slow performance of suffix-tree search algorithms that utilize both forms of inter-node connections.

Second, we present a linear-time, top-down layout algorithm called **Stellar** (Suffix-Tree Edge and Link Locality AmplifieR) that attempts to achieve the goal of optimizing the locality of both tree-edges as well as suffix-links. Through empirical evidence we show the superiority of searching over resulting layout of the persistent suffix-tree.

Finally, we present experimental results to show that searching of suffix-trees without suffix-links incur more than *2 to 3 times* the I/O required when these same searches are carried out over a linked suffix-tree. These results clearly bring out the need for maintaining suffix-links in persistent suffix-trees.

1.6 Thesis Contributions

The main contributions of this dissertation are fourfold:

- First, we present the design and implementation of **BODHI**, a holistic database system that seamlessly integrates the spectrum of data types involved in modern day biodiversity research. In addition to a comprehensive functionality, BODHI is also equipped with a variety of index structures to enhance the query performance significantly.
- Second, we present techniques for efficient construction of persistent suffix-trees, the biological sequence indexes integrated within BODHI. Specifically, we show that traditional linear-time algorithms for suffix-tree construction can be transparently scaled to work efficiently for persistent suffix-trees as well, through the use of a novel suffix-tree-aware buffering policy, **TOP-Q**, and a careful choice of physical representation.
- Third, we show how to organize the persistent suffix-tree nodes on disk such that the overall I/O incurred during search tasks is minimized. We present a technique called **STELLAR** that provides improved spatial locality for both suffix-link and tree-edge traversals, speeding up biological sequence search tasks.
- Finally, we present a detailed experimental evaluation of the performance of various features of BODHI.

1.7 Organization

The remainder of this dissertation is organized as follows: In Chapter 2, we review published related work. Next, in Chapter 3, we describe, in detail, the design and implementation of the BODHI system. Some relevant background material regarding the suffix-tree indexing for biological sequences is presented in Chapter 4. Our TOP-Q buffering strategy for high-performance persistent suffix-tree construction is presented along with a detailed performance evaluation over a variety of genomic sequences, in Chapter 5. Moving on to the search aspect, in Chapter 6, we describe STELLAR, a search optimized storage organization strategy for persistent suffix-trees. We explore applicability of our techniques for high-performance persistent suffix-tree indexes over protein sequences, in Chapter 7. A detailed performance evaluation of the full BODHI system, highlighting the utility of various indexing schemes available, is presented in Chapter 8. Finally, Chapter 9 summarizes the contributions of this thesis, and outlines future research directions.

Chapter 2

Related Work

In this chapter, we review the research literature related to the main contributions in this dissertation – namely, biodiversity information management, efficient construction of persistent suffix-tree indexes, and storage techniques for high-performance searching over persistent suffix-trees.

2.1 Biodiversity Information Management

Biodiversity data consists of both macro-level and micro-level information ranging from ecological information to genetic makeup of organisms and plants. Apart from our work, we are not aware of any other effort that attempts to combine the complete spectrum of information, though the need for it is highlighted in a proposal for GBIF (Global Biodiversity Information Facility) by OECD (Organization for Economic Co-operation and Development) [104]. This proposal identifies the domain level challenges in building a global, interconnected data repository of biodiversity information systems and notes that the urgent requirement in biodiversity studies is a suitable information management architecture for handling vast amounts of diverse data.

Recently, a group of computer scientists, biologists and natural resource managers met to discuss the computer science and information technology needs in biodiversity and ecosystem informatics (BDEI), and the report of the workshop [92, 109] discusses at length the technological and deployment challenges in this area. One of the key information management challenges they highlight is the need to efficiently handle the complexity and variety of biodiversity data.

For a biological data management system to be effective, it is critical that it be able to manage data at multiple levels of complexity, granularity, consistency and scale. In response to the wide variety of data management requirements, researchers in each individual domain have built their own tools that have vastly varying capabilities in handling of available data. In the rest of the section, we review data management solutions proposed for each of the individual domains.

2.1.1 Taxonomy Data Management

In the area of macro-level biodiversity information management, there have been many governmental efforts such as *ERIN* [13], *INBio* [88] and some global initiatives such as *Species 2000* [114], *the Tree of Life* [77], etc. However, the focus of most of these projects is on the collection, curation and dissemination of taxonomy data. They do not specifically address the issues of data management, efficient storage and querying, and do not provide sophisticated query interfaces to perform data analysis.

The Prometheus project [98] addresses the topic of extensible modeling of taxonomic classification data. Due to the lack of a standardized classification model, taxonomists arrange organisms into classification hierarchies according to various criteria (for e.g., morphological similarity, or more recently, DNA relationships). Although newer classification systems appear, it is also important to preserve the earlier classification hierarchies for a variety of reasons. In order to support this requirement, Prometheus proposes a extended object-oriented model with built-in graph functionality [99, 100]. Although the goals of our BODHI system differ from those of Prometheus, modeling extensions proposed in Prometheus can be easily incorporated into the BODHI object model.

Recently, in [86], authors have proposed an architecture for the analytical needs biologists in the area of *cladistics* – i.e., the science of developing and studying the phylogenetic models of evolution. Their present a normalized data model where phylogenies (or taxonomies) are stored as lists of edges, and use formulations involving transitive traversals to answer a variety of queries occuring the domain. While we aim at providing a seamless data integration, it is conceivable to enhance the feature-set of BODHI by incorporating ideas proposed in their paper.

In [105], authors have noted the applicability of OODBMS (object-oriented database management systems) for naturally modeling the inherent hierarchical structure of taxonomic information. Extensive literature is available on the design and implementation issues in OODBMS [134], too vast to be summarized here. We provide a brief overview of two specific indexing issues in this domain, namely, the indexing of inheritance hierarchies and the indexing of aggregation paths, highlighting on the techniques chosen in BODHI.

Indexing Inheritance Hierarchies. A direct implication of the concept of class hierarchy or inheritance hierarchy in object-oriented (OO) data modeling, is that on query evaluation it is necessary to consider the *class scope* of the query. In other words, the query could be evaluated on the extent of only one class in the inheritance hierarchy, or on the extents of all the classes in the hierarchy. There are a number of *class hierarchy index* techniques that have been proposed to address this issue, such as CH-Trees [71], H-Trees [76], Class-division indexing [102], hcC-Tree [115], etc. However, these techniques typically accelerate only one form of class scoping, i.e., only a single class or all the classes in the hierarchy. The *Multi-key Type Index* (MT-Index) [83], chosen as the inheritance hierarchy index in BODHI, provides an elegant approach, based on optimal linearization of inheritance hierarchies, to efficiently evaluate both forms of object retrievals. In addition, it can be implemented with relative ease, using a multi-dimensional indexing structure such as an R-Tree [56], or its variants.

Aggregation Path Indexing. The classes in a OO data model are related to each other through aggregation relationships, forming a directed graph called aggregation hierarchy representing the nested structure of classes. These class-level relationships result in complex inter-relationships amongst data objects (instances of these classes). OODBs typically support queries involving these nested objects, with predicates on either up-

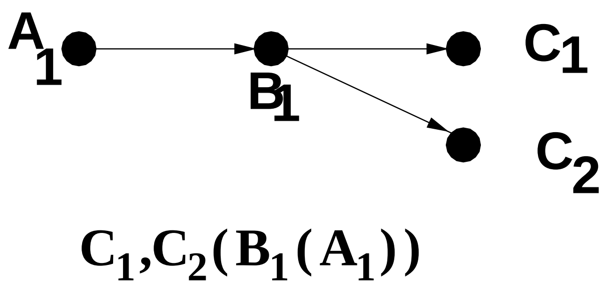


Figure 2.1: Aggregation Graph and its S-Expression

stream or downstream objects of a relationship path. A number of aggregation path index structures are available, such as Multiple-index [78], Join indices [129], Nested and Path Indexes [10], etc. However, many of these indexes efficiently only support either the upstream or the downstream predicates, and in some cases (such as in the case of Multiple-index), the inequality predicates cannot be evaluated.

The aggregation path index used in BODHI is based on the Path-dictionary Index (PDI) [74], which supports efficient computation of all aggregation path queries. The PDI access structure consists of two parts:

1. **Path Dictionary:** It is the central data structure which abstracts out the connections between the objects in the form of a *s-expression*. The s-expression recursively encodes all paths terminating at the same object in a leaf class. Given a path $C_1C_2 \ldots C_n$ the s-expression is defined as follows:

   $S_1 = \theta_1$, where θ_1 is the OID of an object in the class C_1 or null.

   $S_i = \theta_i(S_{i-1}[, S_{i-1}])$ $1 < i \leq n$, where θ_i is the OID of an object in class C_i or null, and S_{i-1} is an s-expression for the path $C_1C_2 \ldots C_{i-1}$.

   Thus, S_i is a s-expression of i levels, in which the list associated with θ_i contains recursively the OIDs of all the *ancestor* objects of θ_i. In other words, it is an encoding of all paths terminating at the object θ_i which is type C_i in the aggregation path above. Figure 2.1 illustrates a toy aggregation graph between objects of three classes – A, B, and C, and also the corresponding s-expression.

2. **Identity and Attribute Index:** These indexes are built on top the path dictionary, to quickly locate the s-expression records of interest in the path dictionary.

> The *identity index* locate the s-expression(s) in the path dictionary that contains a given OID. The *attribute index* is an optional auxiliary index built on attributes of a given class such that path traversals conditioned on attribute values can be performed on the path dictionary without accessing the involved object.

PDI reduces the cost of query processing by speeding up both associative search and object traversals.

2.1.2 Spatial Data Management

A large portion of the data involved in ecological studies comprises of spatial data and the queries over this data are expensive both in terms of I/O complexity as well as computational requirements. Due to the growth and popularity of Geographical Information Systems (GIS), database researchers have studied a variety of data management issues in this area. These include the design of a number of indexing structures for spatial queries, benchmarks designed to measure the performance of spatial data handling and the architecture of large-scale spatial database systems – each of which is reviewed below.

Multi-dimensional Indexing. A number of multi-dimensional structures for indexing spatial data have been proposed, for instance, k-d-b-trees [103], Quad-trees [107], Grid Files [91], R-Trees [56], and so on (a survey of related structures is available in [40]). In recent times, R-Trees and R*-Trees [7] are considered defacto spatial indexes due to their attractive disk-friendly properties. Most of the modern commercial database systems such as Oracle and DB2 provide an R-Tree based indexing structure as part of their spatial extensions. One issue that critically affects the performance of an R-Tree is the technique used in splitting and merging of nodes and their Maximal Bounding Rectangles (MBRs). R*-Trees were proposed to improve the packing of standard R-Trees, through their *forced re-insert* strategy – analogous to the deferred splitting in B$^+$-Trees. In [67], a novel variant of R-Tree called Hilbert R-Tree, was introduced. It uses the ordering of spatial objects imposed by a space-filling curve, in order to minimize the area and perimeter of the resulting MBRs, thus improving the quality of packing. Through experiments on a number of real-world datasets, it is shown that Hilbert R-Trees provide significantly better packing than R*-Trees, leading to improved query performance.

Benchmarking. In order to compare the performance and functionality profile of spatial databases, a well designed benchmark suite is essential. In this regard, the SEQUOIA-2000 [118] benchmark proposal has become extremely popular. SEQUOIA-2000 takes into consideration the massive size of spatial data, complex data types such as geometries of spatial objects, and the presence of sophisticated search requirements to propose a suite of 11 queries and benchmark dataset available at three scales – regional, national and world level. We use a large subset of these queries in our performance study of BODHI.

Finally, the design and architectural issues in building a scaleable spatial database system has been explored in detail as part of the Paradise project [30, 95, 96]. They propose a variety of performance tuning techniques and functionality enhancements to be used in spatial databases, and provide a detailed analysis of the system.

2.1.3 Molecular Data Management

The micro-level biodiversity data, or genetic information of various species, has been growing steadily due to the multitude of genome sequencing initiatives. The specific data management issues in handling such data have been addressed in quite a few proposals [50, 51]. In all of these proposals, the database management architecture has been tailored for the specific purposes of the project. Consider the *ACeDB* (A C.elegans Database) [32] system, originally proposed for the *C. elegans* genome sequencing project. ACeDB is an object oriented data management tool that has many features, including the handling of missing data and schema evolution, that makes it an extremely popular software in many sequencing projects. However, despite its popularity in the genome sequencing community, it cannot be considered for the larger requirements of biodiversity data handling due to the following reasons: (1) Lack of support for geo-spatial data; (2) Weak support for database updates; and (3) Lack of recovery mechanisms necessary in large data repositories.

In BODHI, we have provided the key strengths of ACeDB (its sequence processing algorithms and object-oriented basis), and augmented it with the strong database functionalities and related features that are necessary for a complete biodiversity information repository.

With the publishing of the draft sequence of the complete Human genome [62], as well as the availability of the complete genomes of many other organisms, the data analysis requirements have increased both in complexity and scale. As a result, there have been efforts to integrate the sequence analysis functionality within the database systems to improve efficiency and to reduce the data movement. The approach taken by IBM's DB2 is to provide a high-performance implementation of BLAST as a user defined function (UDF) that can be called from within SQL queries [108]. Similarly, in Oracle 10g, BLAST and regular expression search features are built into the database system and can be used as part of SQL queries [116]. Both these approaches result in query language extensions similar to those proposed in our BODHI system. However, in BODHI we provide, for the first time, a detailed performance evaluation of biological sequence similarity queries.

2.2 Large-scale Biological Information Systems

The coming together of diverse branches of biology has necessitated the design of information systems that present a single common platform for the combined needs of biologists. Although there have been many proposals for the design of such a system, a majority of them have focused mainly on the requirements in molecular biology domain. Drawing on the experience of building, deploying and using such systems, there are a small but growing number of similar efforts aimed at biodiversity informatics.

Existing biological information integration systems can be classified into the following three categories: (i) Navigational integration, (ii) Mediator-based integration, and (iii) Warehouse integration. In the rest of this section, we provide a brief overview of each of these approaches and highlight their applicability in the context of biodiversity information management. We conclude the section by positioning our proposed BODHI framework with respect to these approaches.

2.2.1 Navigational Integration

The navigational information integration refers to the creation of authoritative portals on the World Wide Web (WWW) so that the users can navigate across data sources, starting their exploration at the portal. Within this there are two distinct approaches: (i) Link-based integration, and (ii) Collaborative portals.

The link-based integration of data sources closely resembles the users' mode of accessing information on the WWW through the use of search engines. Typically, there is a centralized portal for starting the search for specific information. Users can express their search intent through the use of simple keywords through a form based interface, and possibly restrict their searches to a subset of data sources linked to the portal. The portal also maintains an index generated by pre-processing the data in the repositories participating in the integration. The keyword queries are used to lookup this index and generate hypertext links that point to a web-page containing the results at the data repository. Note that neither the actual data is stored locally nor are the queries evaluated dynamically at the sources. Some of the initial data integration systems in molecular biology, such as SRS [35], and in biodiversity information systems such as Species 2000 [114], ENVIS [34] etc., belong to this category of integration systems.

In contrast, the collaborative portals are *centralized* data repositories. The data is maintained and curated through collaborative efforts of number of users around the world each responsible for a portion of the repository. For simplifying the process of depositing the data, these systems typically use simple navigational structures and use hyperlinks to connect different parts of the information hierarchy. The Tree of Life project [77] is based on this form of information integration.

The main attraction of navigational information integration is the technical simplicity of its deployment and ease of use. While these have proven to be an initial boon to biologists, they do not scale with increasing volume of data involved as well as the need for increased sophistication in query capabilities. It has been pointed out that the "point and navigate" paradigm employed for data access in these systems imposes severe functionality restrictions that limit their effectiveness [26].

2.2.2 Mediator or Wrapper-based Integration

A myriad of data sources serving data in different formats, catering to different aspects of genomics, biodiversity and GIS information have been already set up. The proliferation of such specialized data repositories, coupled with continued expansion of existing repositories, requires techniques to integrate these diverse federated data sources with *minimal* intervention.

The wrapper-based integration, also called *query shipping* approach of data integration, tries to address this issue by providing a middleware *mediator* layer that encloses many data sources. This middleware provides a generalized user interface, thus shielding the users from having to learn the interfaces and nuances of specific data repositories. For each data source that participates in this integration, a new *wrapper* is available which is responsible for transforming the user queries into the query format of the data source, and to transform the query results into the generic representation format used within the middleware.

Due to the lack of integrated database solutions unifying different facets of biological information, wrapper-based integration has become extremely popular in molecular-biology domain. For example, K2/Kleisli [14], DiscoveryLink [57], and TAMBIS [6] use source-specific wrappers for extracting data from both static sources as well as application programs such as the BLAST family of similarity search programs. Inspired by this, the biodiversity community has also started considering similar integration efforts. For example, the GBIF (Global Biodiversity Information Facility) [104] effort proposes to use a world wide net of independent data sources, accessible through a wrapper mediator layer.

2.2.3 Warehouse Integration

As opposed to the query shipping approach of mediator systems, the warehouse integration uses the *data shipping* approach. In other words, integration of biological information is achieved by extracting, cleaning and curating data from a multitude of sources, made available through a single repository. Thus, warehousing requires that all the data loaded from the sources be converted through some data mapping to a standard format before it is physically stored in the local warehouse.

The main advantage of the warehouse approach is that the system performance tends to be much better since the query optimization can be performed locally and communication latency to access various data sources is eliminated. System reliability is also better since there are fewer dependencies on network connectivity or on the availability of underlying data sources – data sources may go down, or become overloaded and temporarily unable to answer queries. It is also easier to enforce any inter-database constraints. Another advantage of warehousing is that while the underlying data sources may contain errors, often the only feasible way for the integrated view to have correct data is to keep a separate cleansed copy in the warehouse. Furthermore, the researcher may have additional information or annotations to add to the integrated view, which is either entered manually or with the help of a software package guided by a human. The added-value of corrected and annotated data stored in the data warehouse is significant.

As observed in [58], with pervasive data inconsistency amongst independent data sources, the ultimate solution is to install a well curated repository of biological information. They also propose a *Genomics Algebra* that integrates the genomics data management requirements, including the analytics components, in the kernel of the database system. Apart from this recent effort, the only other warehouse driven approach that we are aware of is GUS (Genomics Unified Schema) [27].

2.2.4 Design Approach of BODHI

The above mentioned data integration efforts cater mainly to the analytical needs of biologists. However, there are many situations where simply providing a functional integration is not sufficient. For example, a group of researchers from multiple biological domains investigating endemic medicinal plants might not only require functional integration, but also require (i) efficient storage and query processing, (ii) the ability to create novel information, and (iii) the ability to easily disseminate the data among group members. Thus, the information management system should not only address the integrated view of diverse data as an aid for analytical studies, but also provide effective methods to generate and curate new information locally.

In order to address these extended requirements, BODHI is designed to provide a *database architecture* that can *seamlessly* and *efficiently* integrate diverse data types that are common in biological studies. In addition, BODHI system utilizes XML (along with domain-specific XML DTDs) to publish data and a standardized query language (OQL) interface, making it useful in the development of both warehouse and wrapper-based information integration systems.

2.3 Indexing of Biological Sequences

With the introduction of high-throughput genome sequencing techniques almost two decades ago, the volume of genomic sequence information is growing at an exponential rate. Besides the human genome with about 3 billion basepairs [1], complete genome sequences of many other species have been already sequenced and are available in public repositories such as GenBank. The number of daily query loads over these data repositories are comparable to those of widely used search engines, which motivates the need for developing efficient techniques for the purpose.

Currently popular techniques for sequence-similarity searching over genomic repositories include: dynamic programming-based Smith-Waterman algorithm [113], FASTA [75] and BLAST [1]. Of these, the Smith-Waterman method is an accurate search method, while FASTA and BLAST techniques are based on heuristics that trade precision of results for speed. The dynamic programming method has a $O(|D| * |q|)$ cost, where $|D|$ is the number of symbols in the database sequence and $|q|$ is the length of the query sequence. Although FASTA also has the quadratic worst-case complexity as that of dynamic programming, the associated constant factors are much smaller. On the other hand, BLAST

[1] In genetics, two nucleotides on opposite complementary DNA or RNA strands that are connected via hydrogen bonds are called a basepair (often abbreviated bp).

has a time complexity linear in the size of the database, and is currently the most popular tool for searching genomic repositories. With the rapid growth of repositories, faster techniques than the currently used tools like BLAST are required for sequence-similarity searching.

Naturally, there is a growing interest in developing techniques to build indexes for the sequence repositories, which can be productively used for high-speed searching. Extant genomic indexing techniques can be classified into the following two broad categories: (i) Word-based index structures, and (ii) Sequence index structures. In the remainder of this section, we provide an overview of some of the key secondary-memory indexing techniques in each of these categories.

2.3.1 Word-based Indexes

In word-based indexing techniques, the sequence is seen to be a collection of (possibly overlapping) substrings called *q-grams*. This view is similar to the highly successful bag-of-words approach for representing text in information retrieval domain [5].

1. **CAFE:** This technique employs a two-stage process for searching for all similar sequences in genomic databases [131, 132]. An initial coarse-grained search is done through the use of a compressed inverted-index built using overlapping substrings of a fixed length. In the second stage, a computationally expensive fine-searching is performed on the candidates selected in the first phase to generate a ranked list of similar sequences. Through an empirical study, it is shown that the CAFE approach is significantly faster than the popular BLAST [1] and FASTA [75] search tools.

2. **ED-Tree:** In [122], a novel index structure called ED-Tree was proposed for supporting probe-based homology search algorithms like BLAST. Given a genomic sequence, S, a predefined word-length, w, and a segmentation scheme for each word, they construct a digital search tree based on the segments of the word such that every root to leaf path corresponds to a word of length w. The leaf nodes contain pointers into the indexed sequence where the w-length word matches exactly.

In order to reduce the size of leaf nodes, they apply a frame-of-reference compression scheme after sorting the offset values stored at each leaf node. Experimentally they show that a homology search algorithm using the ED-Tree can be orders of magnitude more efficient than BLAST, and is more sensitive.

3. **Piers:** An indexing application of q-gram based filtering was proposed in [15], where a subset of q-grams (or piers) are used to filter out the regions of low similarity with the query sequence, while minimizing the likelihood of false dismissals. For each sequence s in the data collection, a set of q-grams of length l_p are chosen such that at least k of them are contained in a region of length no less than l_{min} – a given threshold on the length of regions of similarity. These q-grams are then indexed using a compact hash-based structure, and cross-pier similarities are precomputed and stored in a fast lookup table. Each query sequence is then decomposed into all its l_p-length q-grams, and presented to the pier hash structure, which is used to determine the piers that are within a short edit-distance from the given q-gram, and then are further expanded to include piers with larger edit-distance through the use of precomputed cross-pier similarity table. Due to their two-stage refinement process that allows for inexact matches of piers, it is possible to eliminate larger regions of dissimilarity quickly. Through experimental evaluation, they show that their technique is 2-15 times faster than BLAST.

4. **qClusters and c-signatures:** In [16], a two-level indexing technique based primarily on the q-gram filtering was proposed. They consider all possible q-length subsequences on DNA alphabet, and define *qClusters* to be a partition of these q-grams into clusters of equal size. At the first level, DNA segments are hashed, based on the presence or absence of a representative from each qCluster into a compact in-memory hash table. Next, each DNA sequence is represented using 4^q-length bitmap, where each bit position corresponds to one of all possible q-grams. These *q-signatures* are then compacted into *c-signatures* by replacing c bit positions by the count of bits set to 1. These c-signatures are organized into a digital search tree, called *c-tree*, which forms the second level indexing structure. Their experimental

evaluation of search algorithm based on this two-level structure demonstrates 2-3 times performance improvement over BLAST.

5. **DSIM:** One of the obstacles in speeding up DNA sequence search time is the massive size of DNA data collections. In [17], improvements to substring matching performance through a sequence compression technique called *DSIM* was proposed. They borrow the ideas from video compression techniques, to compress the sequence data collection based on the substring edit-distance from a few reference-words. These reference words are chosen initially as the most frequently occuring substrings, and are then incrementally updated based on the statistics gathered from query workloads, as well as based on the updates to the data collection. Through empirical analysis, they show that their techniques outperform BLAST in speed, and are highly competitive in precision.

6. **SST:** The Sequence Search Tree (SST) proposed in [49], considered a heuristics-based solution which runs in $O(\log n)$ expected time. This technique splits the data strings into overlapping windows of length W for some pre-specified overlap amount of Δ. For each such window, they count the number of repetitions of all possible *k-tuples*, and store these values in a σ^k dimensional vector, where σ is the alphabet size. These vectors are indexed using a hierarchical binary tree constructed by repeatedly applying a K-means clustering algorithm. The similarity between the query string and a substring is approximated by using the Manhattan distance between these vectors. Experimental results show that this technique runs more than 25 times faster than BLAST.

7. **String Join using Precedence Count Matrix:** Searching a sequence data collection can be also viewed as an approximate string join problem and [18] proposes a filter-and-refine algorithm for the purpose. The novelty of their work lies in the use of a *Precedence Count Matrix* (or PCM) to efficiently estimate a lower bound for the edit distance between two sequences. A PCM of a sequence of symbols from alphabet Σ, is a $|\Sigma| \times |\Sigma|$ matrix where an entry (a, b) represents the number of

unique occurances of symbol a preceding b (not necessarily consecutive) in the sequence. These PCMs can be computed efficiently, and incur very little overhead due to the small alphabet-size of DNA. Using the PCM-set for all the suffixes/prefixes of each sequence, they show that the performance is highly competitive with that of MRS-indexing (see below).

8. **MRS-indexing:** This technique, proposed by Kahveci and Singh [66], uses an elegant two-level search process based on wavelet transformations. In the first step, each subsequence of the database is mapped into a 2σ dimensional vector space of wavelet coefficients. These 2σ dimensional space of points is indexed with standard multi-dimensional index structures such as R-Trees [56]. Range queries and nearest-neighbour queries can be efficiently performed using this indexing strategy. The technique also guarantees that there will be no false-dismissals with the standard edit-distance metric. Recently, MRS-indexing has been extended for indexing protein sequences and score matrices [65]. They use the index to prune away unpromising portions of the data-sequence, thus enabling tools like BLAST to perform focused computations.

Almost all of the techniques presented above suffer from a serious drawback – they are directly applicable only when the similarity search is based on the standard *edit-distance* or *Levenshtein-distance* metric. However, more often than not, biologists use domain-specific specialized alphabet-scoring schemes while performing similarity searches over the database. For example, in the case of phylogenetic tree construction techniques, biologists weigh the intra-purine/intra-pyramidine ($A \rightleftharpoons G$, $C \rightleftharpoons T$) transformations lower than the transformations across these groups (e.g., $A \rightleftharpoons C/T$ etc.) [53]. This is done in order to account for the estimated evolutionary DNA mutation rate, where intra-purine/intra-pyramidine substitutions are expected to be more likely to occur by chance, and may not be indicative of the true phylogenetic divergence. As a result, it is necessary for a database sequence index to be applicable under different symbol-wise edit metrics as well.

Furthermore, these techniques are aimed at improving the performance of probe-based sequence similarity techniques like BLAST, which may not detect all the homologous regions. In other words, the sensitivity of these algorithms is typically lower than the Smith-Waterman algorithm.

2.3.2 Sequence Index Structures

In contrast to word-based indexes outlined above, sequence index structures preserve the sequential nature of strings while indexing. Suffix-tree is a key index structure in this class of data structures. A rich body of research exists on utilizing the suffix-tree of a sequence in performing many sequence processing tasks [4, 53, 54]. In fact, suffix-trees, discussed in the next section, have been considered as the de-facto indexing strategy in bioinformatics domain. In this sub-section, we present some of the related data structures.

1. **Suffix-Array:** Manber and Myers [79] proposed the Suffix-array data-structure that is very space efficient and can be used to perform exact string matching or substring matching. Conceptually, the suffix-array of a string is an array holding the indexes of all the suffixes of the string sorted in lexicographic order. Since the suffix-array holds only the index values of the string, it requires only $|D|$ integers. When coupled with an additional array holding *longest common prefixes* of adjacent elements, the suffix-array can be used to find all occurrences of the query q in $O(|q| + \log_2 |D|)$ time. This is in contrast to suffix-tree, which can perform the same query in $O(|q|)$ time.

2. **String B-Tree:** Motivated by the lack of external memory index structures that can efficiently handle long text strings, Ferragina and Grossi proposed the String B-Tree structure [39]. String B-Tree is a novel combination of B-Tree structure and Patricia Tries for internal-node indices that is made more effective by adding extra pointers to speed up search and update operations. However, in the biological sequence indexing, String B-Trees are of limited use since the sequences such as DNA or Proteins do not possess clearly demarcated word-structures, which are a

prerequisite for String B-Trees.

3. **SPINE:** In [90], Neelapala, Mittal and Haritsa presented a novel index structure called SPINE (Sequence Processing Indexing Engine), which, like suffix-trees, is based on trie-structures. Conceptually, SPINE can be viewed as a *horizontal compaction* of a trie on the data sequence, in contrast to suffix-trees which vertically compact the trie to remove all the non-branching nodes. The advantage of SPINE over suffix-tree is that it avoids the duplication of paths in the tree, thereby reducing the number of nodes.

It should be noted here that, although in our work we have addressed the issue of making suffix-trees – a fundamental sequence index structure – practical in a database setting, it is imperative to consider the application at hand in order to choose the index structure. For example, there are some applications, where it is sufficient to obtain *most* of the alignments under standardized parameter values. In such cases, it might be worthwhile to consider more space-efficient index structures such as MRS-index or ED-Tree, albeit their limited ability to support more flexible similarity searches.

2.4 Persistent Suffix-Tree Construction

Since the time Weiner [130] introduced the suffix-tree data structure and a linear time algorithm for its construction, there has been growing interest in more space- and time-efficient algorithms for construction of suffix-trees. Conceptually different and space efficient algorithms to build suffix-trees in linear time have been given by McCreight [80], and later, by Ukkonen [126]. Further, McCreight also suggested the use of linked-list implementation of nodes for reducing the space overhead of the suffix-trees. All these algorithms make use of suffix-links to achieve linear-time construction and are implemented for various constant-sized alphabet datasets.

Suffix-trees provide an *accurate* indexing solution for searching over large corpora of DNA or Protein sequences. Initial use of suffix-trees in genomic indexing was restricted to small length DNA sequences [11], where the suffix-tree could fit completely into main memory. In [36, 37], Farach et al. provided the initial theoretical breakthrough in persistent suffix-tree construction. They introduced a novel way to construct the suffix-tree over a large sequence by following a divide-and-conquer approach (as opposed to the traditional suffix-at-a-time approach), and used that to show that persistent suffix-trees could be built with the same I/O complexity as that of external sorting. However, due to huge constants associated with their results, these results are of only theoretical interest. They also pointed out that "traditional" algorithms (such as those of Weiner, Ukkonen, and McCreight etc.), which follow incremental extension of suffix-trees, will be forced to make random I/Os resulting in bad performance. Their observations on traditional algorithms were made without considering the effects of paging/caching policies that could be employed during the construction process. In fact, they state at the end of their paper [36], that it would be worthwhile considering the behavior of construction algorithms in presence of paging, which is a topic addressed in this thesis.

However, until recently, suffix-trees were not considered for persistent construction and maintenance as linear-time suffix-tree construction algorithms show very poor performance. The main bottleneck in the direct application of these algorithms for persistent suffix-trees is considered to be the random seeks induced during construction [36]. It has also been noted in [61] that suffix-links utilized by all these algorithms traverse the suffix-tree "horizontally", while edges span the tree "vertically". Thus, atleast one of them is expected to result in random access of memory. This is true only if the tree is stored on disk using depth-wise traversal of either edges of the tree or links of the tree. But this storage pattern is not feasible during the on-line construction of persistent suffix-trees. Therefore, in practice, *both* edges and suffix-links show non-local access patterns. In fact, even in a recent work [89], it was reported that whenever the dataset is large enough suffix-trees are not a viable option of indexing, since the memory is too small to hold the index completely.

2.4.1 Construction of Suffix-tree without Suffix-links

In [61], a phased construction approach to building suffix-trees was proposed, wherein they use an asymptotically quadratic algorithm for construction of suffix-trees without suffix-links. Their technique is based on a mapping of all suffixes of a given database sequence D to *disjoint* set of partitions, $p_w \in P$, such that suffixes with equal prefix w are mapped to the same partition $p_w = \{wx | wx = s \text{ and } s \text{ is a suffix of } D\}$. The prefix length, w, is chosen such that the size of each suffix-tree partition does not exceed the available main memory size. Otherwise the algorithm fails. Since their phased approach involves constructing parts of the suffix-tree within memory and writing it to the disk completely, their empiricaly evaluation showed its superiority over traditional linear-time suffix-tree construction algorithms. But, their results on the performance of traditional construction algorithms do not consider the effects of paging policies and the storage management issues. In fact, in [63], it has been reported that a possible bottleneck could be the choice of checkpointing scheme of PJama, the underlying persistent mechanism used in [61].

One of the drawbacks of the above approach is that it is not sensitive to the skew in the distribution of symbols of the sequence. Due to this, the size of the partitions of the tree that are built within memory could vary resulting in a non-optimal utilization of available memory. In order to overcome this, an extended partition generation scheme was proposed in [110]. An initial pass over the complete sequence is made in order to estimate the cardinality of each partition. Based on these estimates, the prefix-length used to prepare the partitions is tuned to improve the main memory utilization.

Recently, in [123], a sophisticated partition based technique called *Top Down Disk-based* (TDD) was introduced, that takes into account the effects of buffering policies during the suffix-tree construction. The TDD technique is a well-tuned combination of the partitioning approach introduced in [61] and an earlier main-memory algorithm called *wotd* (write-only top-down) [47] that was shown to incur fewer cache-misses on modern processors.

All of the above persistent suffix-tree construction algorithms suffer from a major drawback – the suffix-tree built using these algorithms is completely devoid of suffix-links. Although a large number of algorithms over suffix-trees do not exploit the presence of suffix-links, some of the critical applications of suffix-trees in bioinformatics such as MUMmer [29] depend on the availability of suffix-links. Therefore it is important to explore ways to speed up the suffix-tree construction without affecting their structure in any way – a focus of the research presented in this thesis.

A novel construction of suffix-trees was proposed by Clifford and Sergot [24], wherein they combine the advantages of partition-based approach with the utility of suffix-link based construction. It uses an extended definition of a suffix-link, such that the suffix-links between nodes in the same partition are retained and can be used to speed up the construction of the suffix-tree partition.

2.5 Storage of Persistent Suffix-Trees

In this section, we briefly summarize the earlier work in the area of disk layout schemes for skewed search trees. The earliest work that explored in detail, the issue of disk-layout for improving the search performance of arbitrary search trees is that of Diwan et al. [31]. They provide algorithms for minimization of external path length (defined as the number of edges to be traversed in a root to leaf path) in digital tree structures such as trie, k-d-b-trees etc., given a uniform access probability over leaf nodes of the tree. The worst-case external path length minimization is achieved through a $O(N)$ algorithm that does a bottom-up packing of nodes into pages. However, for average path-length minimization problem – which is more relevant here, with a large number of queries being posed over the search tree – they have a much more complex, dynamic programming based algorithm that has $O(kN^2)$ time complexity, where k is the page capacity and N is the number of nodes in the tree. Both these algorithms are shown to be optimal. However, empirical results in [31] indicate that a simpler top-down technique, called SBFS, yields the same average path-lengths in most of the cases.

Our work differs from these earlier results due to the following reasons:

- The tree traversals considered in [31] are root-to-leaf traversals commonly found in the space-partitioned trees such as k-d-b-trees, Quad-trees etc. However, the common substring searches over suffix-trees entail a more complicated pattern of traversals – involving both tree-edges and suffix-links.

- Suffix-tree is, surprisingly, a *cyclic structure*, with two tree structures – one induced by the tree-edges and the other by suffix-links – overlapping each other. Thus the optimality results provided in [31], which are applicable only for trees, are not applicable in the search procedures considered in this thesis.

Another work that is related to the suffix-tree storage organization issue, addressed in this thesis, is by Gil and Itai [48]. They consider the issue of packing trees efficiently given the "*access weights*" associated with each node in the tree and provide an optimal dynamic-programming-based packing algorithm. However, the time complexity of their algorithm is $O(BN^2)$, where B is the page size (in terms of number of nodes), and N is the number of nodes in the tree. Although their technique is applicable for suffix-tree packing as well, the quadratic costs associated with their algorithm makes it seem impractical in the case of suffix-trees. Additionally, the algorithms of [31] and [48] do not provide good page utilization (both guarantee only 50% utilization). In the case of persistent suffix-trees, already burdened with significant space overheads, it is impractical to resort to techniques that place additional demands on space. Therefore, we focus on achieving better disk layout schemes that guarantee 100% space utilization.

Chapter 3

The BODHI System

3.1 Introduction

Over the last decade, there has been a revolutionary change in the way biology has come to be studied. Computer assisted experimentation and data management have become commonplace in the biological sciences and the branch of *Bio-Informatics* is drawing the attention of more and more researchers from a variety of disciplines. A key area of interest here is the study of the *biodiversity* of our planet. The database research community has also realized the exciting opportunities for novel data management techniques in this domain [73].

The study of biodiversity, as outlined by the WCMC (World Conservation Monitoring Center) [133], is an integrated study of *Species*, *Ecosystem* and *Genetic* diversity. The data associated with these domains vary greatly in the scale of their structural complexity, their query processing costs, and also their storage volume. Thus, supporting such diverse domains under a single integrated platform is a challenge to the data management tools currently used by the biologists. More often than not, these scientists make use of *different* tools for managing and querying over each of the domains, leading to difficulties in performing cross-domain queries.

In order to address this lacuna, we present BODHI, a prototype system that addresses many of the issues arising in the biodiversity information management, and in addition, provides a platform to implement many computational and analytical solutions required by the biologists. The BODHI system has been designed and developed in collaboration with domain scientists from the Center for Ecological Sciences and the Department of Molecular Reproduction, Development and Genetics, at our institution.

BODHI is a *native* object-oriented system that naturally models the complex objects ranging from hierarchies to geometries to sequences that are intrinsic to the biodiversity domain. In particular, it seamlessly integrates taxonomic characteristics, spatial distributions, and genomic sequences, thereby spanning the range from molecular to organism-level information. To the best of our knowledge, BODHI is the *first system to provide such an integrated view.*

BODHI is fully built around publicly available database components and system software, and is therefore completely free. In particular, the Shore micro-kernel [19] from the University of Wisconsin (Madison) forms the back-end of our software, while the λ-DB extensible rule-based query optimizer [38] from the University of Texas (Arlington) is utilized for production of efficient execution plans. The system is currently operational on a Pentium PC hosting the Linux operating system.

Efficient query evaluation is one of the important goals of the BODHI system. To achieve this goal, a variety of sophisticated access structures, some drawing on the recent research literature, have been implemented to provide efficient access to various data types. For example, the Path-Dictionary [74] and Multi-key Type indexes [83] accelerate access to inheritance and aggregation hierarchies, while the R*-tree [7] and Hilbert R-tree [67] are used for negotiating spatial queries. To improve the performance of a wide class of biological sequence similarity searches, BODHI features *persistent suffix-tree* indexes. In order to support their efficient construction and querying, a specialized buffer management policy and an optimized storage technique are built within the sequence processing engine of BODHI.

The BODHI server is compliant with the ODMG standard [21], supporting an OQL/ODL query and data modeling interface. To enable easy interfacing with the system a web-based graphical query form is provided. Through this graphical interface, biologists can construct complex OQL queries involving taxonomy, spatial and genome-sequence predicates. Further, the server is capable of outputting the result objects in XML format, enabling client-applications to format the results in their favorite metaphor.

In this chapter, we present the design and implementation of BODHI, a database system tuned specifically for the needs of the biodiversity research community. To the best of our knowledge, this is the first such system supporting diverse data domains ranging from genomic sequences to geographical features, and supporting queries that span across these domains. We demonstrate the utility of the BODHI system over a *plant biodiversity* database, although similar databases over other biodiversity entities such as animals, fish etc. can be designed as well.

3.1.1 Organization

This chapter is organized as follows: In Section 3.2, we highlight the design goals for a biodiversity information system. Then, in Section 3.3 we provide a detailed description of the architecture of BODHI. The salient implementation features of BODHI, including the flow of query and metadata within the system are presented in Section 3.4. Finally, we conclude in Section 3.5.

3.2 Design Goals

In this section, we highlight the main features that would be desirable in a biodiversity information system. These include efficient handling of complex data types, facilities for bio-molecular sequence similarity, and user-friendly interfaces, described in more detail below.

3.2.1 Handling of Complex Data Types

Biodiversity data can be broadly classified into the following three categories:

1. **Taxonomy Data:** Taxonomy is the science of systematic classification of organisms. The taxonomy data typically involves deeply nested hierarchies depicting

the relationships between various species based on their observable characteristics. These relationships include *Phenetic relationships* – that are founded on physical or directly observable characteristics of the species, and *Phylogenetic relationships* – derived from evolutionary theory [94]. Modeling of these relationships could involve the extensive use of aggregate types such as Sets, Bags and Sequences. The various characteristics on which these relationships depend may vary in time, due to the discovery of a new class of characteristics, corrections to previously recorded characteristics, etc.

2. **Geo-spatial Data:** The study of ecology of species involves recording the geographical and geological features of their habitats, water-bodies, and ecologically relevant artificial structures such as highways which might affect the ecology, etc. As with any spatial information, the volume of the associated data is huge and the queries involve predicates over geometric relationships that are computationally expensive to evaluate. In addition, there is a need for the spatial data to be available under multiple categories (such as administrative-regions, bio-geographic provinces, forests etc.) at the same time.

3. **Bio-molecular Data:** The genetic makeup of species is becoming increasingly important with the completion of a large number of organism and plant genome sequencing projects. For example, "bio-prospectors" look for indigenous sources of medicines, pesticides and other useful extracts, which can be discovered from the biomolecular and genetic composition of species. In addition, the queries over this form of data are typically complex *similarity* queries – for e.g., one would like to retrieve all the DNA sequences in the database that have regions of significant similarity with a query sequence. Thus, it is important to support modeling and efficient querying of this type of data.

The above data-types have complex and deeply-nested relationships within and between themselves. Further, they may involve complex aggregate structures such as sequences and sets in their relationship paths.

3.2.2 Bio-molecular Sequence Similarity Search

The molecules that are of primary interest in biodiversity are DNA and Proteins. DNA is represented as a long sequence based on a four nucleotide alphabet. There are regions in the DNA sequence, called *exons*, which contain the genetic code for the synthesis of Proteins. The proteins are long chains of 20 amino acids. Each protein is characterized by its amino acid patterns, and is responsible for various functionalities in a cell which, in turn, determine the characteristics of the organism or plant.

The similarity between two genetic sequences is a measure of their functional similarity. Analysis of DNA and Protein sequences from different sources gives important clues about the structure and function of proteins, evolutionary relationships between organisms, and helps in discovering drug targets.

As we mentioned earlier, there are a number of popular algorithms, such as Smith-Waterman, BLAST [1], FASTA [75] etc., for performing similarity searches over genetic sequences. Researchers and bio-prospectors frequently search the database using these algorithms to locate gene sequences of interest. However, the implementation of these algorithms is typically external to the database, making them relatively slow. It therefore appears attractive to consider the possibility of integrating these algorithms in the database engine. In fact, two of the leading commercial database vendors, IBM and Oracle, have recently enhanced their database products by providing BLAST homology search extensions to their query repertoire [108, 116].

3.2.3 Usage Interface

As with all other scientific communities, the biodiversity community relies on timely knowledge dissemination. Therefore, supporting access through the Internet is vital for maximizing the utility of the information stored in the database.

Typically, biodiversity data is autonomously collected and managed by individual research institutions and commercial enterprises. In order to improve data availability, it is necessary that such localized and autonomous data repositories be able to exchange data. The current state of information exchange amongst various biodiversity data repositories is not very satisfactory [104]. However, with the advent of XML, many research groups are proposing DTDs in individual fields of ecology and genetics [3, 12]. A biodiversity information system should support these DTDs for handling data over heterogenous set of repositories.

It is also imperative to have a good visualization interface for the results produced by the system since (a) the end-users are biologists, not computer scientists, and (b) the results could range from simple text to multidimensional spatial objects.

Finally, most of the research in biodiversity is done by small teams of researchers who work with low budgets and are unable to afford high-cost data repository systems. Therefore, solutions that are completely or largely based on public-domain freeware which can be hosted on commodity hardware are essential for these groups.

3.3 Architecture of the System

As mentioned earlier, biodiversity data is inherently hierarchical and has complex relationships. In order to enable *natural* modeling of these entities and their relationships, BODHI is designed as an *object oriented* database server, with OQL/ODL query and data modeling interfaces.

The overall architecture of BODHI is shown in Figure 3.1. The Shore [19] storage manager 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 query processor, based on theλ-DB extensible rule-based query processor and optimizer, interfaces with these functional modules and performs query processing to produce 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 client interface framework and XML publishing engine form the external interface to BODHI.

The BODHI server is partitioned into three service modules: *Object*, *Spatial*, and

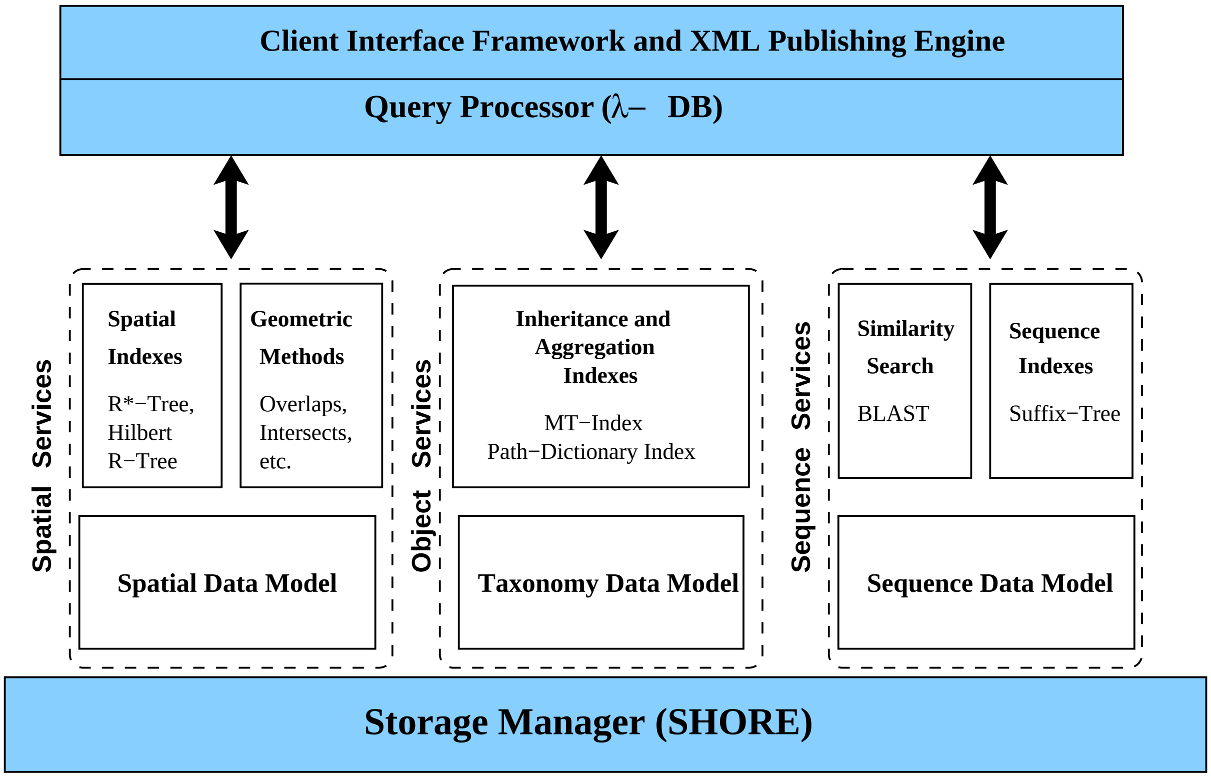


Figure 3.1: BODHI: Schematic of Architecture

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. In the remainder of this section, we describe these core database components as well as the query system and the user interface modules in more detail.

3.3.1 Object Services

While the Shore storage manager handles basic object management, it is necessary to extend the basic data type system to model spatial and genome sequence data primitives. The type-libraries necessary for these extensions are bundled into the object services component of the server.

The data modeling language of BODHI extends the standard ODL [21] by introduc-

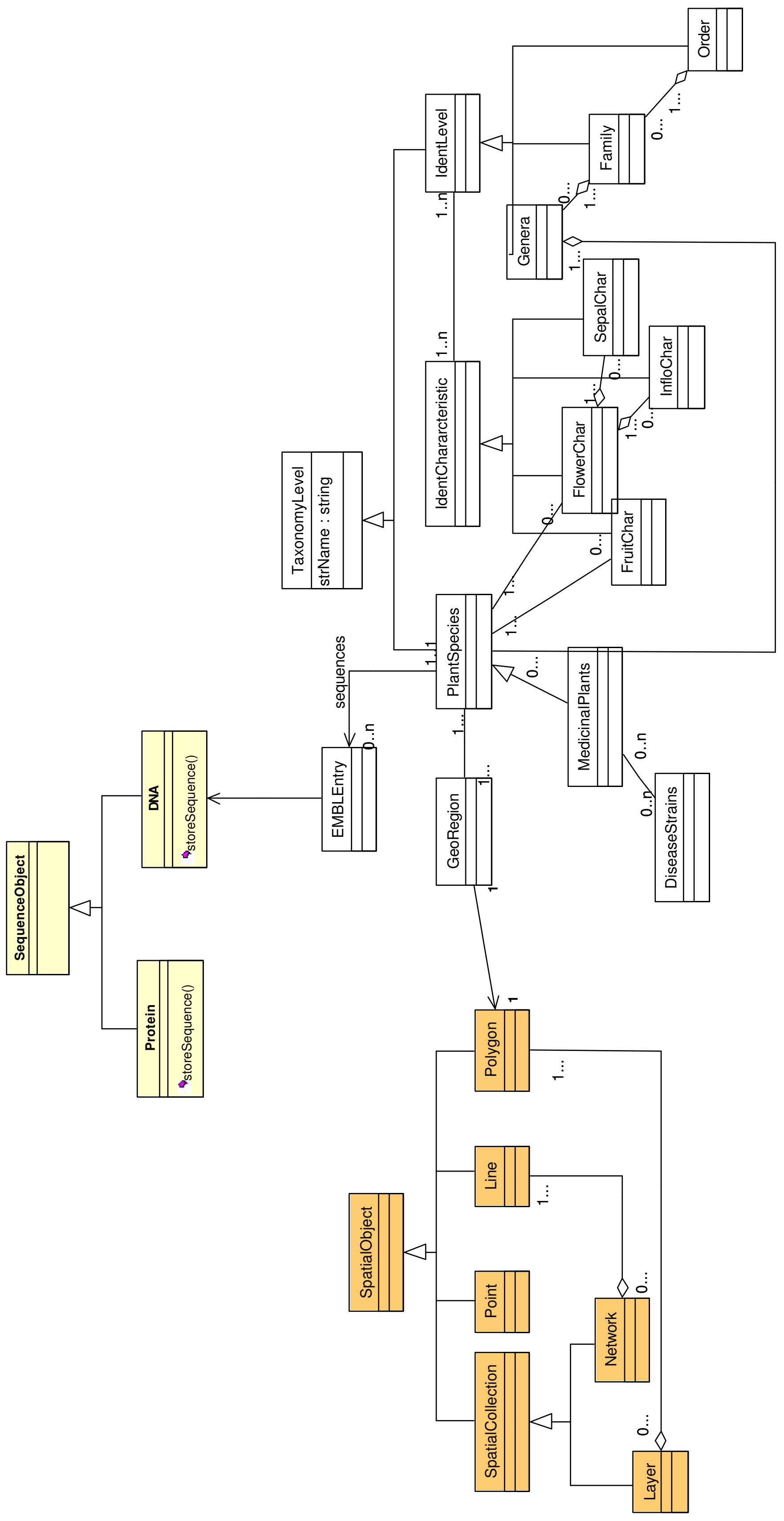


Figure 3.2: A Sample Plant Biodiversity Object Model

ing new primitives for modeling spatial and sequence data. These primitives can be used in conjunction with standard data types provided by ODL and various relationships between objects can be easily modeled. The schema definition enables objects to be interrelated through inheritance hierarchies and object-relationship paths. The queries over the database consist of both value-based queries and also on the context of the object in the relationship graph and the position of its type in the associated class hierarchy.

Indexing the Relationship Paths

Referring to the UML-diagram [127] of a sample plant biodiversity schema given in Figure 3.2, we see that the relationship hierarchy between objects in biodiversity database schema can be arbitrarily deep. Further, it is possible to have *recursive* relationships, for example, the *Predator-Prey* relationship among Species. Queries over such relationship graphs can have either the ancestor class or the nested class, as the predicate class. To illustrate, consider the following pair of queries:

Query 2 *Identify the* **PlantSpecies** *based on one or more of its* **IdentCharacteristics***.*

Query 3 *Retrieve all* **IdentCharacteristics** *of a given* **PlantSpecies***.*

In Query 2, we need to scan for object relationship path(s) culminating at the specified characteristic, and then locate the species that form the root(s) of that(those) path(s). Such queries are called TP (Target-Predicate) queries, following the terminology of [74].

On the other hand, in Query 3, the predicate classes are ancestor classes (*PlantSpecies*) and the target classes are nested classes (*IdentCharacteristics*). These types of queries are called PT (Predicate-Target) queries in [74].

To efficiently handle both PT and TP queries, BODHI implements the *Path Dictionary Index* (PD-Index) [74] approach. The PD-Index consists of two parts: the *path dictionary* which supports the efficient traversal of the path, and the *identity index* and *attribute index* which support associative search. The identity index and attribute index are built on top of the path dictionary.

Conceptually, the path dictionary extracts the compound objects, without their primitive attributes, to represent the connections between these objects. Since primitive attribute values are not stored in the path dictionary, it is much faster to traverse the nodes. In order to support associative search based on attribute values, PD-Index provides attribute indexes which are built for frequently queried attributes. When the identifier of an object is given, the path information is obtained using the identity index built over the path dictionary.

The PD-index supports both forward and backward traversals of the hierarchy with equal ease; further, its performance evaluation in [74] indicated significantly improved access times. A limitation, however, is that it only handles 1:1 and 1:N relationships. Since typical schemas of biodiversity database include aggregations of N:M cardinality, and structures such as sets, bags and sequences in the aggregation path, BODHI provides an extended implementation of the PD-index to handle these constructs.

Indexing the Inheritance Hierarchies

Based on the context, the scope of queries over inheritance hierarchies in the biodiversity domain can be either limited to the immediate objects of the predicate class (i.e. single-class query) or can be extended to include all objects of the sub-hierarchy rooted at the predicate class (i.e. class-hierarchy query). Referring again to the schema of Figure 3.2, consider the following query:

Query 4 *List the names of all* **PlantSpecies** *associated with a* **GeoRegion**.

We have two possible semantics for this query: (i) search in the complete inheritance hierarchy rooted at *PlantSpecies* and return objects of type *PlantSpecies* as well as *MedicinalPlants* associated with the given *GeoRegion*, or (ii) search objects of only *PlantSpecies* type associated with given *GeoRegion*, without searching amongst objects of type *MedicinalPlants*.

To efficiently support both forms of the query, the *Multikey Type Index* (MT-index) [83] approach is used in BODHI. The basic idea behind MT-index is a mapping

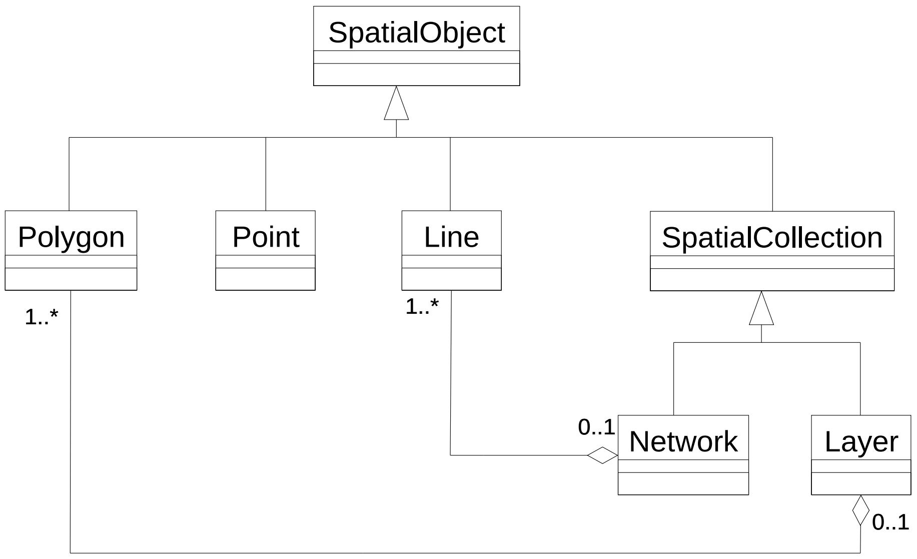


Figure 3.3: Spatial Data model in BODHI

algorithm, called *Linearization Algorithm*, which maps type hierarchies to linearly ordered attribute domains in such a way that each sub-hierarchy is represented by an interval of this domain. Using this algorithm, MT-index incorporates the type hierarchy structure into a standard multi-attribute search structure, with the hierarchy mapped onto one of the attribute domains (type domain). This scheme supports queries over a single extent as well as over extents of all the classes under the inheritance subtree. Further, by extending the number of dimensions in the index, multi-attribute queries can also be supported easily.

Apart from its elegant transformation of the type-hierarchy tree into a linear path, a major attraction of the MT-index is that it can be implemented using any of the multi-dimensional indexing schemes. In particular, BODHI implements them using the R*-Trees supported natively within Shore.

3.3.2 Spatial Services

Moving on to the next data domain, Spatial (or geographic) data, in both vector and raster formats, constitutes the bulk of the biodiversity information. Due to the inherent complexity of spatial operations (such as *overlap*, *closest*, etc.), combined with large volumes of data, spatial query processing is considered by ecologists [87], to be a major bottleneck in the expeditious processing of biodiversity queries. The Spatial Services module provides operations over an underlying *Spatial Data Type* (SDT) library, as well as efficient spatial indexing and join algorithms.

Spatial Data Types

BODHI provides a set of spatial data primitives to represent single spatial entities such as country, state, forest, river, etc., as well as to represent interrelated collection of spatial objects such as "*Political map of India*", which can be modeled as a topologically related collection of polygons, each representing a state. The standard primitive type library of ODL is extended with these data types to enable users to include spatial data definitions in their schema descriptions.

The spatial model of BODHI is based on the *ROSE Algebra* [55], and provides two categories of primitives: *Simple Primitives* and *Compound Primitives*. The simple primitives enable modeling of single objects in space, and includes types for *Point*, *Polyline* and *Polygon*. The compound primitives, on the other hand, are used to model spatially related collections of objects. There are two compound primitives: *Layer* and *Network*, for modeling collections of *Polygon* and *Polyline*, respectively. Figure 3.3 gives the class diagram of the spatial data model of BODHI.

Operations on Spatial Data

Spatial queries consist of selecting objects which satisfy some spatial relationship(s). There are three classes of such spatial relationships,

- *Topological relationships*, such as *adjacent*, *inside* etc. which are invariant under geometrical transformations like translation, scaling and rotation.
- *Direction relationships*, such as *above*, *north-of* etc.
- *Metric relationships*, that are based on the distance measure between spatial objects.

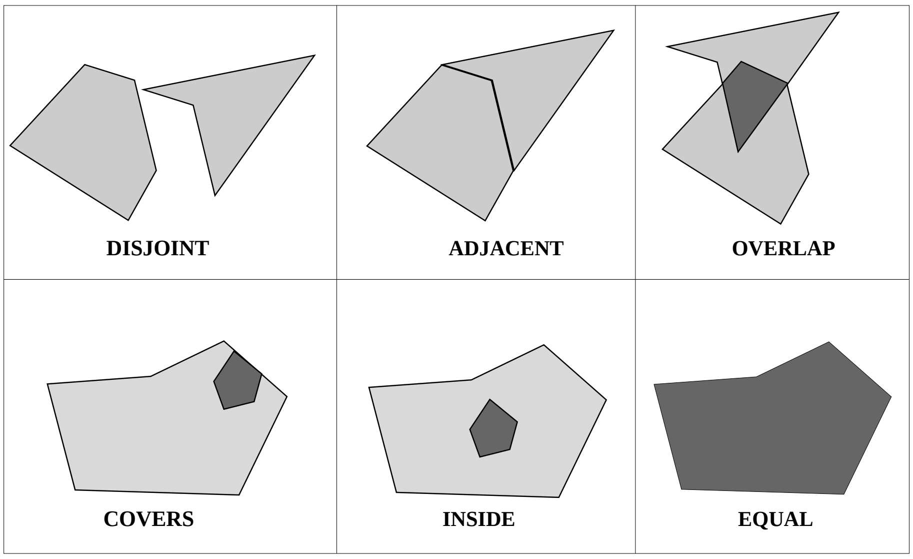


Figure 3.4: Spatial Relationships in BODHI

Of all the relationships in these three categories, [55] observes that only six relationships: *disjoint*, *in*, *touch*, *equal*, *cover* and *overlap*, are the most important relationships in geo-spatial applications. These relationships are illustrated in Figure 3.4. Even though the figure shows these relationships between only polygonal objects, they are well-defined over the complete spectrum of SDTs, as well as between different types (for e.g., between line and polygon). Accordingly, the *Geometric Algorithms* part of the Spatial Services module provides these six relationships. These algorithms form the core of behavioral abstraction of the spatial primitives described above.

Spatial Indexing

For spatial data, Shore natively supports the R^*-Tree [7], which is the most popular spatial access method since it achieves better packing of nodes and requires fewer disk accesses than most of the alternatives. However, a problem with the R^*-Tree is that even though it has tight packing to begin with, its structure may subsequently degrade in the presence of dynamic data. To tackle this, we implemented the Hilbert R-Tree [67], which is designed for handling dynamic spatial data while maintaining good packing of the index structure. It makes use of a Hilbert space-filling curve over the data-space to linearize (i.e. obtain a total ordering of) the objects in the multi-dimensional domain space. A performance evaluation in [67] shows this structure to provide better packing in the presence of dynamic spatial data and, as a consequence, better query performance.

3.3.3 Sequence Services

In modern biodiversity studies, analysis of genetic data in conjunction with the macro-level information plays an important role [82, 124]. The *Sequence Services* module interfaces with the storage manager to provide efficient storage of genomic sequences and sequence similarity search algorithms over them.

Biological Sequence Primitives

The Sequence Services module supports two primitive types: *DNA* and *Protein* to represent the sequence information of associated molecules. In order to provide a compact physical representation, the DNA alphabet of 4 nucleotides is encoded using two bits, and the Protein sequence alphabet of 20 is encoded in five bits. The behavioral part of this data model, provides functions for *translation* of DNA sequence into a Protein sequence and vice-versa, the generation of complementary DNA, and extracting substrings from DNA or Protein sequences.

Similarity Search

Comparison of sequences in order to locate similar subsequences is an important operation in computational biology, and it forms the basis for many other complex operations such as phylogenetic tree construction and multiple sequence alignment.

BODHI provides two algorithms for sequence similarity searching – the BLAST algorithm, and the Smith-Waterman dynamic programming algorithm, both of which are usable as part of an OQL query. The similarity score is computed using a set of default cost metrics, (i) PAM-120 [28] metric for proteins, and (ii) the weighted edit distance metric used in [1] for DNA sequences (+5 for matches, -4 for mismatches).

Biological Sequence Indexing in BODHI

Sequence similarity search algorithms, such as BLAST and Smith-Waterman, typically require a *full scan* of the sequence database for each query context. Not long ago, even spatial data processing required such brute-force scanning and evaluation of topological predicates on all spatial objects in the database. However, with the evolution of efficient spatial index structures such as R*-trees and Hilbert R-Trees etc., spatial data processing has become extremely efficient. Similarly, in the domain of biological sequence processing, the *suffix-tree* datastructure is considered a defacto sequence index forming the backbone of a number of sequence algorithms.

The Sequence services module in BODHI supports these powerful sequence index structures, in a scaleable form, as *Persistent Suffix Trees* which can be used to accelerate a large class of sequence processing queries and sequence similarity searching [53]. As part of this functionality, this module integrates a novel specialized buffer management policy for persistent suffix-trees, called TOP-Q (topic of Chapter 5), and an optimized storage scheme called STELLAR (focus of Chapter 6).

3.3.4 Query Processor

The core of the query processor is *λ-DB*, a freely available query processor and optimizer for ODL/OQL [38]. λ-DB uses monoid comprehension calculus, which is a generalization of list comprehension in functional languages, to represent queries in an intermediate form. The monoid algebra, similar to the nested-relational algebra, serves to translate between comprehensions and physical plans and the algebraic operators. The translations from the calculus form to the physical plan is done through a combination of cost-based as well as rule-based optimization phases, which involve query rewriting, operator reordering and physical operator selection, similar to relational query processing engines.

The query language for BODHI is based on the OQL recommendation from the ODMG standard [21], which has declarative query syntax closely resembling the standard SQL. However, the basic OQL standard does not provide integrated support for primitives and operators from spatial and sequence domains. The OQL syntax in BODHI is enhanced with addition of new operators such as `OVERLAPS`, `ADJACENT`, `BLAST`, etc. With the help of these extensions, we can express the multi-domain query, Query 1, described in the Introduction chapter, as follows:

```
SELECT species2.name FROM
  species1 IN PlantSpecies, species2 IN PlantSpecies,
  dna1 IN species1.DNAEntries, dna2 IN species2.DNAEntries
WHERE
  species1.name = "Michelia-champa" AND
  species1.flowerchar.inflochar = species2.flowerchar.inflochar AND
  species1.georegion OVERLAPS species2.georegion AND
  dna1 BLAST dna2 ≤ 70;
```

The above query illustrates the use of object relationship path-based joins (species.flowerchar.inflochar), geometric predicates on spatial objects (**OVERLAPS**), as well as sequence similarity search functionality (in the form of **BLAST**). The BLAST algorithm provides two parameters on which the output can be filtered – the alignment score, which is illustrated in the above query, and the *p-value*, the estimated statistical significance of the alignment. It should also be noted that the sequences that are involved in the BLAST similarity search computation can be filtered by providing additional predicates on the sequence attributes. It is worth noting that, our query syntax of BLAST homology searching closely resembles the proposed SQL extensions by Oracle-10g [116].

Along with the extensions to the query language, the query optimizer is also significantly extended to generate efficient plans for predicates involving these special operators. The query processor contains, in addition to the techniques available in generic database systems, specialized optimization schemes for:

- Spatial operators, when spatial indexes are available on predicate attributes.
- Relationship path traversals.
- Queries over a type hierarchy of the data model.

In addition, the presence of user defined methods in the synthesized object types (such as, the *Area* function of a Polygon object), forms an obstacle in optimal plan generation, since their costs are not directly available to the query optimizer. A variety of strategies for handling this situation have been proposed in the literature [52, 69]. In BODHI, we have extended the ODL to allow optional definition of cost functions, and functionally equivalent methods. These extensions enable the cost-based optimizer to compute the costs associated with each of the equivalent methods, before choosing the best execution strategy. In addition, the ability to define index on derived values, or return values of class methods, is also provided to further improve the query evaluations.

3.3.5 Client Interface Framework and User Interface

Although BODHI provides a powerful querying language, it is cumbersome for the intended end-users of the system, namely, the biodiversity researchers, to interact through the OQL syntax. Typically these domain-scientists do not have any experience of using such languages and are reluctant to learn them. Therefore, it is essential to provide a user interface that not only renders the results of the query graphically, but also allows construction of complex OQL queries through a simple form-based interface.

We have developed a simple framework, using a host of technologies including HTML Forms, CGI scripting, JavaScript and Java, that addresses this need. The query interface is provided as a form that the users can fill to build OQL queries visually. Currently, it is capable of building complex OQL statements which involve multiple levels of joins, spatial join predicates like OVERLAPS, CONTAINS and INTERSECTS, and sequence similarity through BLAST. Figure 3.5 provides a snapshot of the query interface of the BODHI system.

In addition to the input interface, the results of the query are transformed into an XML tagged format, which can be rendered using the visualization metaphor of choice. Figure 3.6 provides a sample XML tagged output generated by BODHI.

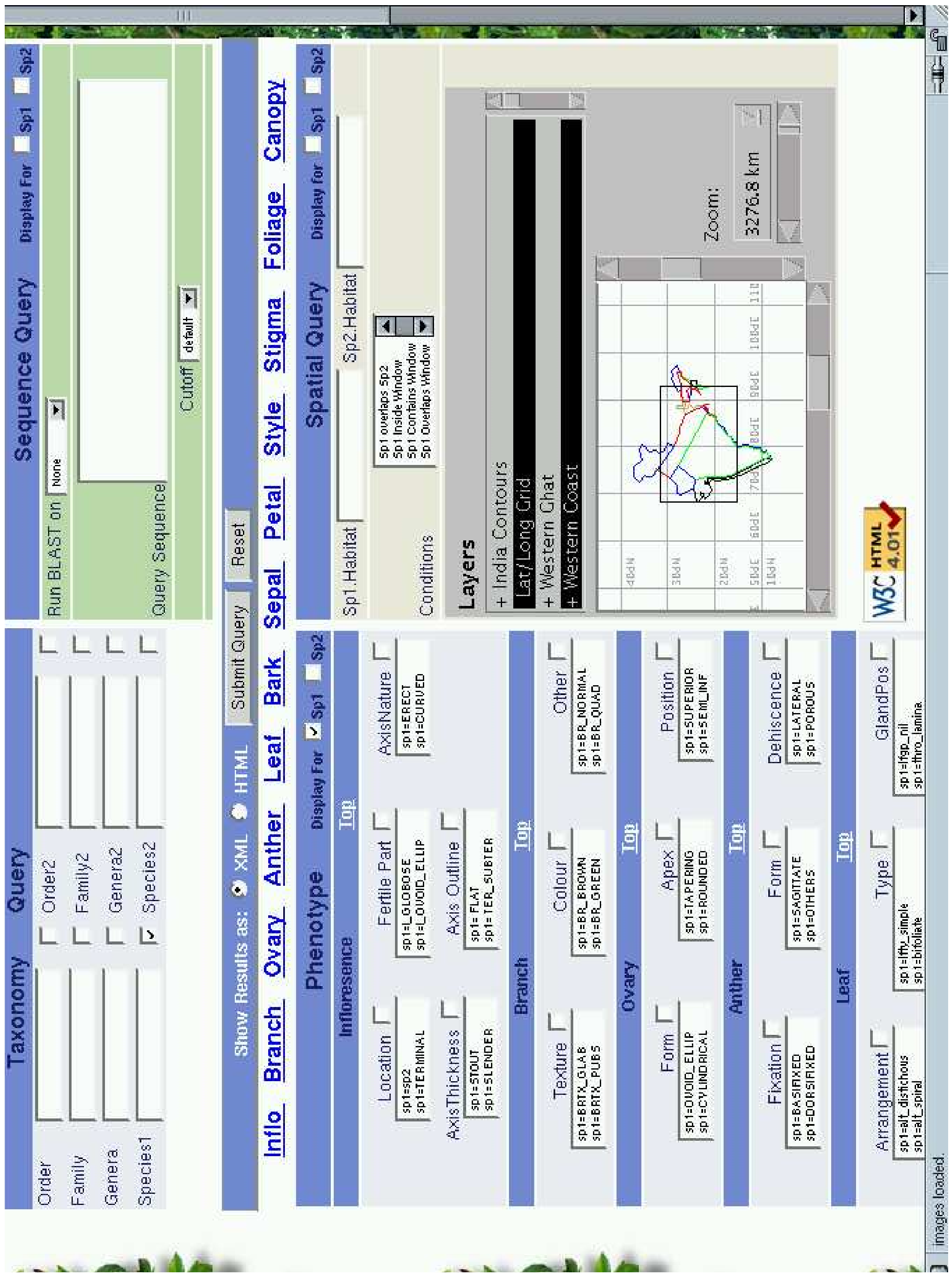
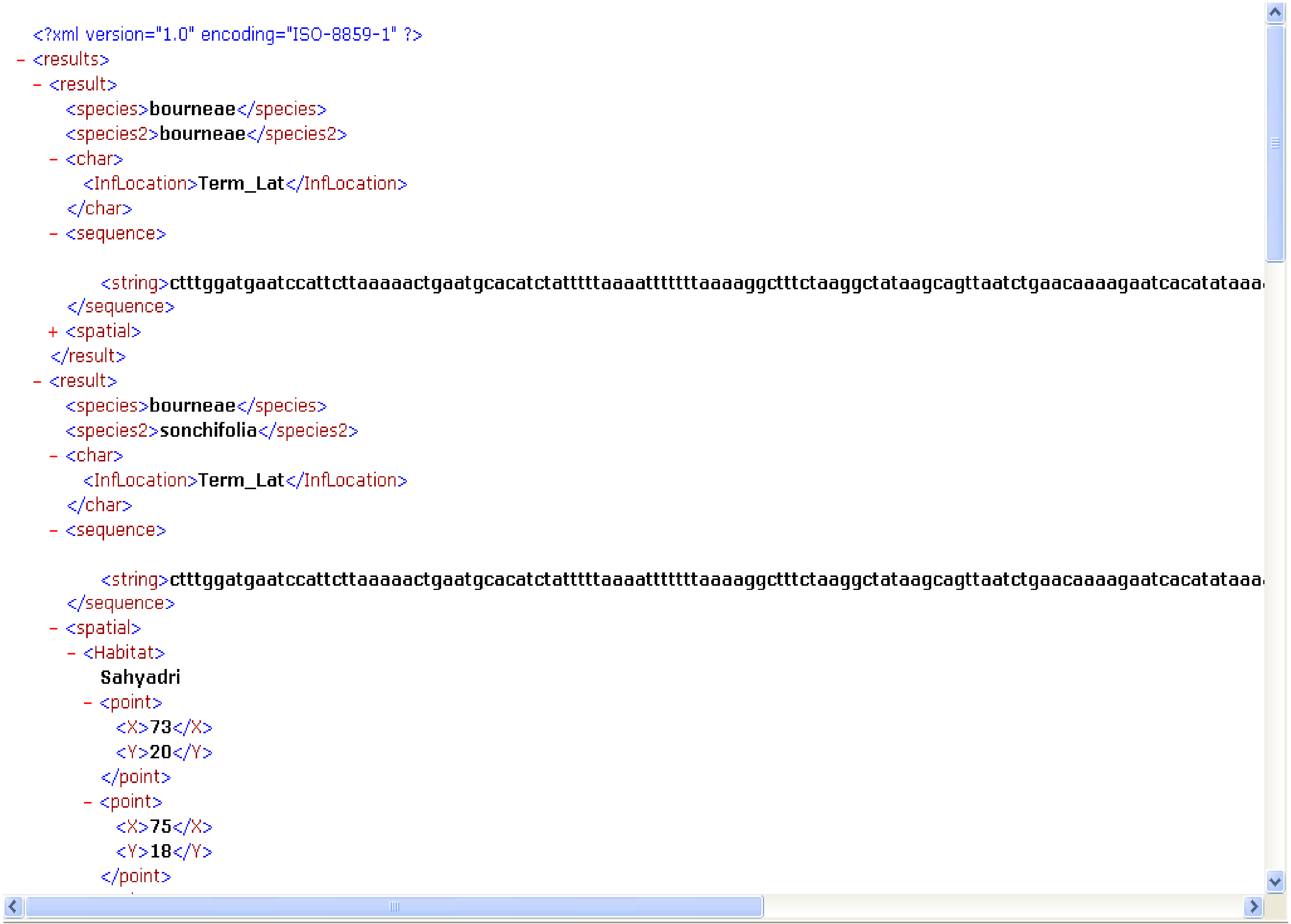


Figure 3.5: Snapshot of the Query Interface



```
  <?xml version="1.0" encoding="ISO-8859-1" ?>
- <results>
  - <result>
      <species>bourneae</species>
      <species2>bourneae</species2>
    - <char>
        <InfLocation>Term_Lat</InfLocation>
      </char>
    - <sequence>

          <string>ctttggatgaatccattcttaaaaactgaatgcacatctatttttaaaatttttttaaaaggctttctaaggctataagcagttaatctgaacaaaagaatcacatataaa
      </sequence>
    + <spatial>
    </result>
  - <result>
      <species>bourneae</species>
      <species2>sonchifolia</species2>
    - <char>
        <InfLocation>Term_Lat</InfLocation>
      </char>
    - <sequence>

          <string>ctttggatgaatccattcttaaaaactgaatgcacatctatttttaaaatttttttaaaaggctttctaaggctataagcagttaatctgaacaaaagaatcacatataaa
      </sequence>
    - <spatial>
      - <Habitat>
          Sahyadri
        - <point>
            <X>73</X>
            <Y>20</Y>
          </point>
        - <point>
            <X>75</X>
            <Y>18</Y>
          </point>
```

Figure 3.6: A Sample XML Output from BODHI

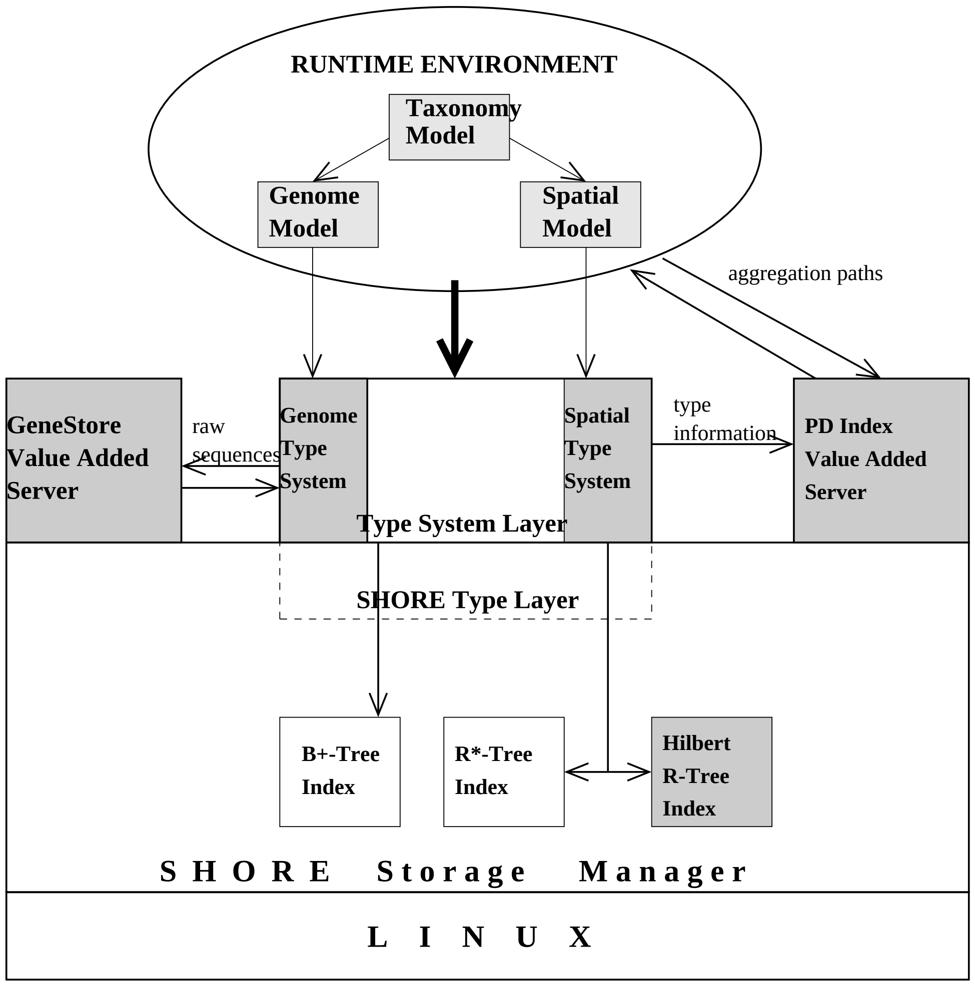


Figure 3.7: BODHI: Implementation Schematic

3.4 Implementation

In any large system such as BODHI, performance gains are obtained not only by the choice of supported access structures themselves, but also by their careful placement in the implementation. One option is to achieve performance improvements by supporting every feature of the system at the lowest level – for example, by implementing at the Shore storage manager level. However, this becomes a huge effort to extend and improve the system by addition of new basic types, new access structures, etc. At the same time, if we provide all the additional features at layers external to the storage manager then the overall performance could suffer. Therefore, we considered these two competing requirements of the system carefully while placing the implementation of the services, and aimed to optimize extensibility while minimizing the performance overhead on the system.

One of the strong features of the Shore storage manager is the presence of a framework to extend the functionality of the server, called Value Added Server (VAS) framework. This feature has been utilized in BODHI to provide additional database server-side features including path-dictionary index, and genome sequence storage and retrieval algorithms.

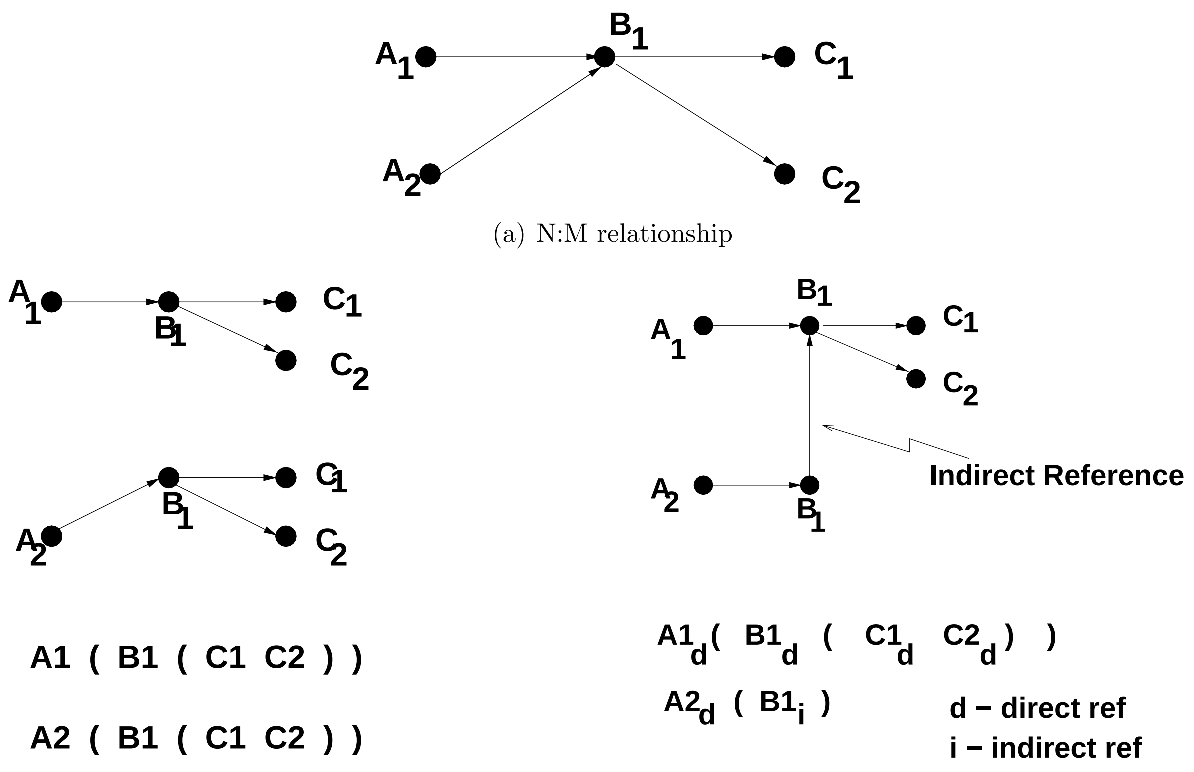
The schematic in Figure 3.7 shows the placement of various components of BODHI in the overall system implementation. The gray filled boxes in the figure indicate the significant enhancements and additional features added to the components used in BODHI. In the rest of this section, we describe the implementation rationale in the context of each service module.

3.4.1 Object Services

As mentioned previously, this module bundles the Path-Dictionary and Multi-key Type indexes over object aggregation and type hierarchies, respectively. The Path-Dictionary structure is implemented as a VAS, which maintains the path-dictionary on a data repository – with its own recovery and logging facilities – independent from the main database. This gives the query processor an opportunity to scan the path-dictionary repository *in parallel* to the other data scans active at the same time. Further, the locking overheads are distributed over different storage management threads. The Multi-key Type index, on the other hand, is instantiated as an R^*-Tree, which is available for spatial indexing, with linearized type system as a dimension and each object treated as a "point" in the spatial sense.

Path-dictionary Implementation

While implementing the Path-Dictionary-based indexing (introduced in Section 2.1.1) for aggregation path queries, we found that the index structure as presented in [74] cannot be used in a stream based query processor such as λ-DB, without breaking the pipeline



(a) N:M relationship

(b) Equivalent 1:N relationships with replicated paths

(c) Equivalent 1:N relationships with indirect references

Figure 3.8: Representing N:M relationships

structure and materializing the query results at that join node. We addressed this problem by *inverting* the storage of paths to proceed from the top of the aggregation graph instead of the suggested bottom-up approach. Thus, the s-expression definition for a path $C_1C_2 \ldots C_n$ is modified to

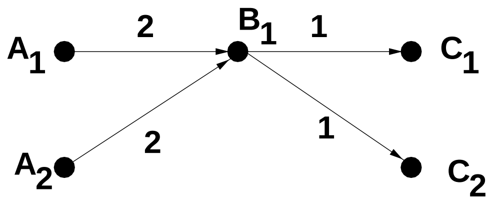
$$S_i = \theta_i(S_{i+1}[, S_{i+1}]) \text{ for } 1 < i \leq n.$$

Note that the above representation is inverse of the original definition of s-expression scheme presented in Section 2.1.1. While this inversion may partially reduce the effectiveness of the path-dictionary, the major benefit of avoiding the huge cost of joins over object extents is retained.

We have extended the implementation given in [74] to support the additional requirements of allowing N:M relationships, as well as bags and sequences in the aggregation path. The main idea behind our extensions for the of N:M relationships is to break them into multiple 1:N relationships. But a straightforward application of this idea introduces complications in maintenance of s-expressions used for compactly representing the object relationships.

Supporting N:M relationships: Consider a N:M relationship between classes A and B, with C representing the remaining downstream objects (successor objects), as shown in Figure 3.8(a). In other words, the figure represents the aggregation path $AB\langle C_1 C_2 \ldots \rangle$, where N:M relationship is between A and B, and C represents the path within the angular brackets, without loss of generality. If we break this into multiple 1:N relationships, the graphs and the corresponding s-expressions look as in Figure 3.8(b). Note the redundancy in the corresponding s-expressions: The children of B_1 are replicated in the s-expressions of both A_1 and A_2. This problem can be solved by using a flag in the entries of the s-expression. This flag denotes whether the entry is a direct reference or an indirect reference. All the descendant entries of an OID will be stored only in the entry which contains direct reference to that OID. This modification is shown in the form of a graph in Figure 3.8(c) with corresponding s-expressions. Note that the suffix for each entry denotes whether it is a direct reference or an indirect reference. Though this modification duplicates (with different flag values) the B_1 entry, we avoid duplicating the children of B_1, thus saving space.

Extensions to support Bags and Sequences: The previous modification works fine for storing ordinary references and sets. But in the presence of bags, further redundancy is possible. The example for this is shown in Figure 3.9(a) – where classes A and B are related through a N:M multi-relationship and C represents the downstream objects from B. The number on the edge from a node, a, to another node, b, denotes the number of times b appeared as a reference in the bag of a. The corresponding s-expressions for this graph using the above implementation are given in Figure 3.9(b). Note that the entry of B_1 is repeated n times in each expression, where n denotes the number of times B_1 is referenced in the parent object. This replication can be eliminated by introducing one more field in the entry of s-expression which stores this replication count. This reduces the storage overhead



(a) N:M relationship with Bags

$A1_d \; (\; B1_d \; (\; C1_d \; C2_d \;) \; B1_i \;)$

$A2_d \; (\; B1_i \; B1_i \;)$

(b) Equivalent 1:N relationships with indirect references

$A1_{<d1>} \; (\; B1_{<d2>} \; (\; C1_{<d1>} \; C2_{<d1>} \;) \;)$

$A2_{<d1>} \; (\; B1_{<i2>} \;)$

(c) Equivalent 1:N relationships with indirect references and reference counts

Figure 3.9: Representing N:M Relationships in presence of Bags

for storing bags since OIDs are not duplicated. The s-expressions with this modification are shown in Figure 3.9(c). The implementation also supports sequences by maintaining the order of the children of a given parent in the s-expressions.

3.4.2 Spatial Services

In addition to the R^*-Tree provided by the Shore storage manager, the spatial services module provides the Hilbert R-Tree which is intended for use with highly dynamic spatial workloads. This index could be implemented as a VAS external to the database, utilizing the Shore SM interface which allows introducing new logical index structures. With this approach, however, no page-level storage control is provided, thereby making it infeasible to implement index structures such as the Hilbert R-Tree that rely on physical packing of data for performance benefits. We were thus forced to implement the Hilbert R-Tree by refactoring the existing R^*-Tree implementation.

We had the option of implementing the spatial type system, illustrated in Figure 3.3, either as part of the basic type system (similar to the support of types like integers, strings, references, etc.) or at the same level as a user defined type system. In the former approach, we do gain the storage efficiency and low object creation overhead, but we lack the extensibility and ease of implementation available in the latter approach. The final choice was to go for an extensible type system, that is, to provide the spatial type system (along with sequence type system – discussed below), as a user level library which can be modified and extended by the database administrator without having to work on the storage manager layers.

3.4.3 Sequence Services

The type system of the Sequence Services, consisting of *DNA* and *Protein* types, are provided in the same way as the spatial types, which we have described above. In addition, the DNA sequence type has extra requirements for its storage. The DNA sequences are usually very long – few thousands to millions of basepairs, and consist of only 4 symbols. Instead of storing them as character strings, we store them in a compressed form and perform queries over the compressed records rather than on the character strings. The efficient storage of the raw sequences is implemented as a separate VAS which provides advantages similar to those mentioned in the Path-Dictionary implementation.

Implementation of BLAST

The BLAST algorithm, first described in [1], has evolved over years into a powerful suite of tools for biological sequence analysis. Further, the BLAST software is not typically designed to work with a fixed memory budget – it utilizes the full virtual memory space available in the system. In BODHI, we implemented the BLAST algorithm in its original form (i.e., BLAST version 1.0), without associated components such as filters for repeat and low complexity regions, and to work within the buffer space limitation specified for the database instance.

In the first phase of the algorithm, we build an inverted index over $|Q| - W + 1$ substrings of length W over the query string Q. Using this inverted index, we locate all the initial "*hits*" – W-length exact matches – over the database sequence. These hits are extended in both directions (without introducing gaps) until the weight of the alignment stays within half the maximum weight found so far. Unlike typical implementations of BLAST algorithm, the whole data sequence is not brought into the memory. Only those pages of the data sequence that are accessed during the extension phase are read into the buffer space, and are managed through a global buffer manager using the CLOCK replacement policy [101].

3.4.4 Query Processing

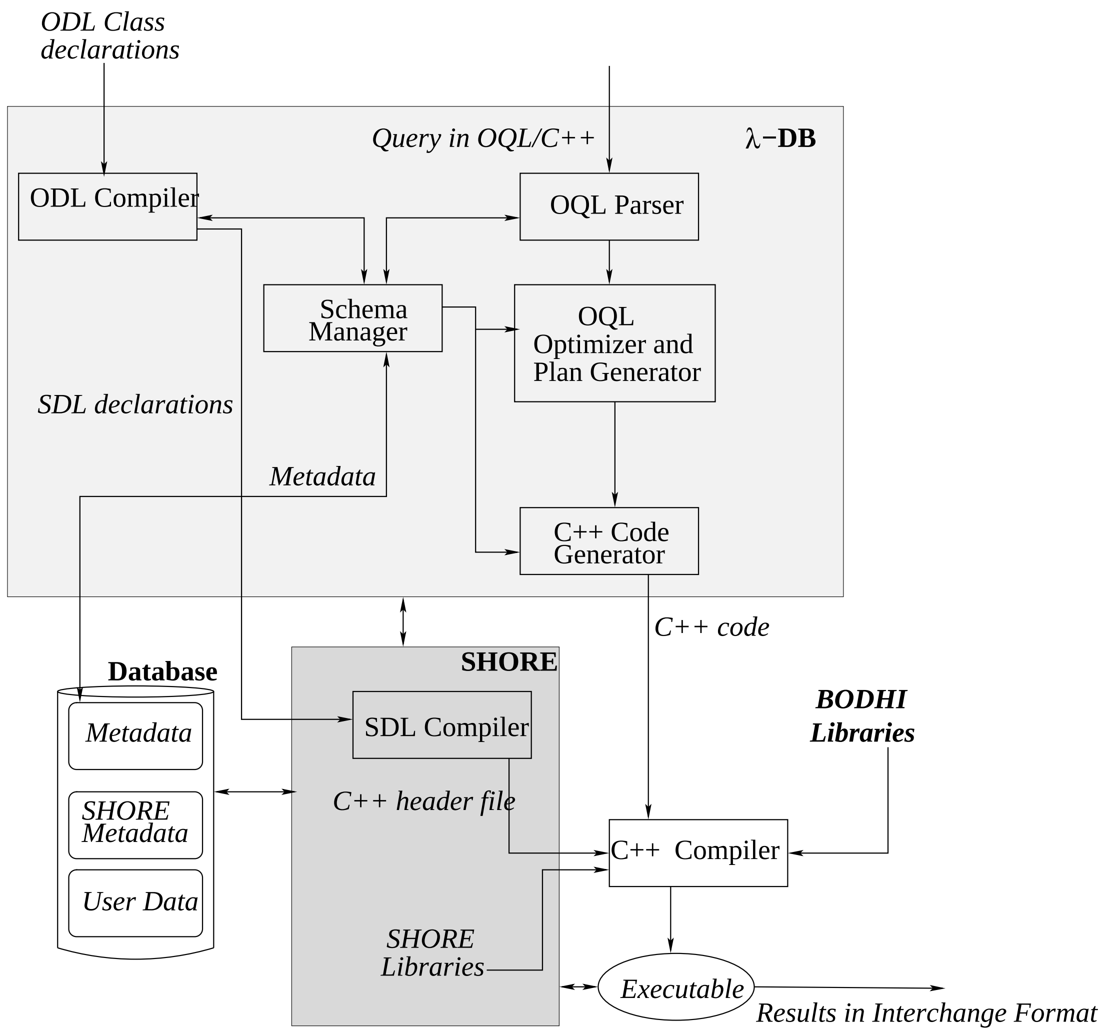


Figure 3.10: Schema Definition and Query Flow in BODHI

The query processor of BODHI integrates the features provided by the service modules through extended ODL/OQL for modeling and querying the database. In addition, it optimizes the queries using the metadata and index information. λ-DB performs all optimizations on the query at compile-time, producing a corresponding executable, resulting in extremely fast query executions. The schematic representing the flow of schema definitions and queries over the database, is illustrated in Figure 3.10.

Schema Definition

The schema is defined, as already mentioned, using ODL – with extensions necessary for BODHI. The schema declarations are first converted into SDL, before getting compiled into an C++ header file. During this phase, the *Schema Manager* of the query processor obtains the metadata needed for typechecking and optimization of queries and maintains it in the database. The implementation part of the schema declaration is abstracted into a C++ code and is available for compilation into a linkable library.

Query Flow

The query strings obtained are type checked, parsed and compiled into C++ code with execution plans generated after incorporating the rules specified in the query optimizer. The type library of spatial and sequence data primitives and the implementations of various operations defined over them, which are precompiled into linkable libraries and header files, are linked to generate the executable of the query.

This executable contains implementations for interacting with the Shore storage manager, Value Added Servers of Object and Sequence Services, and the implementation needed for transforming query results into interchangeable format.

3.5 Conclusions

In this chapter, we presented the design and implementation details of the BODHI database system that addresses a number of datatype integration and performance issues arising in the biodiversity information management. To the best of our knowledge, BODHI is the first system to provide an integrated view from the molecular to the organism-level information, including taxonomic data, spatial layouts and genomic sequences.

BODHI is operational, completely free and is built around publicly available software components and commodity hardware. In order to provide efficient access to different data types, BODHI incorporates a variety of indexing strategies taken from the recent research literature. Further, BODHI is equipped with a specialized sequence indexing solution in the form of a *persistent suffix-tree*, that helps to perform many biological sequence processing tasks efficiently.

Chapter 4

Background on Suffix-Trees

NOTE: In this chapter, for the purpose of completeness, we briefly present background material on suffix-tree structures. An excellent survey of suffix-trees and their biological applications is available in the textbook written by D. Gusfield [53]. Readers familiar with suffix-trees can skip this chapter, and directly move on to Chapter 5.

4.1 Introduction

With the advent of high throughput genome sequencing techniques, biological sequence data is being generated at speeds exceeding the growth of modern day computational speeds. As per the latest statistics published from GenBank [42] – the global annotated collection of all publicly available DNA sequences – 37,893,844,733 (*37 billion*) basepairs in 32,549,400 sequence records are deposited in the databank. This is expected to grow exponentially with ever expanding applications of genome sequence analysis. One of the most important computational tasks on this voluminous amount of sequence data is that of efficiently locating all the matches in the database to a given pattern sequence. In a recent survey of bioinformatics practitioners [117], it was reported that more than 50% of their tasks involved sequence similarity search, pattern search and sequence retrieval from sequence databanks such as the GenBank.

In these tasks, the matching criteria could be either *exact* or *inexact* based on a similarity metric. The similarity metric associated with inexact matching, or similarity matching, is typically based on special cost measures such as BLOSUM [59] or PAM [28] family of costs for proteins and weighted edit-distance costing for DNA. Clearly, the choice of metric to use will be highly specific to the biological task at hand. Due to these non-traditional search requirements over unprecedented scale of sequence data, handling biological sequences efficiently has become an important challenge for database research.

The area of bioinformatics, which has developed almost independently of database research, has considered the *suffix-tree* data structure (along with its variants such as *suffix-array*, *DAWG*, etc.) as the defacto preprocessing of the genomic sequence. This is mainly due to the adaptability of suffix-tree to solve many sequence processing problems that are otherwise computationally extremely hard to solve (see [53] for a collection of such problems). In addition, suffix-trees have linear time and space complexity, which make them attractive for use with large scale sequence processing tasks.

A suffix-tree is a data structure that exposes the internal structure of a sequence in a deeper way than any other datastructure such as inverted index. In this chapter, we provide a brief introduction to the suffix-tree structure, their construction, and search algorithms over them.

4.2 Suffix-Tree

Let $S = s_1 s_2 \ldots s_n$ be a sequence of length n with each s_i drawn from an alphabet Σ. A *substring* of the string S is a string $S[i \ldots j] = s_i s_{i+1} \ldots s_j$ for some $0 < i \leq j < n$. A *suffix* of the string S is a substring such that $j = n$ – *i.e.*, it is a part of the string starting at any location, i, in the string continuing upto the end of the string. We represent a suffix starting at position i as S_i. Thus, there are exactly n suffixes from a string of length n, one for each position in the string.

Definition 1 *A suffix-tree T_S for a n-character string S is a rooted directed tree with exactly n leaves numbered 1 to n. Each internal node, other than the root, has at least*

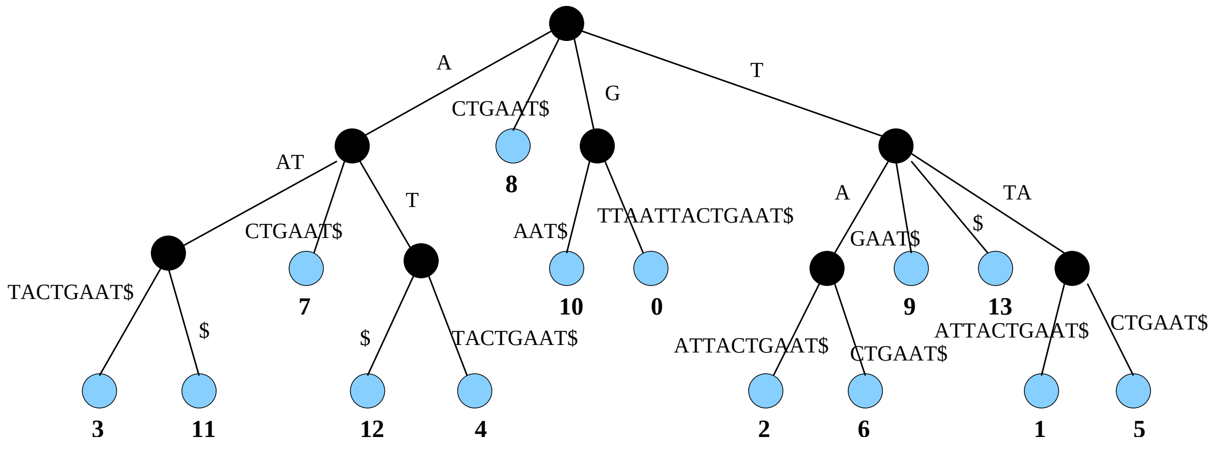


Figure 4.1: Suffix-tree over a DNA fragment `GTTAATTACTGAAT$`

two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix-tree is that for any leaf numbered i, the concatenation of the edge-labels on the path from the root to that leaf exactly spells out the suffix of S that starts at position i. That is, it spells out S_i. □

Note that the above definition does not *guarantee* that a suffix-tree exists for every string S. If there is a suffix S_i that *exactly* matches another *substring*, $S[j \dots k]$ for $j \neq i$, then S_i ends at a non-leaf. In order to overcome this, a *delimiter* symbol, denoted by \$, is concatenated with the string. It is assumed that \$ does not appear anywhere else in the string, and $\$ \notin \Sigma$. With this assumption, it is guaranteed that there is a unique suffix-tree for every string $S\$$. The leaf node corresponding to the i-th suffix, S_i, is represented as l_i. An internal node, v, has an associated length $L(v)$, which is the sum of edge lengths on the path from root to v. We represent by $\sigma(v)$, the string at v, to represent the substring $S[i..i + L(v)]$, where l_i is any leaf under v.

The suffix-tree for a DNA fragment `GTTAATTACTGAAT$` is shown in Figure 4.1. The dark nodes are the internal nodes and the lightly shaded nodes are the leaf nodes. Each edge has an associated label, which is a substring of the string $S\$$, and entry under each leaf node is the index i associated with the suffix corresponding to the leaf node.

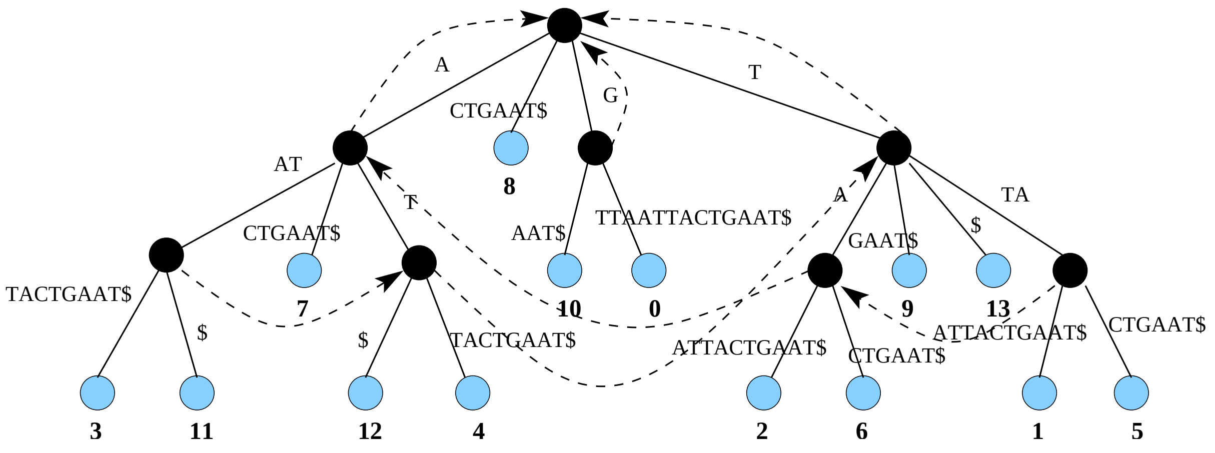


Figure 4.2: Linked suffix-tree (LST) over a DNA fragment `GTTAATTACTGAAT$`

4.2.1 Suffix-Links

Although the definition given above is the commonly used one for suffix-trees, it does not incorporate one significant structural augmentation to the suffix-tree – namely, the notion of *suffix-links*. In practice, the suffix-trees are augmented with additional edges called *suffix-links* that are necessary to achieve their linear time construction and also to significantly enhance the subsequent string searches. Suffix-links are edges (or pointers) that span across the suffix-tree, between two internal nodes which may not be related through a parent-sibling relationship.

Definition 2 *Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a possibly empty substring. For an internal node v with path-label $x\alpha$, if there is another node s_v with path-label α, then a pointer/edge from v to s_v is called a* suffix-link. $\square$

The suffix-link of the root of a suffix-tree is defined to be pointing to itself. Other than this, the suffix-links are well defined for *all* the internal nodes [53]. And, $sl(.)$ – the entire set of suffix-links, forms a tree rooted at the root of T_S, with the depth of any node v in this $sl(.)$ tree being $L(v)$. Figure 4.2 illustrates the suffix-tree with suffix-links, built over a genome fragment `GTTAATTACTGAAT$`. The dotted lines between internal nodes of the tree are the suffix-links, with the direction of the arrow indicating the pointer direction.

The suffix-links, in the present form, were first introduced by McCreight [80] and since then they are implicitly assumed to be present in the suffix-tree. In addition to the linear time construction, the presence of these links enable a much richer set of traversals over the suffix-tree resulting in many high-speed search algorithms [22, 125]. On the other hand, suffix-links are also considered a source of additional space overhead, and more significantly, a reason for poor locality properties of suffix-tree construction and search algorithms. Therefore, there are some proposals [47, 61, 123] which resort to quadratic time construction of suffix-trees by completely dispensing with the suffix-links.

In this thesis, we distinguish between these two structural variants of suffix-trees based on the presence of suffix-links as follows:

Un-linked Suffix-Tree (UST). This is the suffix-tree that strictly adheres to Definition 1. The suffix-links in the UST have been either dropped post-construction, or the tree has been constructed using algorithms that do not need suffix-links.

Linked Suffix-Tree (LST). In contrast to USTs, the LSTs retain the suffix-links in the tree providing much richer traversals across the suffix-tree. We focus mainly on the construction and search performance of persistent version of LSTs. Hence, unless mentioned explicitly, we use the terms LSTs and suffix-trees interchangeably.

4.3 Notation

For ease of reference, Table 4.1 summarizes the terminology associated with suffix-trees, used in this thesis.

4.4 Linear Time Construction of Suffix-Trees

As mentioned previously, suffix-trees can be constructed in time linear in the size of the input string for a fixed alphabet. Many algorithms for constructing the suffix-tree in these time bounds have been proposed [80, 126, 130] – all of them depending on the

S	Sequence of length n
Σ	Finite alphabet of symbols
$\$$	Delimiter symbol such that $\$ \notin \Sigma$
s_i	Symbol at position i in S, drawn from Σ
$S[i \dots j]$	Substring of S starting at position i and length $(j - i + 1)$
S_i	Suffix of the sequence S starting at position i
T_S	Suffix-tree built for the sequence $S\$$
l_i	Leaf in the suffix-tree T_S corresponding to the suffix S_i
$L(v)$	Path length of a node v, the sum of edge lengths on the path from root to v
$\sigma(v)$	Substring $S[i \dots i + L(v)]$ associated with node v in the suffix-tree
$sl(v)$	Suffix-link starting from the internal node v

Table 4.1: Notation

availability of suffix-links. In this thesis, we consider the online construction algorithm due to Ukkonen [126] – which we call as OnlineSuffixTree algorithm.

A high level description of OnlineSuffixTree is given in Algorithm 1. The algorithm reads the sequence from left to right, one character at a time, incrementally building the suffix-tree for the string seen so far. During the execution of the algorithm, the labels of leaf-edges extend, while some edges are split and new leaf nodes are introduced to accommodate the new character read from the string. It should be noted that due to the lack of the delimiter symbol \$ in the middle of the sequence, the intermediate suffix-trees are no longer guaranteed to have a one-to-one mapping with each suffix of the string read so far. Hence, these intermediate suffix-trees are called *implicit suffix-trees.* Once the delimiter symbol is read at the end of the sequence, the algorithm automatically generates the final explicit suffix-tree, as required.

The OnlineSuffixTree can be viewed to consist of two phases in each iteration – a *Locate phase* and an *Insert phase.* If implemented naively, the locate phase would have linear time complexity, resulting in an overall $O(n^3)$ time complexity. Reducing this to a $O(n)$ algorithm is based on speeding up the locate phase (line 6 of Algorithm 1) through the use of the following:

Suffix-links: After the first extension (with $j = 0$) for each value of i, it is possible

```
OnlineSuffixTree ($S[0 \dots n]$)
**Input**
$S[0 \dots n]\$$ : The string to be indexed
**Output**
$T_S$ : The suffix-tree over the string $S\$$
**Complexity**
$O(n)$, where $n$ is the size of the input string.

1: $T_0 \leftarrow$ Implicit suffix-tree for $S[0 \dots 0]$
2: for $i = 0$ to $n$ do
3:   $j \leftarrow 0$
4:   while $j < i + 1$ do
5:     {LOCATE PHASE}
6:     Locate $\beta = S[j \dots i]$ in $T_i$
7:     {INSERT PHASE}
8:     if $\beta$ ends at a leaf $l_k$ then
9:       Extend $l_k$ by adding $s_{i+1}$
10:    else {$\beta$ ends at an internal node, or the middle of the edge}
11:      if from the end of $\beta$ there is no path labeled $s_{i+1}$ then
12:        $T_{i+1} \leftarrow$ split edge in $T_i$ and add a new leaf
13:      else
14:        $T_{i+1} \leftarrow T_i$ {$\beta s_{i+1}$ already exists in $T_i$}
15:      end if
16:    end if
17:  end while
18: end for
```

Algorithm 1: Online Algorithm of Ukkonen

to quickly locate the node in the suffix-tree where the next extension has to be performed through the use of suffix-links. The steps required are:

1. Locate a node v above the end of $S[j \dots i]$ that has a suffix-link, and let γ denote the string between v and the end of $S[j \dots i]$.
2. Traverse the $sl(v)$ to s_v, and walk down the tree-edges for string γ.
3. Using the extension rules, ensure that the string $S[j \dots i + 1]$ is in the tree.
4. If a new internal node is created, then it is the suffix-link target of a previously created internal node. Create the suffix-link.

Skip-count Technique: In the step 2 above, it is possible to avoid individual character comparisons for the string γ, through the observation that γ must already exist (from a previous step of the algorithm). It is only needed to locate the position where γ ends. As a result, this step can be reduced to simply locating the appropriate branch from each internal node encountered, reaching to the end of the edge and skipping appropriate number of symbols in γ, until the number of symbols remaining in γ is less than the length of the current edge – this is the edge that needs to be split so as to insert $S[j \dots i+1]$.

Early Stop: Finally, if at any stage, the condition in step-14 is reached then we can break out of the while-loop (step 4 through 17), guaranteed that the rest of S[j . . . i+1] entries already present in the tree.

With these three algorithmic optimizations, the locate-phase of the algorithm can be accomplished in an amortized constant time. This immediately gives us an $O(n)$ construction algorithm.

It is also important to note here that although the mechanics of OnlineSuffixTree seemingly differ from an earlier suffix-tree construction algorithm due to McCreight [80], it has been shown [46] that these two algorithms are closely related to each other. In fact, they provide a transformation of OnlineSuffixTree into McCreight's algorithm by modifying the control structures of the algorithm, but leaving the sequence of tree constructing operations invariant. Hence, the results presented in this thesis are equally applicable in the case of linear time construction of suffix-tree using McCreight's algorithm as well.

4.5 Searching over the Suffix-Tree

Suffix-trees provide most efficient solutions to a myriad of string processing problems [4]. The fundamental query of whether a given pattern sequence Q occurs in a sequence S can be answered in $O(|Q|)$ steps – independent of the length of S, once S has been preprocessed into a suffix-tree T_S. A long list of string processing problems that are especially relevant for biological sequence processing is available in [53].

As discussed earlier in Section 4.1, the most important sequence processing task over biological sequence databanks is that of approximate pattern location or similarity searching. The dynamic programming approach [111] provides the general solution to this problem, but is computationally expensive, taking $O(|Q| * |S|)$. This is clearly highly impractical for genome scale databases. A class of algorithms, called *Filtering algorithms*, are designed based on the observation that the dynamic programming methods spend much more time verifying the non-existence of approximate matches in the database than on computing the matches. Some of the efficient algorithms in this class [22, 25, 125] exploit the power of LSTs in a critical way to achieve linear and sub-linear time complexities, under both simple edit-distance model as well as *under general scoring models*. There are many bioinformatics tools, prominent among them being MUMmer [29], that use suffix-trees for similar purposes.

4.5.1 Locating Maximal Common Sub-strings

One of the most popular approximate search algorithms that exploit suffix-trees was proposed by Chang and Lawler [22]. Their algorithm utilizes the suffix-tree to quickly identify all the *maximally matching substrings* between the query pattern, Q, and the database sequence, S, and then use this information to filter out large sections of wasteful comparisons that dominate the computational cost. The user given parameter to this process is the lower-bound on the length of the match, λ, to reduce the number of verification steps that need to be performed.

Definition 3 (Maximal Common-substring Search) *Given a database sequence S, and a query sequence Q, locate, for each position i of Q, the longest matching substring $Q[i \dots i+j]$, that appears somewhere in S. In practice, it is desired that only matches that satisfy a user-defined minimum threshold length, λ, are reported.*

In other words, for each i, $1 \leq i \leq |Q|$, locate all (i,j,k) triples where $j = \max j'$ such that $Q[i \dots i+j'] = S[k \dots k+j']$ and $Q[i+j'+1] \neq S[k'+j'+1]$ and $j' \geq \lambda$.

□

MaximalSubstringSearch (S, $\mathcal{T}$, Q, λ)
Input
S : Database sequence
$\mathcal{T}$: Suffix-tree over the database sequence S
Q : Query string
λ : Minimum match-length to be reported
Output
$\mathcal{L} = \{(l, q, d) \mid Q[q \dots q+l] = S[d \dots d+l],\ Q[q+l+1] \neq S[d+l+1],\ l \geq \lambda,$ and l is maximal given $q\}$
Complexity
$O(|Q| + loc)$, where loc is the number of locations of match.

```
1: $v \leftarrow$ root of $\mathcal{T}$; $j \leftarrow 0$; $k \leftarrow 0$; $\mathcal{L} = \phi$
2: for $i = 0$ to $|Q|$ do
3:    $(v', j) \leftarrow$ StepDown$(v, Q[i \dots])$ {$v'$ is the node at which matching
      has stopped, and $j$ is the length of the match}
4:    if $j \geq \lambda$ then
5:       $\mathcal{L} = \mathcal{L}\ \cup$ TraverseSubtree$(v')$
6:    end if
7:    if IsLeaf $(v') = true$ then
8:       $k = v'.edgelen - j$
9:       $v' = v'.parent$
10:   end if
11:   $v = v'.suffixlink$
12:   $v =$ SkipDown$(v, k, Q[i \dots])$ {Use the skip-count trick [53] to tra-
      verse *without* comparisons}
13: end for
```

Algorithm 2: Maximal Common Substring Search

To further illustrate this, consider the database sequence – GTTAATTACTGAAT\$ which has been preprocessed into a suffix-tree shown in Figure 4.2. Now, given a query sequence CTAATGACT, with threshold λ set to 3, the desired common maximal substrings between the database sequence and the query sequence are: $\{\texttt{TAAT}, \texttt{AAT}, \texttt{TGA}, \texttt{ACT}\}$. Note that although CT is a common substring, it is not reported since it does not satisfy the match length restriction.

The basic idea of the LST based algorithm to locate maximally matching substrings, which we call MaximalSubstringSearch, is to locate the first longest match between Q and S by walking down the suffix-tree T_S using symbols of Q – matching them "letter-at-a-time". Subsequent longest matches are found by following the suffix-links and going down the tree at the target of the traversed suffix-link. Note that this algorithm depends heavily on the availability of suffix-links. A brief pseudo-code of the algorithm is provided in Algorithm 2.

Locating Maximal Common Substrings over UST

In order to locate all the maximal common substrings between S and Q when S has been processed into an UST (Unlinked Suffix-Tree) T'_S, we use the observation that every common substring must result in a prefix match between corresponding suffixes in S and Q. This leads us to the following algorithm – use each suffix of Q to walk down the suffix-tree T'_S from the root node, until either the suffix is completely located or there is a mismatch. If the length matched is greater than the value of λ, then add to the output set, $\mathcal{L}$, all the leaf nodes under the current location. Follow this process for all the suffixes at positions from 0 to $|Q| - \lambda + 1$. We refer to this algorithm as $\mathsf{MSS}_{\mathsf{UST}}$ in the rest of the thesis.

4.6 Implementation of Suffix-Trees

Until recently, suffix-tree indexes were considered as main-memory index structures. As a result, the main concern in suffix-tree implementation has been that of reducing the associated space overheads in order to be able to index larger sequences within a fixed memory budget.

The design issue associated with the implementation of suffix-trees is the choice of representation for the outgoing branches of internal nodes in the tree. The simplest technique is to use an *array* of size $|\Sigma|$ at each internal node. This array is indexed by individual characters of the the alphabet, and contains the pointer to the child node with its edge label beginning with the character indexed at that position. This array allows constant-time access of child-nodes and updates. We term this representation as *array representation* in the rest of the thesis. Although simple to implement, the array representation has not been favored since it could result in a lot of wasted space, with many entries in the array being *null*.

As suggested by McCreight [80], a space efficient alternative is to use a *linked list* of child pointers at an internal node with the internal node storing a single pointer to the head of the linked list. When a new edge from the internal node is added, a pointer to the child node is inserted into the list, and tree edge traversals are implemented by sequentially scanning the linked list to locate the required character. However, this additional factor of traversing a $\Theta(|\Sigma|)$ size linked list can result in adding the $|\Sigma|$ factor to the time bounds of the suffix-tree operations. This representation is termed as *linked-list representation*.

Although other alternatives such as using *balanced binary search trees* and *perfect hash tables* for storing these child pointers at every internal node have been proposed [53], the best in-memory implementation reported in [72] has used a variation on the linked list approach – resulting in suffix-tree size upto 12.69 times the length of the sequence. Hence, most suffix-tree implementations have preferred the linked list approach for their representation.

Chapter 5

High-performance Persistent Suffix-Tree Construction

5.1 Introduction

A unique aspect of suffix-trees is that, unlike traditional database indexes which are typically a fraction of the database contents, their size is larger than the underlying sequence data. In fact, standard implementations of suffix-trees require in excess of *an order of magnitude* more space than the indexed data! As a case in point, the entire 3 Gbp of Human Genome is fully representable in about 1 GB (with each DNA symbol represented with 2-bits), whereas the corresponding most space-economical suffix-tree occupies close to 38 GB ($= 3\ Gbp \times 12.69\ bytes$).[1] That is, it is straightforward to host the sequence data in main memory, but the suffix-tree itself needs to be disk-resident!

Thus, it becomes untenable to consider a suffix-tree residing fully in memory, indexing an ever growing sequence corpus such as the GenBank maintained by NCBI. An obvious solution to handle this space problem is to maintain the suffix-tree index on disk. Unfortunately, due to seemingly random traversals induced by the linear-time construction algorithms, resulting in unacceptably high I/O costs, the folk wisdom is that disk based

[1]Although in the case of USTs further space optimizations are possible [110], resulting suffix-tree is still almost an order of magnitude larger – ≈ 25 GB index-size for the Human Genome.

implementations of suffix-trees are unviable [89].

In order to overcome this infamous "memory bottleneck" [36] of persistent suffix-tree construction, there are two possible approaches:

1. The suffix-tree construction algorithm and its structure could be modified to make it more suitable for on-disk implementation.

2. Tune the parameters of the environment in which suffix-tree is implemented, *without modifying either the structure or the construction algorithm.*

The former approach primarily consists of completely abandoning the use of *suffix-links* that are crucial in obtaining linear-time construction of suffix-trees. This enables efficient batch-wise construction techniques, although with theoretically quadratic worst case time complexity. The works of Hunt et al. [61] and Tata et al. [123] take this approach. However, due to the resulting structure without suffix-links, some of the fast approximate string processing algorithms that make use of suffix-links, such as computing *matching statistics* [22], are rendered unusable.

In the work presented in this chapter, we take the second approach, and identify the parameters that affect online persistent suffix-tree construction and quantify their impact. Specifically, we make the following contributions:

First, we propose a novel buffer management strategy called TOP-Q, that takes into account the behavior of traversals induced by the suffix-tree construction. This strategy exploits the path length invariant (defined in Chapter 4) of suffix-tree nodes and hence has almost no computational overhead for suffix-tree node access. The datastructures associated with TOP-Q are extremely simple and are easy to maintain.

Second, we study the choice of suffix-tree implementation on the performance of its construction. The previous work on suffix-tree representations [80, 72] had noted the superiority of linked-list representation of suffix-tree nodes – due to its space optimality over a variety of datasets. We show that this approach of suffix-tree implementation is extremely expensive in terms of disk I/O (hence in terms of construction time). Instead, we show that a simpler and often neglected array representation of suffix-tree edges provides far superior performance.

Third, we present an empirical study of a variety of buffer management policies in the context of suffix-tree construction in terms of their buffer space utilization. We show through the results of this empirical analysis that TOP-Q outperforms popular buffer management strategies such as LRU, and the highly sophisticated LRU(2) [93], implemented using the high performance equivalent 2Q algorithm [64].

Finally, we describe an implementation of the TOP-Q policy on top of the *CLOCK* replacement policy, typically used for buffer management in database systems [101]. Specifically, we present our implementation of TOP-Q based suffix-tree indexing within the BODHI system.

In our experiments, we use the OnlineSuffixTree algorithm, described in Chapter 4. However, we also show that the results remain the *same* even in the context of the construction using McCreight's algorithm. This is not very surprising, since, in their classical paper [46], Giegerich and Kurtz have shown that these two algorithms have highly similar structural properties. Our evaluation testbed consists of a variety of real DNA sequences, and a synthetic symmetric Bernoulli sequence over a 4 character alphabet.

5.1.1 Organization

The remainder of the chapter is organized as follows: A study of node access patterns observed during suffix-tree construction is described in Section 5.2. Then, in Section 5.3, we present observations that help to identify the frequently accessed nodes during construction, which form the basis for the design of TOP-Q. The TOP-Q buffer management policy is described in Section 5.4. In Section 5.5, we describe the two suffix-tree implementation choices (linkedlist and array-based), before describing the evaluation framework in Section 5.6. The experimental results are presented and analysed in Section 5.7. Implementation details of TOP-Q within the BODHI framework are presented in Section 5.8, before concluding in Section 5.9.

5.2 Persistent Suffix-Tree Construction

As described in Section 4.4, the suffix-tree over a string S can be built in a time proportional to the length of the string, and the number of nodes in the resulting suffix-tree is upper bounded by $2n$. However, when we move the suffix-tree construction from memory to disk, these linear bounds no longer reflect reality, since they were obtained with a RAM machine model, where every memory access has the same cost, *irrespective* of its address. On the other hand, access-costs in secondary memory are dependent on the address to which the previous access was made. For example, a long chain of accesses to spatially contiguous addresses (block accesses) could cost much less than fewer but random accesses.

During the construction of a suffix-tree, accesses to nodes are spatially non-contiguous. Specifically, in the locate-phase, already constructed parts of the tree are re-accessed many times via suffix-links. These traversals are not necessarily spatially local, leading to seemingly random traversals over the tree. The following words of Giegerich and Kurtz [45], typifies the behavior of these algorithms:

"The active suffix creeps through the text like a caterpillar. At the same time, the corresponding active node swings through the tree like a butterfly".

Thus, it is strongly believed that the accesses are random in nature – with no obvious useful patterns discernible from the access traces.

5.3 Locating Preferred Nodes

In this section, we closely analyse the traces of accesses to nodes during online suffix-tree construction, show that some of the nodes are indeed accessed far more frequently than others, and provide a simple observation that helps to identify such nodes during construction of the tree.

Before we proceed to analyse the traces, we note that the nature of accesses that are expected during the construction of the suffix-tree is intricately linked to the stochastic

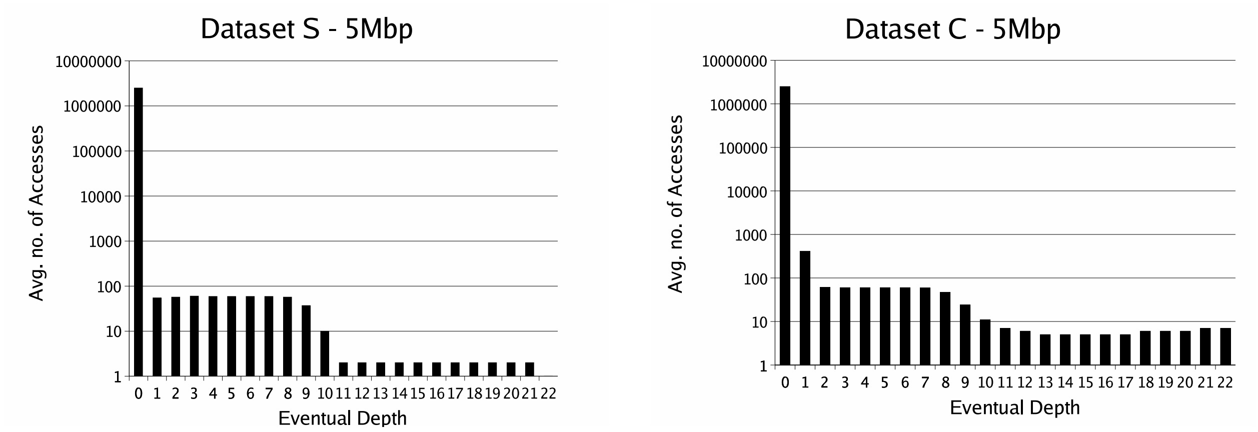


Figure 5.1: Node Access Frequency

properties of the specific sequence at hand. There have been many efforts to classify the sequences based on their stochastic properties [121]. One of the simplest sequence models proposed as an approximation to genome sequences is that of *Bernoulli generators.* In this model, symbols of the alphabet are drawn independently of one another; thus a string can be described as the outcome of a sequence of Bernoulli trials. In addition, if all symbols are drawn with equal probability, then the sequence is called *symmetric*, otherwise, it is *asymmetric.*

Now, consider the internal node access statistics during suffix-tree construction for a symmetric Bernoulli sequence (dataset S) derived from the access traces, shown in Figure 5.1. These provide the correlation between the average number of accesses made to a node and the eventual depth of the node in the tree, illustrating that, during the suffix-tree construction, *nodes higher up in the tree are accessed more number of times than nodes lower in the tree.* This correlation is also evident for suffix-tree construction over real chromosomal sequences (dataset C) as shown in Figure 5.1.

Thus, it seems reasonable to cache the nodes that end up higher in the tree, in order to serve these accesses faster. However, due to the nature of the edge splits during construction, the depth of a node cannot be maintained without propagating the update throughout the subtree under the node.

5.3.1 Estimating the Depth of Internal Nodes

Although the depth of internal nodes cannot be maintained accurately, a simple observation on the structure of the resulting suffix-tree provides us with a means to *estimate* this value efficiently. Considering sequences drawn from a symmetric Bernoulli stochastic model, it is straightforward to see that:

1. Substrings of equal length are equally likely, and every substring forms the prefix of some suffix of S.

2. If S_α is the set of all the suffixes of S which share a common prefix α, then the probability of finding atleast one pair $a_i, a_j \in S_\alpha$, such that they differ in atleast one position within the next s symbols is directly proportional to the value s.

Applying these to the behavior of the suffix-tree during its construction, we get:

Observation 1 *The longer the edge in the suffix-tree of a symmetric Bernoulli sequence, more the likelihood of its being split as the length of the indexed sequence increases.*

And obviously, no edge can be split once it has reached the limiting minimum length of 1. From the above observation, we can infer that a node, in the extreme case, can move further down in the suffix-tree until its incoming edge has only one symbol as its label. Thus $L(.)$ of any internal node forms an *upper bound* on its eventual depth. In addition, this measure of path length is an *invariant* for the node, which results in easy maintenance of this information during the construction of the suffix-tree. Hence, for suffix-tree nodes over large sequences, *we can approximate their eventual depth in the tree by their path length.*

5.3.2 Impact of Asymmetric Distribution

In deriving the above approximation to the eventual depth of a node, we made the assumption that the sequence is drawn from a symmetric Bernoulli model. However, it is unlikely that any real-world DNA sequence would conform to this restrictive model. The asymmetry of real-life distribution results in substrings containing a larger proportion of frequent symbols, having higher probability of occurrence than other substrings of the same length. This implies that if a node v has its label $\sigma(v)$ containing smaller number of frequent symbols, it has lower probability of being split – resulting in a possible over-estimation of its eventual depth by its path length $L(v)$.

The graphs in Figure 5.2 show, for the four DNA datasets used in our experiments, the error in estimation of eventual depth of a node vis-a-vis the actual depth of the node, as well as the corresponding number of nodes at each depth of the tree. These graphs were obtained after processing 5Mbp of each of the datasets. The figure shows the statistics for nodes only upto a depth of 20, since the number of the internal nodes at depths greater than 20 is too small to make any impact although the error in estimation of eventual depth is quite large for such nodes.

The average error at each depth was computed as the arithmetic mean of $(L(.) - Depth(.))$ for all nodes at that depth. Note that since $L(.)$ forms an upperbound on the depth of the node, this value is always positive. The following points have to be noted with respect to these graphs:

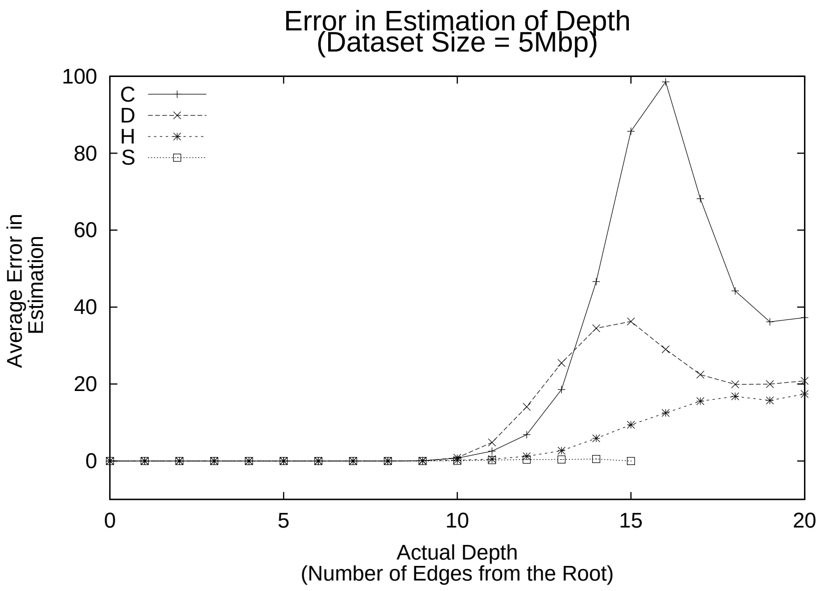
1. For all the datasets, path length corresponds *exactly* to the depth of the node up to a certain value of actual depth, which is dependent on the length of the sequence processed. As shown in the graphs, this value is 10, after processing 5Mbp of each of the datasets.

2. The number of nodes peaks at a depth of 11, at which point the error in the estimation of depth is small for all the datasets.

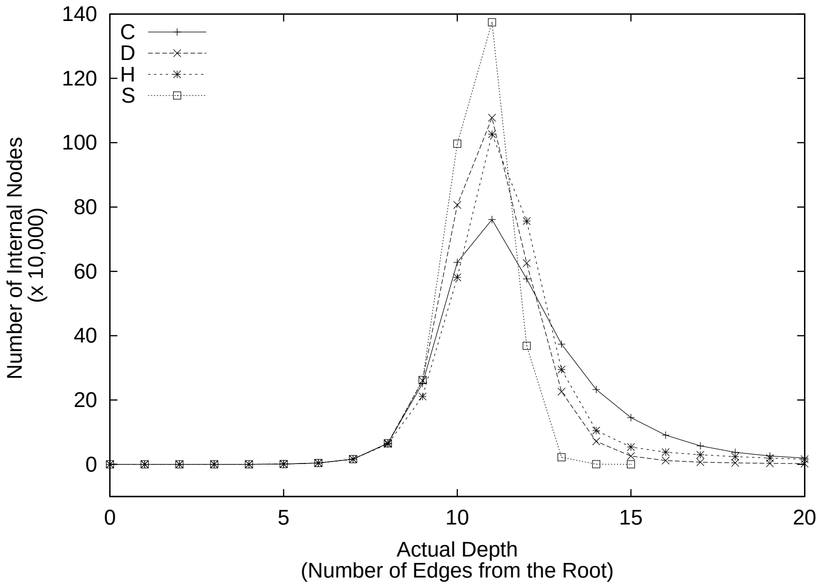
3. For the dataset S, estimates are accurate throughout providing empirical evidence for the soundness of Observation 1.

4. The worst estimation errors are with dataset C, which has a highly skewed distribution of basepairs. However, a majority of nodes in the tree occur within depth 14, where errors are not too large.

5. The depth of a node approaches its path length with the increase in length of



(a) Error in Depth Estimation



(b) Distribution of Nodes

Figure 5.2: Relative Impact of Errors in Depth Estimation

the sequence indexed. Therefore, the estimation errors continue to decrease as we process sequences of greater length.

In summary, these graphs clearly demonstrate that even for datasets that deviate from the symmetric Bernoulli model, the impact of errors incurred by the proposed approximation to eventual depth is not very significant, and the quality of approximation improves with the increase in the length of the sequence.

5.4 Design of TOP-Q

Before describing the TOP-Q buffering strategy, we present the design of TOP, a simpler version of TOP-Q, which exploits the observation that higher number of accesses are to the nodes that are eventually higher in the tree, as well as the approximation to the eventual depth presented above.

We consider the situation where each disk-page contains a collection of nodes of the suffix-tree – either internal or leaf, but not a mix of both. In order to minimize the storage cost of maintaining the path length, each disk page contains an associated path length, which is the average of the path lengths of all nodes packed in it. Each disk-page is completely packed with nodes as they created, and since the path length for each node is an invariant, the path length of the page can also be computed at the time it is committed to the disk.

Definition 4 (TOP Ranking) *Let $\boldsymbol{d_i}$ be the average of path lengths of all nodes in a disk-page $\boldsymbol{b_i}$. Then, we define a ranking function $\boldsymbol{R}$, over disk-pages as follows:*

$$\boldsymbol{R(b_i) > R(b_j)} \text{ \textit{if} } \boldsymbol{d_i < d_j}$$

We call the ranking generated as the **TOP Ranking**.

Definition 5 (TOP Buffering Algorithm) *The TOP algorithm specifies a page replacement policy when a buffer is needed for a new page to be read from the disk: the page $\boldsymbol{b}$ to be dropped (i.e., one chosen as the replacement victim) is the one whose TOP rank* *is the smallest. Only time this choice is ambiguous is when more than one page has same* ***R*** *value. In this case, we can resolve the tie using a subsidiary policy. We follow a simple policy of choosing the page whose creation time is oldest among the tied candidates for eviction.*

We call this buffering policy as *TOP*, to indicate that it tries to retain those pages of the suffix-tree that are estimated to be top pages i.e., pages containing the top nodes of the tree.

5.4.1 Accommodating Correlated Accesses

Although the TOP buffering policy exploits the preferential access to the nodes with lower path lengths, it ignores the presence of correlated access patterns exhibited by the suffix-tree construction. We provide below an example situation in the construction process, where this makes a impact on the performance of TOP.

The construction proceeds by splitting an edge, introducing a new branching node and a leaf node at that location, and filling in suffix-link pointers if needed. Every node stores the details of its incoming edge, i.e., $(start, end)$ indexes into the sequence and the length and label of the edge. When an edge $p(v) \rightarrow v$, is about to be split, the following actions are performed:

1. create a new branching node, v', and a leaf node l,
2. set the incoming edge details for both v' and l,
3. update the incoming edge details for v (the edge length is shortened, and the *start* value and corresponding edge label are changed), and finally,
4. v' is set as the child of $p(v)$ in place of v, and v is now located under v'.

In addition, by the nature of the algorithm, only v has an active reference to it, and $p(v)$ is to be accessed through the parent pointer available with v. Therefore, $p(v)$ is not guaranteed to be pinned in memory during this process.

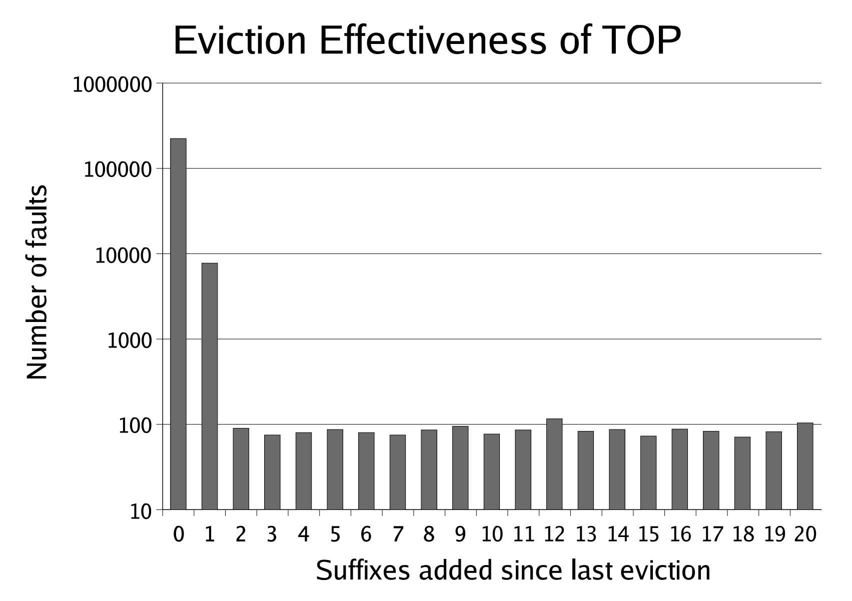


Figure 5.3: Correlated Accesses in TOP

As the construction progresses, the internal nodes have ever-increasing path lengths associated with them. Therefore, the TOP policy evicts the pages as soon as they are filled and are not pinned through any active reference, since the internal pages also have larger path length values as the construction proceeds. Therefore, there is a possibility that $p(v)$, more precisely, the page containing $p(v)$, is not retained in the buffer for long.

We evaluated the impact of such correlated accesses to nodes, by measuring the the number of characters processed by the algorithm between the instant a page is evicted and the instant it is next requested, generating a fault on the buffer pool. The initial interesting portion of the results is shown in Figure 5.3, which plots the number of faulted pages on a logscale and the number of characters processed since they were last evicted from the buffer pool. As shown in these graphs, the number of evictions that have pages with immediate reference – i.e., before the complete addition of the next character into the suffix-tree – is *orders of magnitude larger* than the evictions of pages that are accessed after many more characters are added.

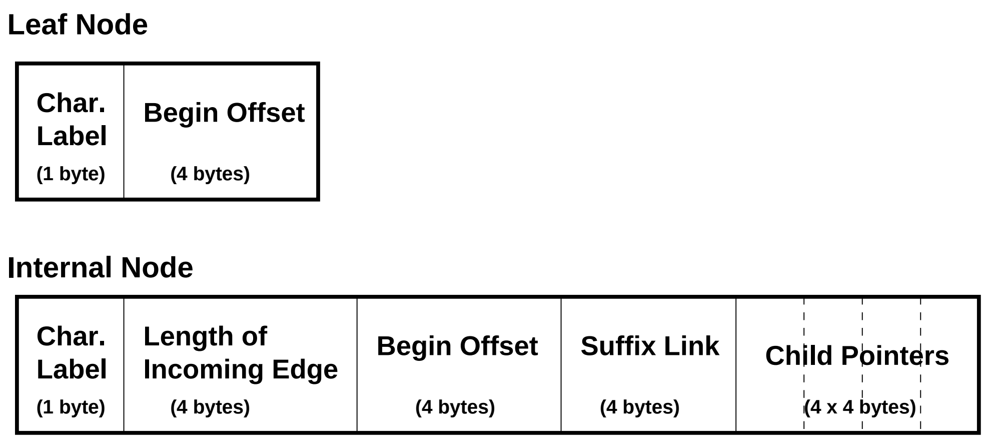
The *TOP-Q* strategy compensates for this un-responsiveness of TOP to such accesses, by splitting the buffer pool into a collection of pages maintained in the order of their path lengths – implemented as a *Heap* structure, and a short fixed-length queue of pages to hold the pages evicted from the heap. The buffered pages in the heap are chosen for eviction just like in the TOP policy. However, unlike TOP, these pages are moved to the short, fixed-length queue part of the buffer pool managed in a FIFO fashion. The presence of the queue of pages effectively introduces a *delay* in the eviction of pages, satisfying almost all the immediate references to the page. We have used a queue of 10 pages with buffer-pool sizes ranging from thousands to hundreds of thousands of pages in our experiments and found it to perform well in practice.

5.5 Suffix-Tree Representation

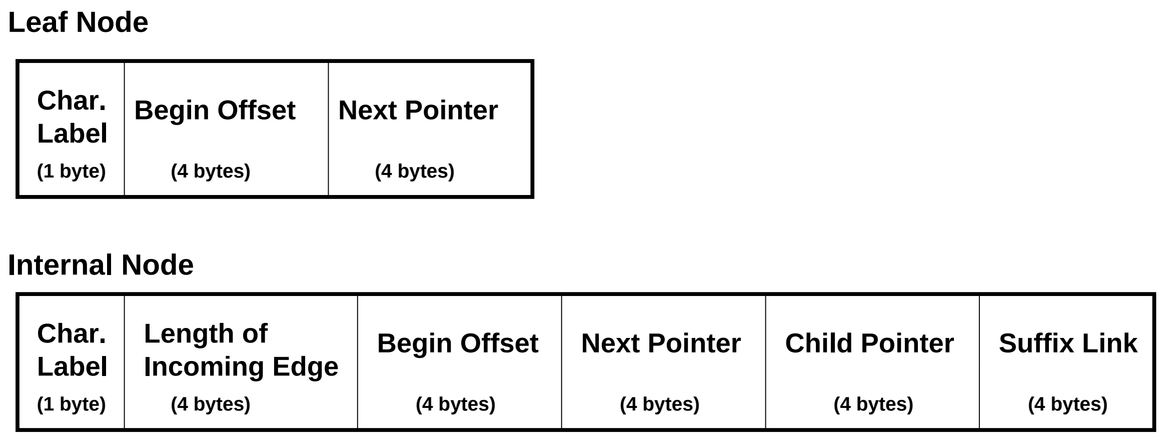
We now turn our focus to the physical representation of the nodes of the suffix-tree. Much attention has been paid to reducing the size of these nodes [72], the goal being to maintain the tree entirely in memory, so that non-local accesses over the tree induced by linear-time construction algorithms are not affected by the virtual memory paging [53]. However, it is not known which of these node representations is more appropriate when the suffix-tree has to be constructed and maintained completely on disk.

The simplest but most space consuming strategy is to use an array of size $|\Sigma|$ (where Σ is the size of the alphabet) at each internal node of the tree. Each array entry corresponds to an edge, whose edge-label begins with the character associated with the array entry. These edges are implemented as pointers to corresponding child nodes in the suffix-tree. We term this representation as *array implementation* of suffix-tree nodes. This representation, although simple to program, is not preferred in most practical implementations since this results in a lot of wasted space, with many pointers containing *null* values. This overhead is especially severe in nodes lower in the tree since the tree edges become sparser at lower portions of the tree.

As suggested by McCreight [80], a space efficient alternative to the array is to use a linked list of siblings, and at every internal node maintain a single pointer to the head of the linked list containing its child nodes. Traversals from the internal node are implemented by sequentially searching this list for the appropriate child node. By storing the linked list in a sorted order, it is possible to halve the traversal overhead. We refer to



(a) **Array Representation**



(b) **Linked-list Representation**

Figure 5.4: Structure of Suffix-tree Nodes

the resulting representation as the *linked-list representation* of the suffix-tree node. This structure is a popular choice, due to its simplicity in implementation as well as superior space economy.

The structure of internal nodes and leaf nodes, for the array and linked-list implementation is given in Figure 5.4. It is clear that the linked-list representation achieves its superior space economy by significantly reducing the size of every internal node – in our implementation, from 29 bytes per internal node in array representation to 21 bytes per internal node.

Name	**Description**	**Length** (in Mbp)	**%-age Distribution of nucleotides**			
			A	**T**	**C**	**G**
S	Symmetric Bernoulli	25	25	25	25	25
D	Drosophila Melanogaster genome	25	29	29	21	21
H	Human Chromosome II	25	30	30	20	20
C	C.elegans Chromosome I	15	32	32	18	18

Table 5.1: Characteristics of the Datasets

5.6 Evaluation Framework

In this section, we describe the framework used for evaluation of various buffering strategies, and node implementation choices, during the online construction of persistent suffix-trees.

In our evaluation, we use a total of four DNA datasets, three of which are drawn from C.elegans Chromosome I (dataset C), Human Chromosome II (dataset H) and complete genome of Drosophila Melanogaster (dataset D). The remaining dataset is a synthetic symmetric Bernoulli sequence over DNA alphabet (dataset S). The details of these datasets are summarized in Table 5.1. From these statistics we see that these datasets comprise of sequences ranging from symmetric distribution of alphabets (dataset S) to highly skewed distribution (dataset C).

5.6.1 Implementation Details

As already mentioned in Section 5.4, suffix-tree nodes are packed into fixed size pages before they are committed to the disk. The pages on disk are either internal pages or leaf pages, depending on whether they store internal nodes or leaf nodes of the tree. The variation in the internal and leaf node sizes leads to the packing density (i.e., the number of nodes in a disk page) of leaf pages being greater than that of internal pages.

The storage of both internal nodes and leaf nodes is in their order of creation. Each page is committed immediately to the disk, as soon as all the space in the page is utilized. Note that, at the time of committing to the disk, all the entries in nodes of the page may not be filled – some of them may be defined and updated at later times in the construction process. Also, a page is pinned in memory if there are any active references pointing to nodes in the page.

The internal and leaf pages are distinguished also in terms of their buffer pools. This is due to the distinctly different access patterns made during the construction of the tree. Internal nodes of the tree are used repeatedly (with or without suffix-links) for reaching the location of the next suffix. On the other hand, leaf nodes are re-visited only when the suffix being located ends in a leaf node. In fact, our buffer management system is designed such that separate policies can be applied to internal and leaf page buffer pools.

5.6.2 Buffer Management Policies

The design of buffer management policies has been an active area of research for many years, and a host of policies that show improved hitrates over various database workloads have been proposed [23, 33, 93, 106].

We compare the static policy of TOP-Q against the following popular policies that are based on page access statistics, commonly used in database management systems:

LRU (*Least Recently Used*) In case of a page fault, it replaces the least recently used page from the buffer pool to accommodate the new page. It incurs a constant time computational overhead for every access, in order to manipulate the list of pageframes maintained in the order of recency of access.

2Q The 2Q algorithm [64] is a constant time overhead approximation of LRU-2 [93] that is found to perform as well as LRU-2 for a variety of reference patterns. The 2Q algorithm covers the most important drawback of LRU-2, by reducing the computational overhead from $log(N)$ for *every access* in LRU-2 to a constant time overhead.

The metric of comparison between these popular buffering policies and our TOP-Q strategy are the *overall* buffer hit-rates observed during the construction of the suffix-tree, with increasing length of sequence indexed, and for a fixed amount of buffer space.

5.6.3 Buffer Pool Allocation

In our evaluation of buffering policies, we maintain separate buffer pools for leaf and internal pages, with the buffering policy applied within each of the buffer pools. With a fixed amount of memory space at our disposal as the buffer space, it is interesting to see if there is an effective way to *partition* this space between the two classes of pages of the suffix-tree.

The simplest partitioning is to distribute the available buffer pages equally between both leaf and internal nodes. It results in more number of leaf nodes being buffered than the internal nodes due to the better packing density of the leaf pages. Therefore, the effectiveness of the buffer pool can be improved by partitioning it to hold equal number of internal nodes and leaf nodes. Although the number of internal nodes is 0.6 - 0.8 times the number of leaf nodes (for typical DNA sequences), the level of activity over internal nodes, in terms of their accesses and updates, is much higher than over leaf nodes. Hence, partitioning schemes that are skewed to hold more number of internal pages than leaf pages can be expected to perform better in practice.

Additionally, it should be noted that, as the construction of the suffix-tree progresses, the overall size of the tree increases, leading to traversals over the tree covering a larger number of pages. In fact, a point may arrive when the available fixed size buffer may not be sufficient to efficiently handle the requests over an extremely large suffix-tree. In order to compensate for this growing size of the data-structure and provide a normalized performance measure for all the policies, we consider the steady hitrates obtained, when a fraction of the suffix-tree size is provided for buffering. In other words, as the suffix-tree construction progresses, more pages are introduced into the buffer pool such that the ratio of buffer pool size to the total size of the suffix-tree (measured in number of pages) is held constant. The distribution of steady hitrates obtained under various values of buffer fraction – starting from 100% buffering to a small fraction of the tree – will reveal the performance of each of the buffering policies, independent of the size of the tree. Furthermore, the performance of a buffering policy at a value f on this distribution, will also be its best performance with a fixed-size buffer when the ratio of buffer-pool size

	Array Representation	**Linked-list Representation**
Internal Node	29 bytes	21 bytes
Leaf Node	5 bytes	9 bytes
Page Size	4096 bytes	

Table 5.2: Default Experimental Parameters

to the current tree size equals f. In practice, the instantaneous performance could only be smaller due to the possible "memory effects" – i.e., retention of pages which may not be useful in the future.

5.7 Experimental Results

In this section, we present the results of our empirical evaluation of the buffering policies and the node implementation choices outlined in previous sections. The parameters applicable for all our experiments are summarized in Table 5.2.

5.7.1 Construction with Fixed-size Buffer

The fixed buffer size experiments were conducted with total memory space allocated for the buffer pool restricted to just 32MB, a total of 8000 pages. This enabled us to work with smaller length sequences, and perform experiments for collecting buffer pool statistics using a simulated memory hierarchy. However, in practice, much larger datasets will be indexed and therefore proportionately larger buffer pool sizes should be used.

As described earlier in Section 5.6.3, the available buffer space can be partitioned between internal and leaf buffer pools, ranging from equal partitioning to skewed partitioning in favor of the internal buffer pool. We experimented with many buffer partitioning schemes, and found that the overall hitrate is dominated completely by the internal node accesses alone. Thus, the performance improves with increased skew in the partitioning, with more space allotted for buffering the internal pages. This observation holds for both the array as well as linked-list representations of the suffix-tree. In this thesis, we present results for equi-partitioning of the buffer pool with 4000 pages each for managing internal and leaf pages and a highly skewed partitioning with 7950 pages to internal buffer pool and remaining 50 pages as leaf buffers.

Array-based Suffix-Tree Construction

The buffer hitrate obtained for page accesses, of both internal and leaf pages, using array representation of nodes is shown in Figure 5.5. The graphs also show the hitrate for simple TOP, in order to provide a measure of gains obtained by the TOP-Q extension.

Figure 5.5 shows that TOP-Q provides consistently higher hitrates than LRU and 2Q. In addition, the following observations can be made about these results:

- The hitrates with skewed partitioning of buffer pool are higher than with equi-partitioning. This clearly shows that the overall performance is dominated by the effectiveness of the buffering over internal pages.
- TOP-Q, as expected, performs better than the plain TOP strategy and provides hitrates that degrade slower with increasing suffix-tree size, as compared to the other policies.
- LRU exhibits the lowest hitrate, and, in fact, was found to have performance no different to the policy of evicting *a randomly* selected page – i.e., Random Replacement strategy! [2]
- The performance of TOP is better than LRU in the initial stages of the construction, and with increased skew in the buffer allocation, TOP improves to match the performance of 2Q.
- The ideal sequence for TOP is the dataset S, which is a symmetric Bernoulli sequence. With this dataset, it provides improved hitrates over even the highly sophisticated 2Q algorithm.

[2]Random Replacement strategy is considered to provide a practical lower-bound on the hit-rates of any buffering strategy, as it does not use any knowledge of past reference behavior [33].

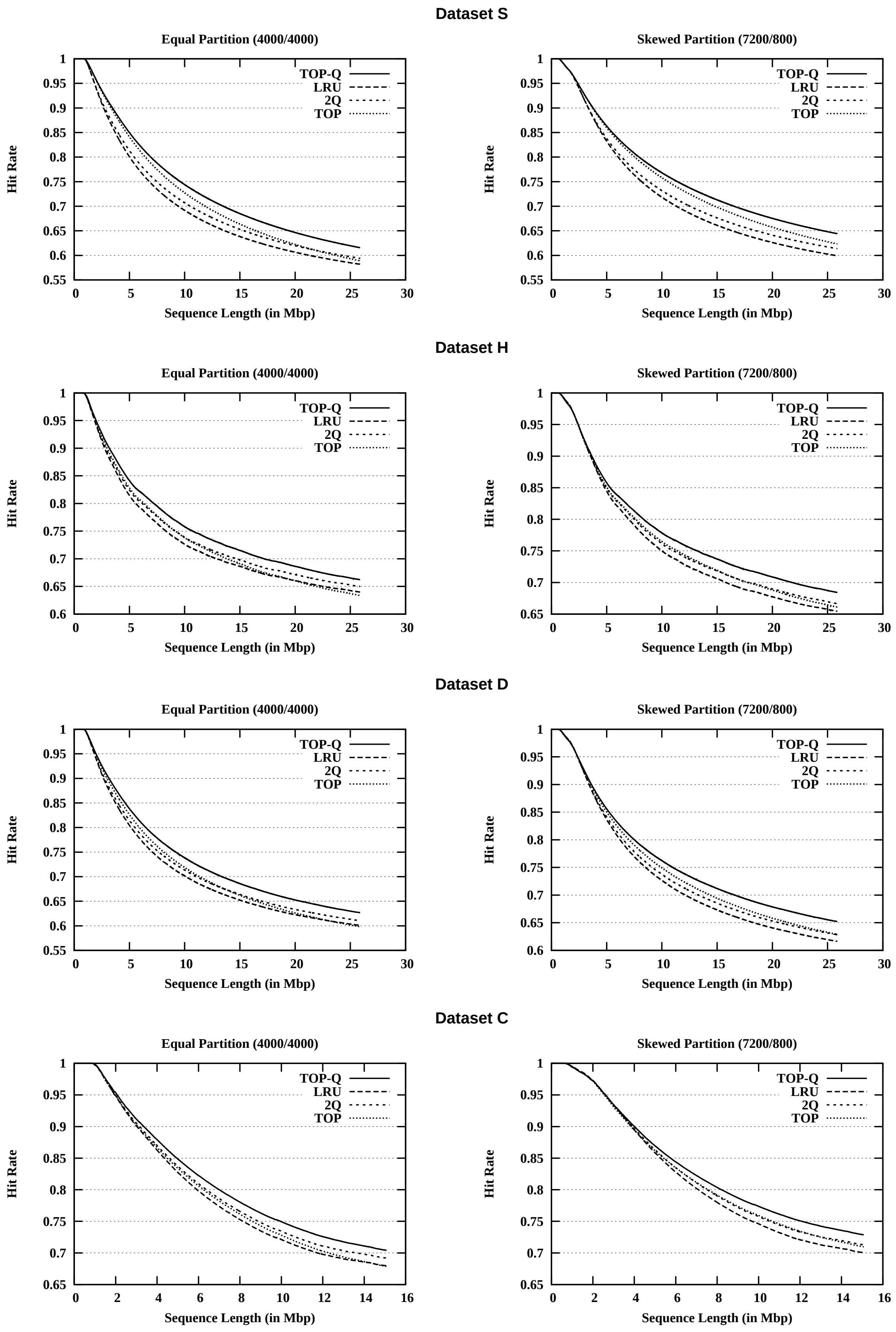


Figure 5.5: Hit-rates for Construction with Array-based Nodes

In summary, TOP-Q emerges as the best buffer management policy with any buffer partitioning strategy over all the datasets. Moreover, it should be noted that TOP-Q has an extremely low computational overhead as compared to LRU and 2Q, since its control data-structures are not updated at every access to a page.

Linked-list based Suffix-Tree Construction

The behavior of all the buffering policies for linked-list representation of nodes is shown in Figure 5.6. First of all, it is worth noting that all the algorithms provide better hitrates than in the case of array representation, with the sole exception of TOP. This is due to greater presence of correlated accesses, similar to those discussed in Section 5.4.1, arising out of traversals over the linked list of siblings. But the TOP-Q algorithm still maintains an edge over the LRU and 2Q algorithms, although the improvements are not as significant as in the case of the array representation. Another interesting point is that both LRU and 2Q show almost the same hitrates, with both equal and skewed partitioning of the buffer pool.

Choice of Implementation

The comparison of graphs in Figure 5.5 and Figure 5.6 indicates that the linked-list representation provides better hitrates than array representation. This is partly due to the fact that for the same amount of buffer space at our disposal, the linked-list representation buffers more number of internal nodes due to its improved space economy. This seems to suggest that the linked-list representation is better suited for persistent construction with buffering.

However, this conclusion is misleading since the sequences of page references in both cases are very different and the hitrates are normalized within each reference sequence. The absolute number of disk accesses made during the construction provides a metric that is independent of the reference sequence. Figure 5.7 shows the absolute number of read and write disk accesses, using the TOP-Q buffering policy, for the dataset H. As these numbers indicate, the linked list representation has a significantly higher I/O overhead

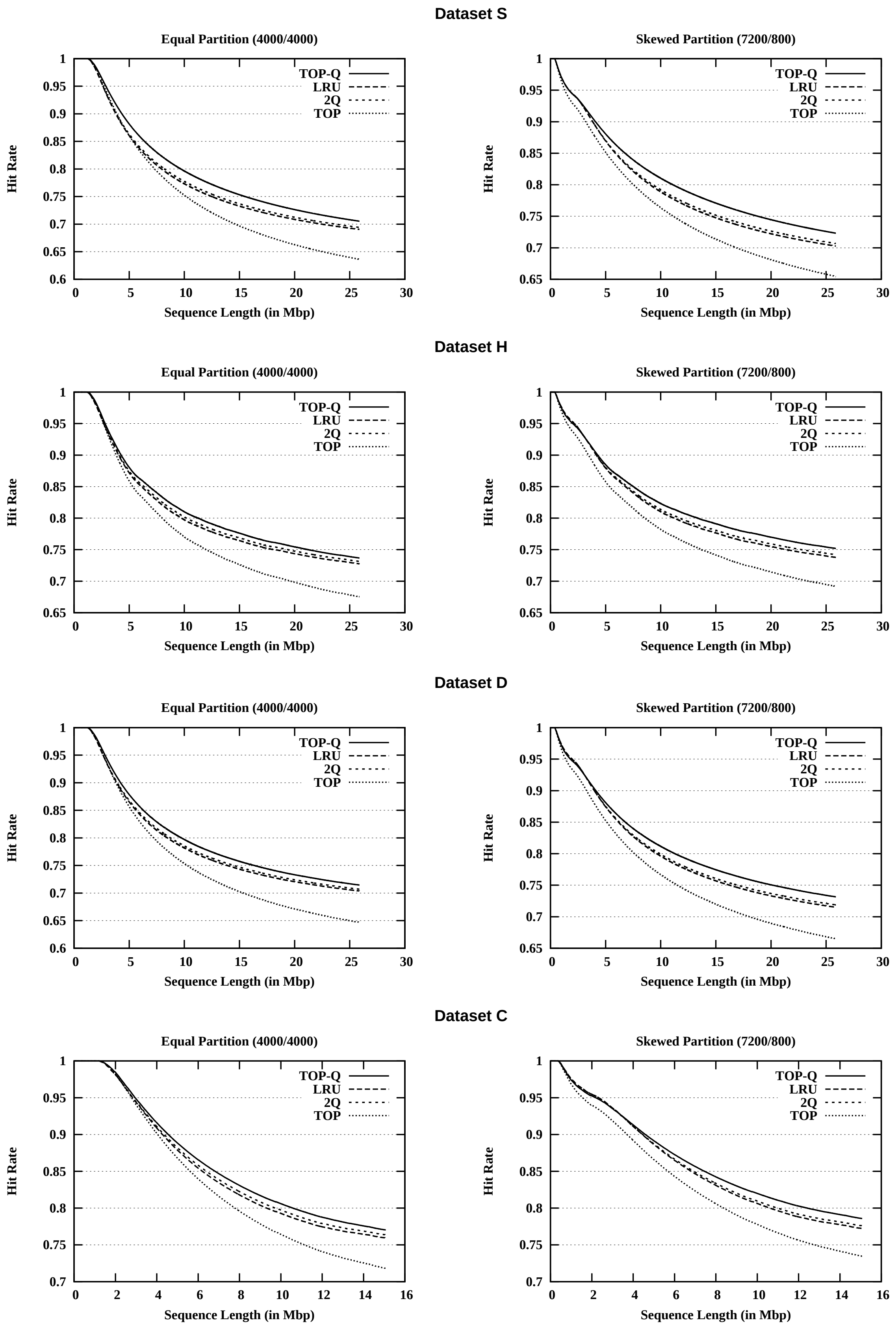


Figure 5.6: Hit-rates for Construction with Linked-list Representation

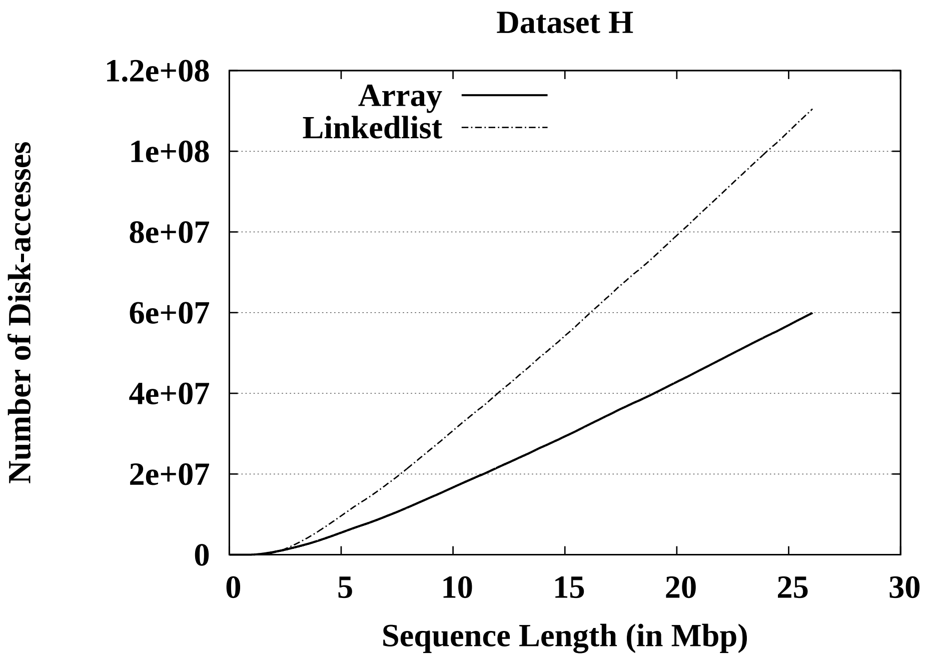


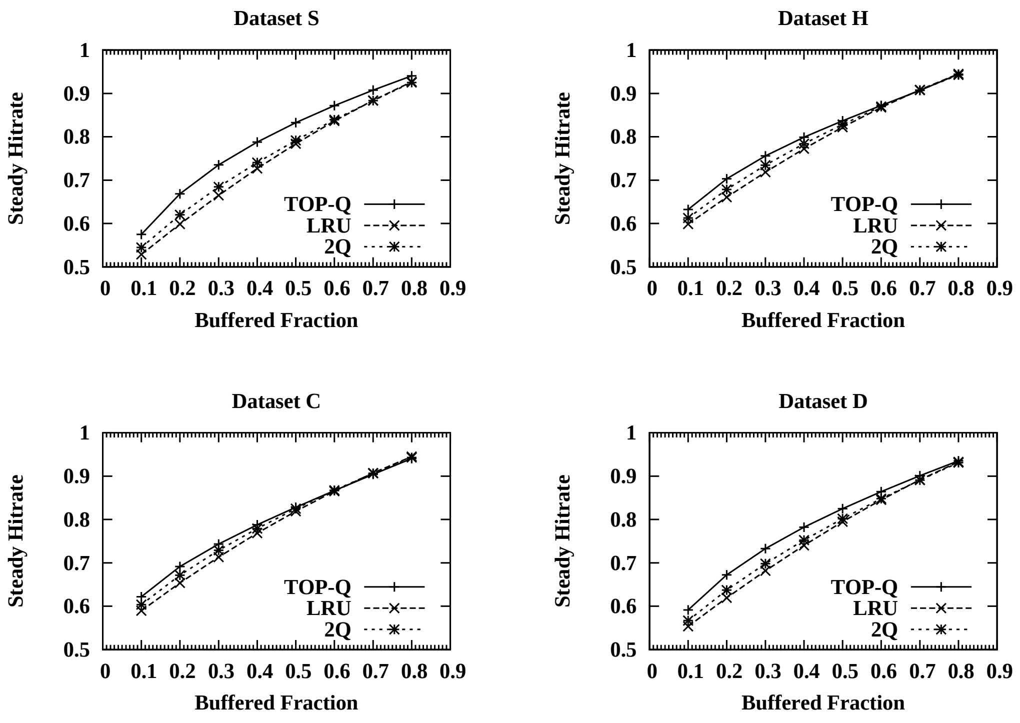
Figure 5.7: Disk-accesses during Construction

than the array representation. This overhead is primarily due to traversals over siblings in the linked-list to locate the appropriate child to follow. Each of these siblings could have been created at different points during the construction, resulting in their non-contiguous storage on disk.

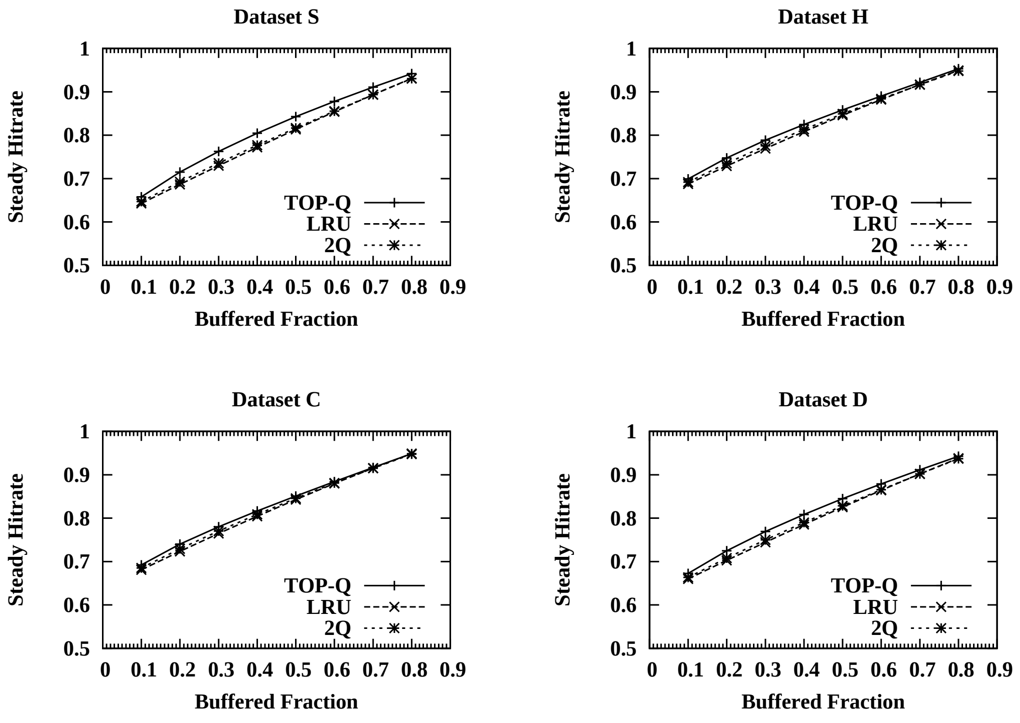
5.7.2 Construction with Proportional Buffering

We now move on to proportional buffering, discussed in Section 5.6.3. The resulting steady hitrate values are plotted in Figure 5.8.

As shown in these graphs, the performance of LRU improves almost linearly with the buffered fraction of the datastructure and 2Q is only marginally better than LRU. On the other hand, TOP-Q provides super-linear improvements with diminishing returns, with increasing fraction of buffering. The performance of TOP-Q peaks for about 25% of the tree in the buffer providing close to 72.5% hitrate for construction over real-life DNA sequences. When more than 60% of the suffix-tree is buffered, then all the three buffering strategies perform equally well with upto 85% steady hitrate.



(a) Array Representation



(b) Linked-list Representation

Figure 5.8: Behavior with Proportional Buffers

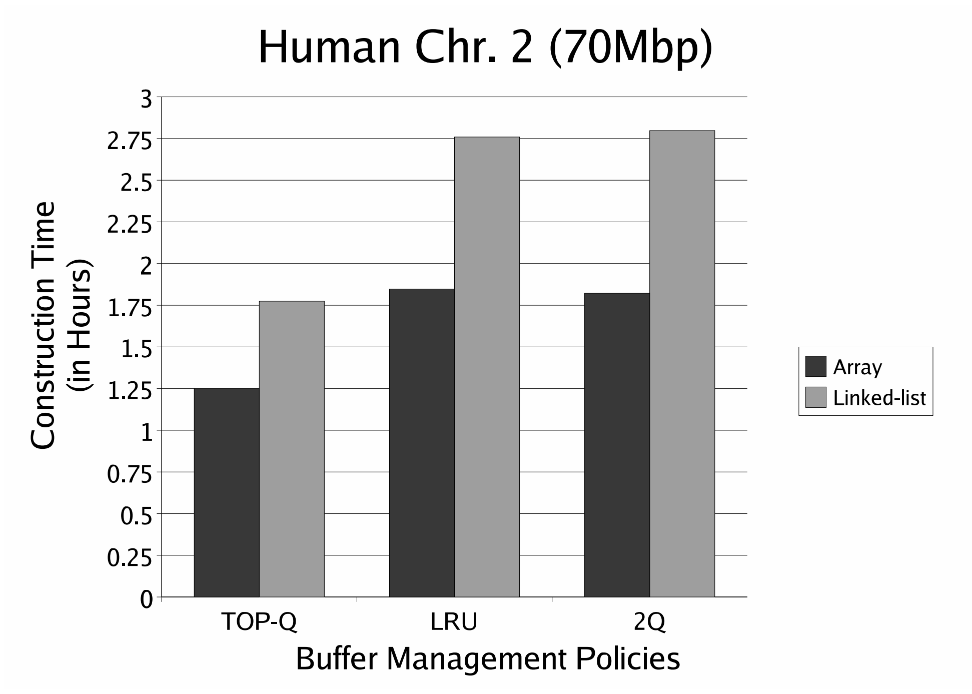
5.7.3 On-disk Construction

In order to consider the practical impact of the improved performance statistics for TOP-Q buffering and array representation of nodes, we built persistent suffix-trees for large DNA sequences on two different classes of machines. One was a PC class machine running Linux Redhat 8.0 and having an 18GB 10,000-RPM SCSI hard disk (IBM DDYS-T18350M model). The other machine represents the server class hardware used in current day large bioinformatics projects – a HP-Compaq ES45 server running Tru64 Unix 5.1 and 432GB (6*72GB) storage with single channel RAID controller at RAID-0 configuration. We refer to these two platforms as *PC* and *ES*, respectively, in the rest of this discussion.

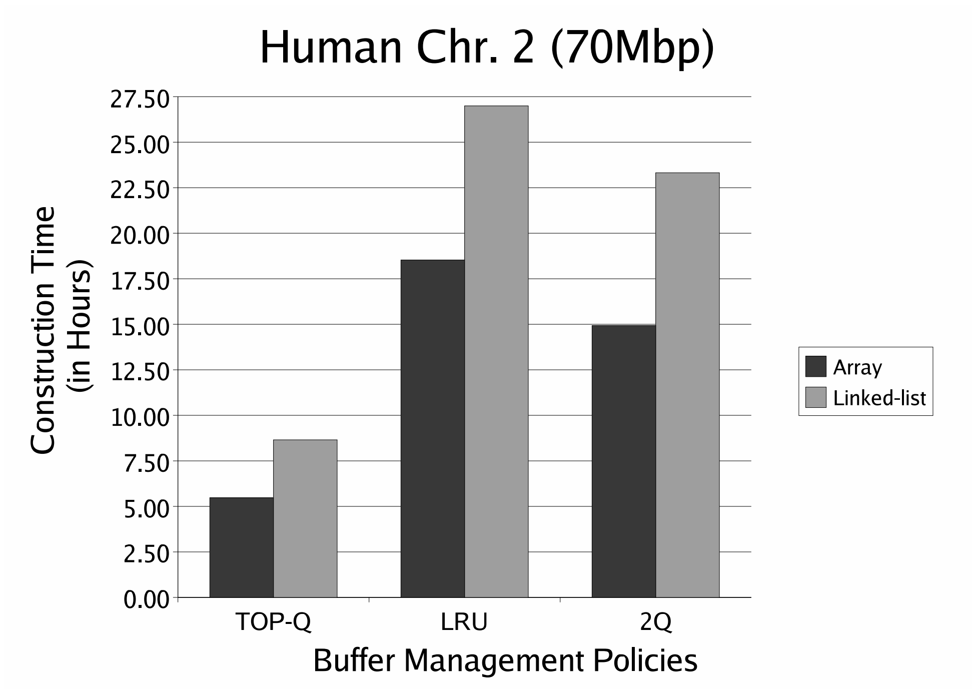
We compared the total execution time for constructing a persistent suffix-tree using 1GB of buffer space, split in 2 : 1 ratio, in favour of internal pages. Figure 5.9 provides the performance of various strategies on both the platforms. These results show that TOP-Q with array representation provides 50% to 80% improved construction time over both platforms considered. Further, we constructed a persistent suffix-tree over 250 Mbp sequence from the *Oryzasativa* (Rice) genome. Using TOP-Q strategy with array-based representation, the index could be constructed within *10 hours*, on the ES45 server.

5.8 Implementing TOP-Q Policy in BODHI

The underlying Shore storage manager (SM) for BODHI, has a global buffer-manager for each server-instance, that uses CLOCK replacement policy for managing a shared buffer pool. However, it has no direct support for changing the buffering policy based on the page-type and workload characteristics. Instead, SM provides an API for setting a “hate-hint” value for a record (and thus the page containing it), which is used by the buffer manager to determine how long to retain a pageframe in the buffer pool after it has been unpinned. By setting this value to 0, it is possible to schedule the page for eviction immediately, larger values retain the page for longer duration. In addition, SM also provides standard interfaces for pinning and unpinning of records in memory, ensuring that they are not evicted from the buffer-pool while they are pinned. We utilized these



(a) Platform: ES



(b) Platform: PC

Figure 5.9: Persistent Construction Times

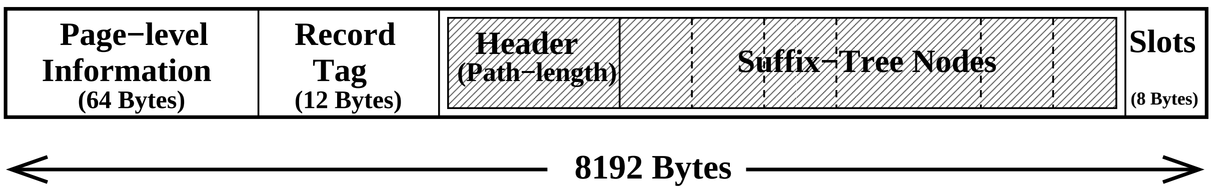


Figure 5.10: Storage Structure of Suffix-tree Index in BODHI

features to develop a VAS-layer implementation of TOP-Q buffering policy for persistent suffix-tree management in BODHI.

5.8.1 Storage Structures for Suffix-Tree Index

The default page-size used by the Shore SM is 8192 bytes, out of which 72 bytes are used for maintaining page-specific book-keeping information, including two 4-byte slot-entries, starting at the end of the page. A record consists of a header part, and the data part – header typically holds record details such as type information of the object, etc. For each record, Shore also adds a 12-byte tag with record specific information such as its serial number, record type, header length, and record length. Additionally, for each record beyond the first two records in the page, Shore also maintains a 4-byte slot-entry in the data space of page.

If one were to naively implement each suffix-tree node (internal or leaf) node as a record, we would spend more than 2-times the data-size just on the SM specific overheads – leaf nodes with 5-byte size will now be 17-bytes long! In order to minimize this overhead, we designed a single record that spans the entire data portion of the page, holding a number of suffix-tree nodes. Figure 5.10 illustrates the implementation of these records. The path length of the record is maintained as a 4-byte header information.

5.8.2 TOP-Q with CLOCK

The sequence services component of BODHI which is responsible for genome sequence storage, also maintains the suffix-tree index over the sequence collection. The records that are to be buffered by the TOP-Q policy – those in the TOP heap structure and in the eviction queue – are kept pinned in the buffer. As soon as a page is evicted by the TOP-Q policy, it is unpinned, and the hate-hint is set to 0 in order to schedule for eviction from the SM buffer pool as well. Note that a page will be retained in the buffer-pool, although it is scheduled for eviction, until the clock hand (of CLOCK replacement policy) passes over the corresponding pageframe. This can be viewed to be similar to increasing the length of the eviction queue in the TOP-Q policy.

5.9 Conclusions

In this chapter, we have evaluated the impact of buffering and internal node implementation choices on the construction of a suffix-tree in secondary memory. We also proposed a novel low overhead buffer management policy called TOP-Q, which exploits the pattern of accesses over the suffix-tree during its construction. Through an extensive empirical study involving both DNA and Protein sequences, we showed that TOP-Q performs better than other popular buffer algorithms such as LRU and LRU-2. The TOP-Q algorithm saves more than 75% of disk I/O by buffering merely 25% of the tree.

In addition, it was shown that that the commonly used, space-economical linked-list representation of the suffix-tree is extremely expensive for construction on secondary memory. Instead, a simple implementation using arrays at each internal node is shown to be far better suited for persistent suffix-tree representation.

A performance evaluation of TOP-Q with array representation of nodes against a popularly reported linked-list representation with LRU buffering policy showed that significant speedups to the tune of 50% to 70% were obtained, over different platforms.

Chapter 6

Search Optimized Suffix-Tree Storage

6.1 Introduction

As we discussed in previous chapters, despite the utility of suffix-trees in accelerating a number of sequence processing tasks, their practical usage has been limited over small length sequences due to their space overheads. Further, this piquant situation is rendered even worse due to suffix-trees not being disk-friendly, as a consequence of the random traversals across tree nodes induced by the standard construction and search algorithms. In addition to the techniques proposed in Chapter 5, there has been significant recent research activity to address this problem and design high-performance persistent suffix-trees [61, 110, 123].

However, these efforts have mainly focused on the *construction aspect*, that is, on how to build the tree efficiently on disk.[1] In this chapter, we take the next step of considering the *search aspect* in detail and investigate the associated efficiency concerns. Specifically, our focus is on whether it is possible to optimize the *layout* of the suffix-tree with regard to the assignment of tree nodes to disk pages, such that search is optimized. While layout

[1]Although in [60, 110], authors have reported the search performance on the resulting persistent suffix-trees, they have not explored the issue in detail.

has been well-studied in the database literature for access structures such as kdb-trees, Quad-trees etc., we are not aware of any similar work on suffix-trees. Further, carrying out this study for suffix-trees poses *new* problems arising out of the following:

- The patterns of search traversals over suffix-trees are much more complex than those found in traditional index structures, since both tree-edges and suffix-links are involved.
- Presence of suffix-links turn the suffix-tree into a *cyclic* structure.
- Suffix-trees are not inherently balanced data structures, unlike typical secondary memory index structures.

Our experiments with a variety of real genomic sequences against representative query workloads demonstrate that the currently available layout choices are *extreme* – they either optimize "vertical" traversal through the tree-edges, or optimize "horizontal" traversal through the suffix-links. But, sequence search algorithms typically need to traverse *both edges and links* – for example, to find all maximal matching substrings between the database sequence and a query, tree-edges are used to walk down the tree matching the query sequence along the way, and the subsequent matches are found by following the suffix-links [22]. Many popular genomics software such as MUMmer [29], and BLAST [1] achieve speedups through the resulting high-speed maximal substring location technique.

Given the above motivation for designing a holistic algorithm that optimizes the layout for both kinds of traversals, we present in this chapter **Stellar** (Suffix-Tree Edge and Link Locality AmplifieR), an algorithm that attempts to achieve this goal. Stellar is a linear-time, top-down strategy that utilizes the structural relationship between the suffix-links and the tree-edges under associated subtrees, to achieve high locality of both suffix-links and tree-edges. We quantify its effectiveness with a detailed performance study.

In summary, the contributions of this chapter are as follows:

Firstly, we demonstrate that the standard layouts of suffix-trees optimize only either edge traversals or link traversals, resulting in slow searches of genomic sequences.

Next, we present Stellar, a suffix-tree layout that optimizes both kinds of traversals, thereby providing significantly improved search performance.

Finally, through detailed empirical evaluation, we show that sequence searching over a LST is superior, in terms of disk I/O, than the same task performed over an UST, using $\mathsf{MSS}_{\mathsf{UST}}$ algorithm. These results quantify the search utility of suffix-links, thereby highlighting the need to retain them in persistent suffix-trees despite the associated space overhead.

Organization

The remainder of this chapter is organized as follows: Section 6.2 presents the index layout strategies currently available, and their ineffectiveness in the context of persistent suffix-trees. The design of our new Stellar layout algorithm is given in Section 6.3. The experimental setup is described in Section 6.4 before highlighting the results of our experimental analysis in Section 6.5. In Section 6.6, we present results to quantify the utility of suffix-links in search tasks despite the additional space overheads they impose. Finally, we summarize our results in Section 6.7.

6.2 Persistent Suffix-Tree Layout

Suffix-trees, unlike popular persistent index structures such as B^+-Trees and R^*-Trees, are not inherently balanced data structures – their structure depends entirely on the combinatorial characteristics of the sequence being indexed. In the worst-case, the tree can degenerate into a linear chain of internal nodes. Considering the example suffix-tree shown in Figure 4.2, leaf-node 8 is an immediate child of the root, while leaf-node 1 is at depth 3.

In addition, the fan-out degree of suffix-tree nodes cannot be varied to suit the disk-page size, since the fan-out of each internal node of a suffix-tree is upper-bounded by the size of the alphabet of the indexed sequence. Hence, many nodes of a persistent suffix-tree will be stored on a page, with nodes interconnected within as well as across pages.

Therefore it becomes critical to choose the nodes that will be placed in the same disk-page in order to reduce the overall disk I/O cost of traversing the suffix-tree during search.

Earlier research on disk layout of persistent indexes [31] has shown that a heuristic-based linear-time algorithm, henceforth called SBFS, that does recursive localized breadth-first layout of the tree, outperforms other commonly considered tree layout methods such as Breadth-first and Depth-first strategies. Through empirical studies they also show that the I/O cost of SBFS-ordered tree is within a small factor of the cost of an optimal *quadratic* time layout algorithm.

The basic idea behind the SBFS packing strategy is to recursively perform many local breadth-first traversals, beginning from the root of the tree, packing nodes in the order of visiting them into disk pages. Once enough nodes have been visited to fill a page, or there are no more nodes to be visited, the nodes visited so far are assigned to a page. Each of the remaining nodes in the BFS queue then becomes the root of a separate SBFS traversal. The recursion terminates when all nodes have been visited.

6.2.1 Issues in Persistent Suffix-Tree Layout

The storage layout of persistent suffix-tree introduces novel issues due to the inherent structural complexity of suffix-trees and also the non-traditional search traversals over the resulting structure. Note that the general problem of persistent graph layout is shown to be NP-complete [48].

Structural Complexity: In addition to the issue of complex search traversal patterns, suffix-trees exhibit higher inherent structural complexity than typical tree index structures due to the presence of *cyclic substructures.* As pointed out in Section 4.2.1, the collection of tree-edges as well as the collection of suffix-links in a suffix-tree form two separate rooted tree structures. Also note that in the tree structure induced by the collection of suffix-links, the links between nodes are *reversed* from the natural "parent-to-leaf" direction. That is, there exists a directed path starting at any internal node to the root of the suffix-tree (also the root of the tree induced by suffix-link collection), via a chain of suffix-links. And, from the root node, any of the internal nodes are reachable through a chain of tree-edges, thus completing a cyclic path.

Complex Traversal Patterns: The search algorithms over suffix-trees exhibit complex traversal patterns, significantly different from those commonly found in traditional indexing structures. In typical index structures the queries are mostly lookup queries involving root-to-leaf traversals. On the other hand, searching over suffix-trees involves simultaneous use of tree-edges and suffix-links. Thus, the layout strategy should take into account the two "orthogonal" traversal paths during suffix-tree based search.

Due to these complexities, none of the previously proposed layout strategies that are designed to work with either tree structures or DAG (directed acyclic graph) structures are directly applicable in the context of suffix-trees. Nevertheless, to serve as a comparative yardstick, we investigate the efficacy of SBFS strategy outlined above for laying out a persistent suffix-tree on disk, by *ignoring* the suffix-links during the layout process.

6.2.2 Search Utilization of Links and Edges

We now quantify the *relative utilization* of suffix-links and tree-edges during searches, in order to evaluate whether the search tasks indeed require combined locality of both forms of inter-node connectivities in the suffix-tree.

Figure 6.1 shows, for different query collections (described in Section 6.4), relative utilization of tree-edges over that of suffix-links during maximal substring search as λ, the minimum match-length threshold, is varied in the biologically significant operational region. Note that we also include the match-location reporting phase, which uses only tree-edges to traverse the subtree under the match.

These graphs demonstrate that although searches involve more traversals of tree-edges than suffix-links for lower values of minimum match length, the differential is within a small constant factor. Further, as the λ value increases, the utilization of tree-edges converges to within a factor of 2 of the suffix-links used, i.e., for every 2 edges, 1 suffix-

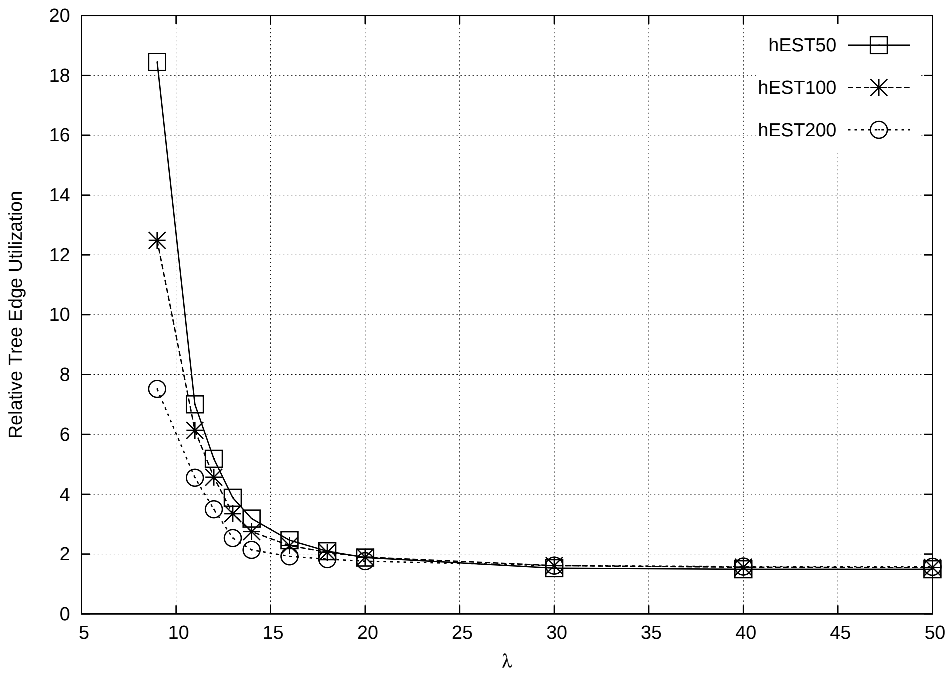


Figure 6.1: Relative Edge Utilization

link is traversed by the algorithm. Therefore, the number of suffix-links traversed is comparable to the number of tree-edges used during searches – suggesting that the search algorithms can significantly benefit by simultaneously improving the number of intra-page suffix-links as well as tree-edges.

6.2.3 Comparing the Quality of Layouts

Before we can evaluate different layout strategies, it is required to develop a metric that can effectively capture the structural variations between suffix-trees laid out with alternate layouts. One straightforward way to evaluate the quality of layouts obtained using different storage strategies is to execute a number of queries over the suffix-trees laid out using these strategies and measure the disk I/O cost. However, this evaluation depends heavily on the characteristics of the query workload. It does not immediately reveal to us the structural properties of the layout that could affect general search workloads.

The overall efficiency of a disk layout depends on the amount of inter-connectivity of nodes within a disk page. The nodes in the suffix-tree are interconnected through either tree-edges or suffix-links (or both). Hence, it is possible to capture the structural effectiveness of a layout strategy through the numbers of suffix-tree edges and suffix-links that are entirely *within a page*, i.e., the source-target pairs are placed in the same diskpage.

Using this metric, we first evaluated the layout obtained at the end of suffix-tree construction using OnlineSuffixTree algorithm, by ordering the nodes as they are created during the construction. We call this layout as CO (Creation Order) layout. We found that CO-layout provides practically *no tree-edge locality* – 0.2-0.5% of tree-edges were intra-page, while suffix-link locality was reasonably high – 39-42%.

Next, we used the SBFS strategy to layout the persistent suffix-tree – ignoring the suffix-links during the layout process. This resulted in the other extreme in locality characteristic, with 75-80% of tree-edges being intra-page, but virtually *no suffix-link locality* – less than 0.1% of suffix-links were local!

Table 6.1 summarizes the results of this evaluation, providing, in percent, the amount of intra-page tree-edges and suffix-links when the index is laid out using each strategy. These values were obtained with suffix-trees built on a 25 million base-pair (Mbp) length DNA sequence drawn from Human Chromosome 2, 15 Mbp length of C. elegans Chromosome 2, 25Mbp part of Drosophila Melanogaster genome, and, a 25Mbp symmetric Bernoulli sequence, with disk pagesize set to 4 KB. As these results indicate, CO and SBFS layouts represent (negative) extremes in persistent suffix-tree layout.

As a contrast, results for the suffix-trees ordered through our Stellar layout, described in detail in next section, are also presented in Table 6.1. The suffix-link locality of Stellar (40.0%) is close to that of CO, and tree-edge locality (62.6%) is comparable to that of SBFS – clearly optimizing both forms of connections simultaneously.

Before we move on to the description of Stellar, we explore the reasons for this extreme behavior of CO and SBFS layouts:

Dataset	**Storage**	**Suffix-Links**	**Tree-Edges**
Human Chromosome 2	CO	41.8%	0.2%
	SBFS	0.1%	77.5%
	Stellar	**40.0%**	**62.6%**
C. elegans Chromosome 2	CO	39.7%	0.3%
	SBFS	0.01%	76.4%
	Stellar	**39.6%**	**61.6%**
Drosopila Melanogaster genome	CO	33.1%	0.006%
	SBFS	0.0%	68.5%
	Stellar	**38.8%**	**59.2%**
Symmetric Bernoulli	CO	27.6%	0.0%
	SBFS	0.0%	69.8%
	Stellar	**38.8%**	**57.7%**

Table 6.1: Static Edge and Link Localities

CO Layout

During the suffix-tree construction, two successive internal nodes v_1 and v_2 are created typically as follows:

1. Traverse the suffix-link of the $parent(v_1)$ to reach $ancestor(v_2)$, and
2. Walk down the tree from $ancestor(v_2)$ using tree-edges, until a mismatch in the tree-edge results in the creation of v_2.

And, most importantly, the nodes v_1 and v_2 are related to each other through a suffix-link, since they correspond to consecutive suffixes of the sequence processed so far by the online construction. Due to this sequencing of tree node creation, a large fraction of suffix-links in the tree tend to be contained within a page.

We performed similar experiments with McCreight's construction algorithm [80], and found that the results are exactly identical.

SBFS Layout

The SBFS ordering, in contrast, is designed to cluster the tree nodes related through tree-edges into a diskpage. In a suffix-tree, the nodes related through a tree-edge share a common prefix – for e.g., in Figure 4.2, the leaf-node labeled 2 and its parent node share the common prefix `TA` (also shared by leaf-node 6). Thus, SBFS layout translates into a preferential clustering of suffix-tree nodes that correspond to substrings with common prefixes. However, the nodes with a common prefix have very low probability of also being related through a suffix-link – a situation that could occur only due to consecutive run of a symbol in the sequence. Even though the alphabet-size of DNA sequences is small, due to the pseudo-random distribution of symbols, long runs of a symbol are rare. Thus, the suffix-link locality of SBFS is extremely poor.

6.3 Design of Stellar

The design of Stellar is based upon the relationship between nodes connected through a suffix-link and the tree-edges under them, as shown through the following theorem:

Theorem 1 *If $v_2 = sl(v_1)$, then all the suffix-links originating from the nodes under v_1 point only to nodes under v_2.*

Proof: Let the path label of v_1 be defined as $\sigma(v_1) = x\alpha$, where x is a symbol from the given alphabet Σ, and α is a non-empty substring of the string being indexed. Then, $\sigma(v_2) = \alpha$, since $v_2 = sl(v_1)$.
Now, consider the subtree under node v_1, and note that the path labels of all nodes under v_1 have a common prefix defined by $\sigma(v_1)$. Therefore, we have, $\forall u \in descendent(v_1), \sigma(u) = x\alpha\beta$, where β is a non-empty substring of the indexed string. Similarly, $\forall u' \in descendent(v_2), \sigma(u') = \alpha\beta$. Further, if $u \in descendent(v_1)$ then, $\sigma(sl(u)) = \alpha\beta$. By definition of a suffix tree, there is only one path outgoing from root whose label is α, and that terminates at v_2. Hence the $sl(u)$ has to be under the subtree rooted at v_2. This completes the proof. $\square$

In other words, if two nodes are related through a suffix-link, then *all* the nodes under the source of this suffix-link have their suffix-link targets *only* in the subtree of the target. This property gives us a way to reconcile between the edge and suffix-link locality in the suffix-tree.

6.3.1 Stellar Algorithm

A pseudocode of Stellar algorithm, that utilizes the structural relationship in suffix-tree described above, is presented in Algorithm 3. The algorithm starts the suffix-tree traversal at the root of the suffix-tree, and recursively traverses the subtree below. When a node is visited, the suffix-link target of the node is visited next, if not already visited through the tree-edges. Thus an internal node and its suffix-link target are treated as a "buddy" pair, and are scheduled for recursive traversal in sequence. This results in subtree under a node and the subtree under corresponding suffix-link target to be recursively processed in succession – resulting in a large fraction of suffix-links that span these two subtrees to be intra-page, in addition to the tree-edges of each subtree. When enough nodes have been visited to fill a page, each node in the queue is scheduled for a separate recursive Stellar traversal, until all the nodes have been processed.

It is easy to observe that Stellar's complexity is linear in the size of the suffix-tree being processed – a node is visited only once during the top-down traversal of the tree. Additionally, it does not impose inordinate space overheads, as the only transient data structures required during the layout process are a queue of node ids, and a bit flag for each node of the tree indicating whether it has been visited or not. In our experiments we found that the queue never needs to hold ids of more than 100 nodes, even over DNA sequences exceeding 25Mbp.

In order to visually contrast the node clusterings produced by Stellar, SBFS and CO, consider the *intra-page connectivity* diagram of a suffix-tree laid out using each of these algorithms, presented in Figures 6.2, 6.3 and 6.4. These diagrams map the *intra-page tree-edges* as dark solid lines and *intra-page suffix-links* using dark dashed lines. The inter-page tree-edges and suffix-links are mapped in gray. The nodes of the suffix-tree are presented in their order of distance from the root. The suffix-tree presented here is built over a toy 100 basepair DNA sequence, with disk pagesize set to hold 5 nodes.

A visual inspection of these diagrams reveals that Stellar with 14 intra-page tree-edges and 22 intra-page suffix-links, generates tree layouts that exhibit better overall locality.

Stellar (r, B)
Input
r : Root of the subtree to be traversed
B : Capacity of the disk-page in terms of no. of nodes
Output
An ordering of the subtree under r

```
1: $queue \longleftarrow r$; {push root into the BFS queue}
2: $nodecount \leftarrow 0$; {initialize the counter}
3: while $queue$ not $\emptyset$ do
4:    $r' \longleftarrow queue$; {remove head of the $queue$}
5:    if $r'$ not visited then
6:       mark $r'$ as visited and increment $nodecount$;
7:    end if
8:    for all $c$ such that $c$ is a child of $r'$ do
9:       $s \leftarrow sl(c)$;{$s$ is the suffix-link of $c$}
10:      if $c$ not visited AND $nodecount < B$ then
11:         mark $c$ as visited and increment $nodecount$;
12:         $queue \longleftarrow c$;
13:      end if
14:      if $s$ not visited AND $nodecount < B$ then
15:         mark $s$ as visited and increment $nodecount$;
16:         $queue \longleftarrow s$;
17:      end if
18:   end for
19:   if $nodecount \geq B$ then
20:      while $queue$ not $\emptyset$ do
21:         $m \longleftarrow queue$;
22:         Stellar($m$,$B$);
23:      end while
24:   end if
25: end while
```

Algorithm 3: Stellar Algorithm

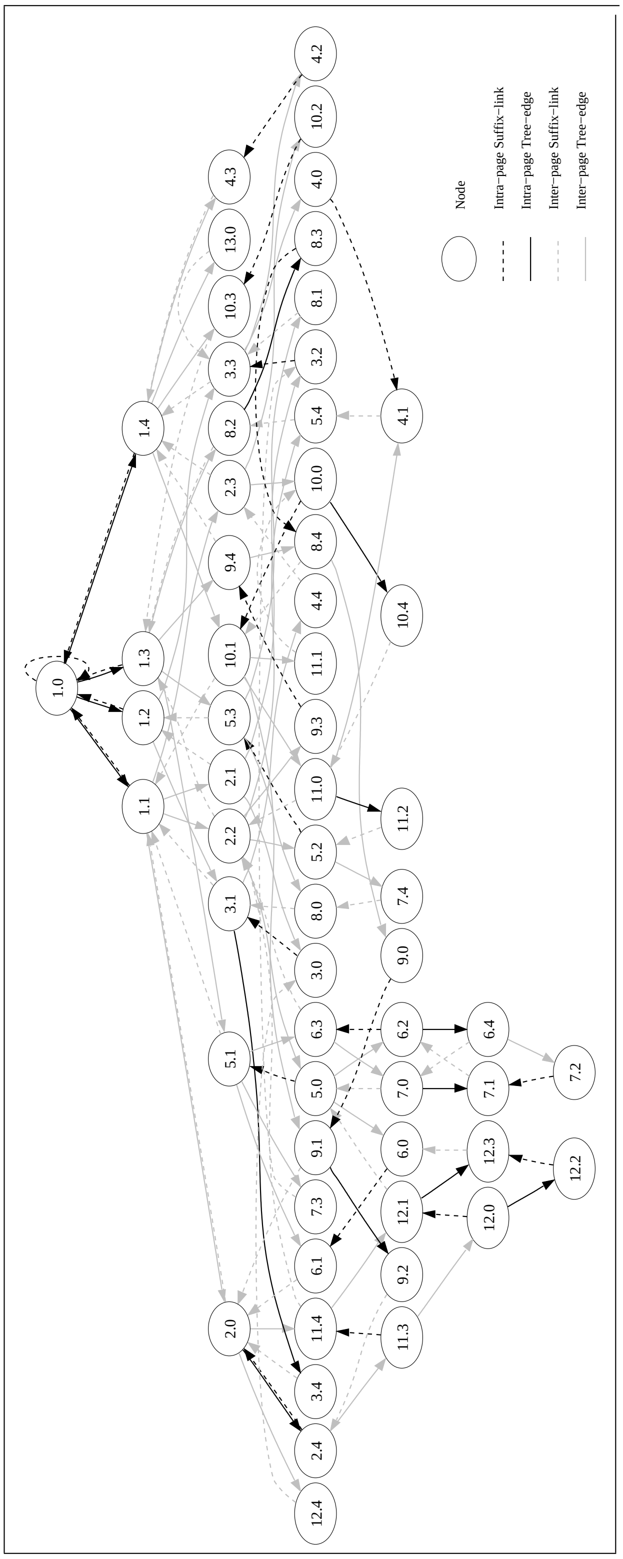


Figure 6.2: Intra-page Connectivity under Stellar

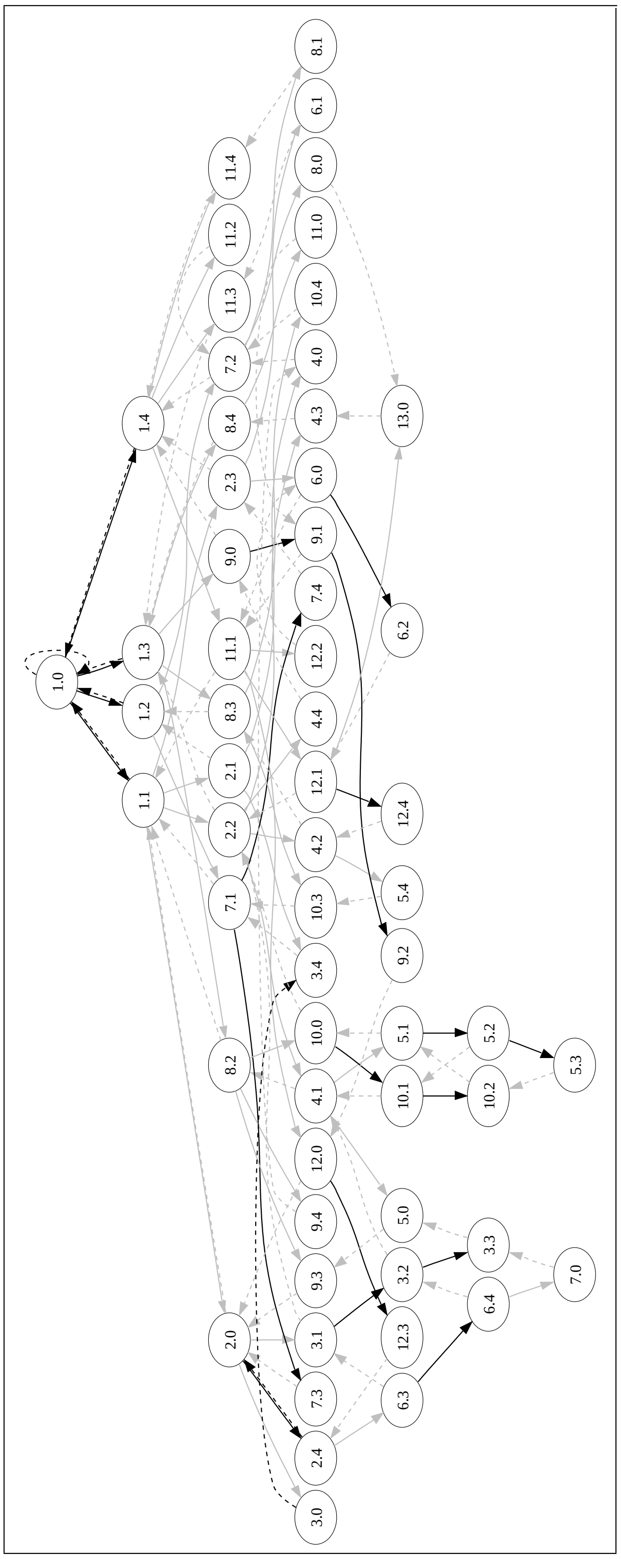


Figure 6.3: Intra-page Connectivity under SBFS

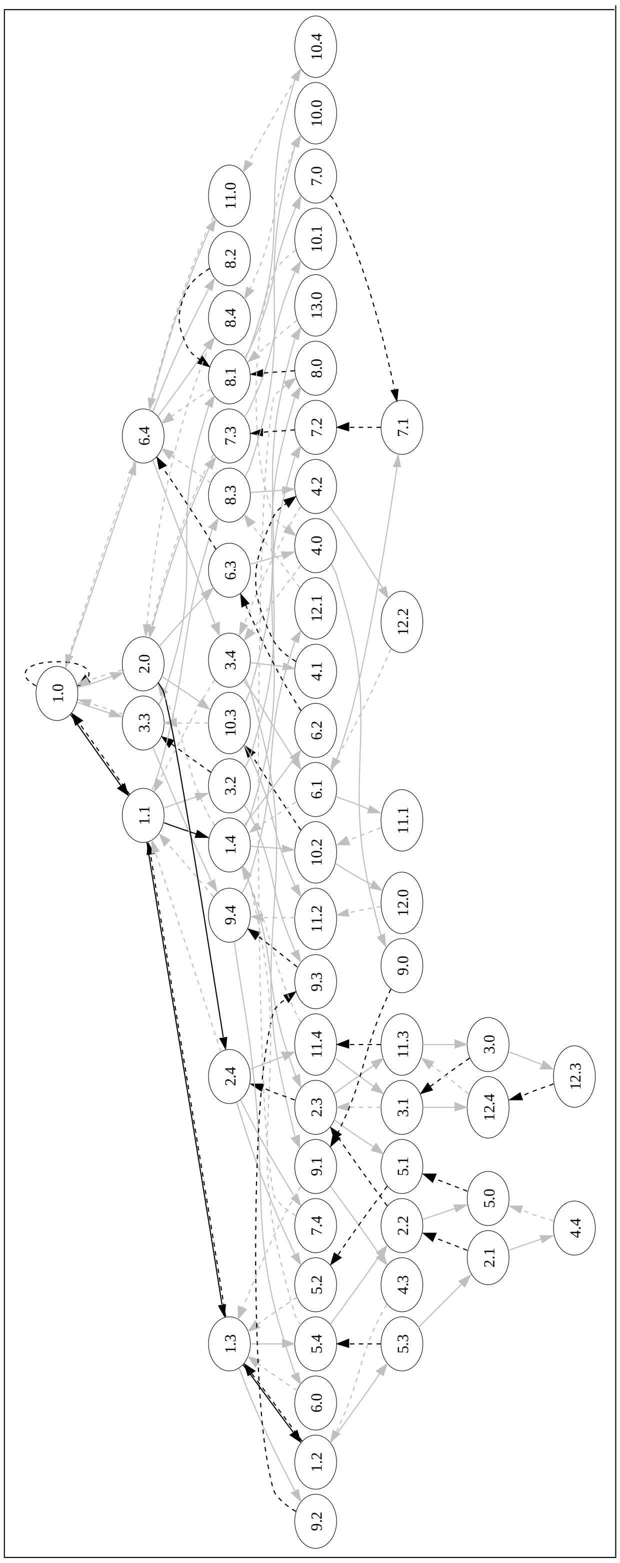


Figure 6.4: Intra-page Connectivity under CO

6.3.2 Level-wise Locality Variation

In addition to the overall locality of tree-edges and suffix-links obtained by the layout schemes, it is also critical to consider the distribution of such locality improvements in the suffix-tree. If most of the locality gains are restricted only to a small portion of the tree that may not be accessed frequently by the search process then the effectiveness of locality improvements is significantly reduced.

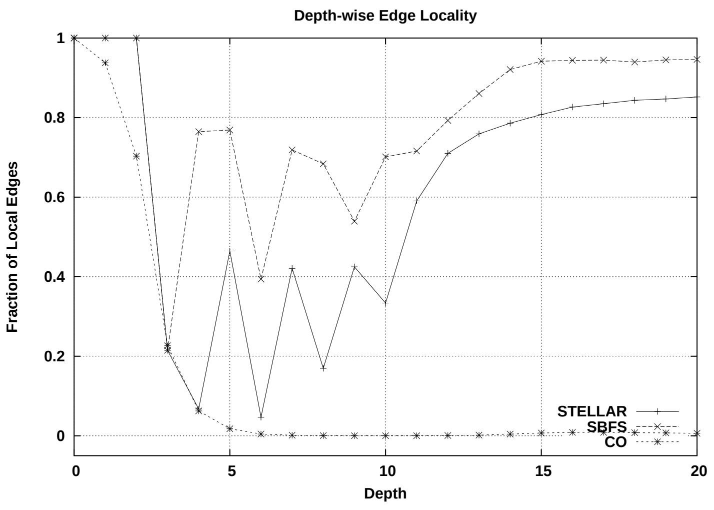
Figure 6.5 illustrates the distribution of locality of tree-edges and suffix-links for the suffix-tree over Human Chromosome 2 dataset (25 Million base-pair) under different layout schemes, including Stellar. These values represent the number of local tree-edges (suffix-links) at every level in the suffix-tree as a fraction of all the tree-edges (suffix-links) outgoing from that level. For example, there are a total of 2,417,879 outgoing edges from level 10, of which approximately 40% are intra-page due to the Stellar layout strategy.

As these graphs indicate, the tree-edge and suffix-link locality of all the three layouts are comparable at the top portion of the suffix tree. However, as the depth of the suffix-tree increases, the suffix-link locality of CO layout outperforms SBFS significantly, while at the same time SBFS shows significantly better tree-edge locality over CO. On the other hand, the locality due to Stellar algorithm is comparable to the best in both tree-edge and suffix-link locality metrics. In the middle portion of the suffix-tree, due to the large number of tree nodes, the locality fraction (of both suffix-links as well as tree-edge) is lower than in the top and bottom parts of the tree under all the layout strategies.

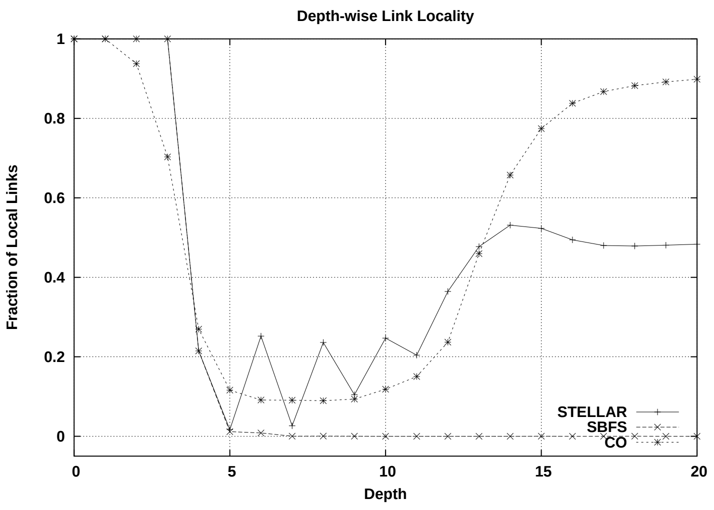
6.3.3 Impact of Pagesize Variation

It could be thought, at first glance, that increasing the page-size could significantly impact the locality property of layout algorithms. With increasing pagesize one can hold more number of tree nodes within a page, which in turn could potentially result in more number of tree-edges and suffix-links local to the page.

Therefore, we evaluated the tree-edge and suffix-link localities of all the three layouts, with varying size of diskpage. The results are shown in Figure 6.6. As these graphs show, the relative locality characteristic of all the three layout strategies does not vary



(a) Edge Locality



(b) Link Locality

Figure 6.5: Locality with varying Depth

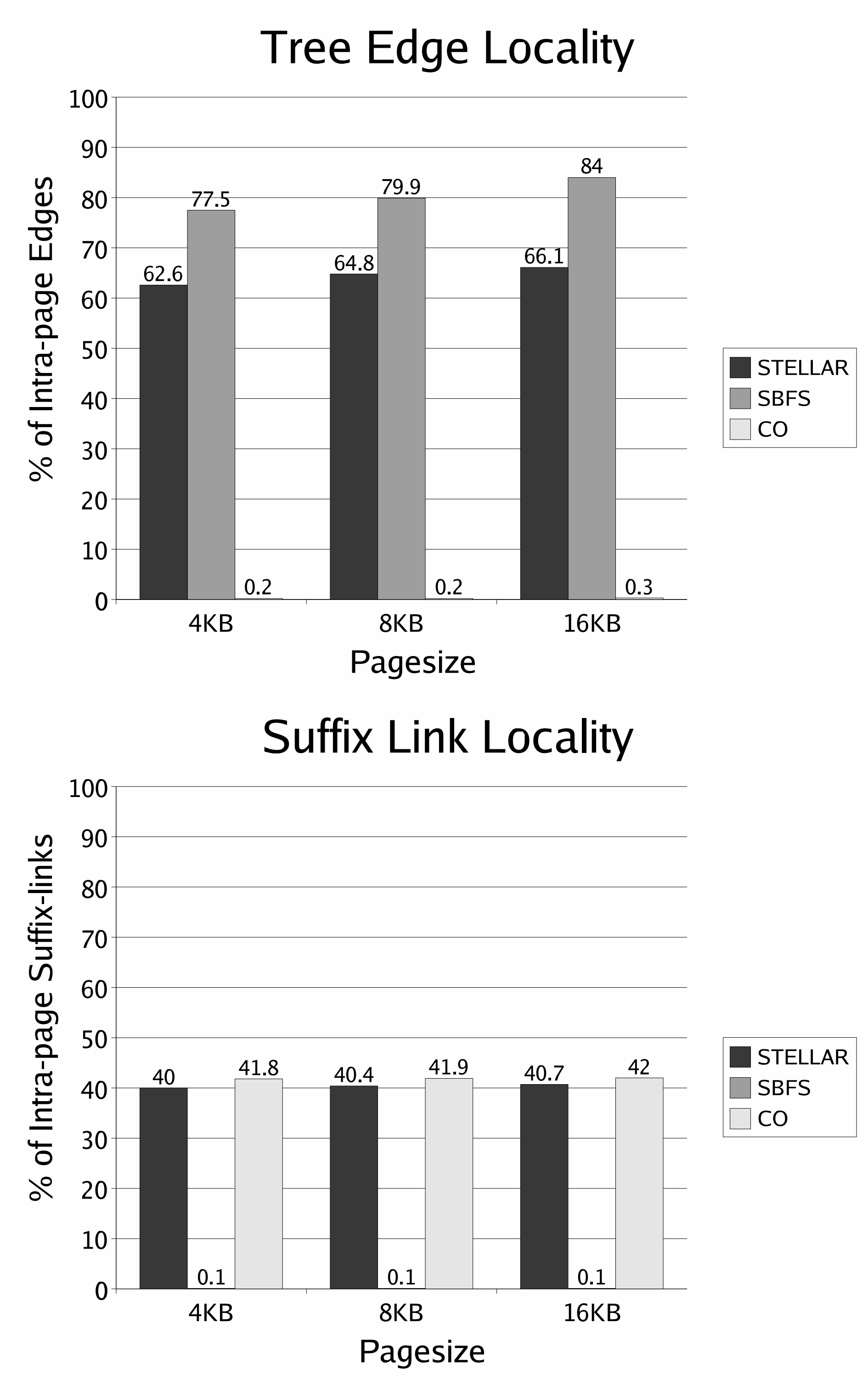


Figure 6.6: Locality with varying Pagesize

significantly, and Stellar continues to be close to the best for suffix-link or tree-edge locality metric.

6.4 Evaluation Framework

In this section, we present the experimental setup used for evaluation of different suffix-tree layouts during maximal substring searches between genomic sequences.

In our evaluation, we used a variety of real-world DNA sequences available from GenBank [9] repository as the datasets to build persistent suffix-trees. We present results for suffix-trees built over a 25Mbp sequence drawn from Human Chromosome II. The results over datasets show similar behavior, and have been omitted for purposes of clarity.

The suffix-tree implementation used in our experiments is based on the efficient array-based tree node representation, which as shown in Chapter 5, has significantly improved I/O characteristics than the alternative approaches. The indexing overhead in our implementation is about 22.5 bytes per symbol.

Suffix-tree nodes are densely packed into fixed size pages before they are committed to the disk. The pages on disk are either internal pages or leaf pages, depending on whether they store internal nodes or leaf nodes of the tree. During construction, the storage of both internal nodes and leaf nodes is in their order of creation. Each page is committed immediately to the disk, as soon as all the space in the page is utilized. Once the suffix-tree is completely constructed, the resulting suffix-tree is traversed in the required storage order and the reordered suffix-tree is built during this post-construction process. Unless mentioned otherwise, all experiments were conducted with disk pagesize set to 4K bytes, a typical pagesize in today's systems.

6.4.1 Query Collections

In order to evaluate the performance of the search algorithms and the tree layout strategies presented so far, we need to pay attention to the following characteristics of the query sequences that have considerable impact on the search process.

Query length: The length of the query impacts the overall time for locating all the matching subsequences, as it directly determines the total number of iterations (number of suffix-link traversals) during maximal substring location. In addition, increasing length of the query could also result in larger number of matches, increasing the overhead due to reporting of results.

Value of λ: As mentioned earlier, the maximal substring search algorithm takes as input a user-specified threshold λ, that serves as the lower-bound on the length of a match before all instances of the match are reported. If this value is too small, then there could be a large number of "noisy" matches that get reported throughout the database and if the value is too large then it will filter out potentially interesting similarities. Hence, this value is subject to variations in the domain as well as the task for which the suffix-tree indices are being utilized. The typical operational region of this parameter in genomic DNA sequence retrieval software is between 9 (for distantly related genomes) and 50 (in case of whole-genome alignments), which is used in our experiments to demonstrate the utility of the Stellar algorithm. A popular genome alignment software, BLAST, uses a default value of 11 to trade computational ease for some precision.

For DNA sequence searches, we used a collection of sequences from Expressed Sequence Tag (EST) database available from GenBank, as the base query collection. The Human-EST collection consists of 856,008 sequences with average length of each sequence being about 357.6 basepairs. The ESTs have been found to be extremely useful in high-throughput location of genes, genome mapping, etc., and form a key data collection in genomics research. Using this Human-EST collection, we generated length-restricted query collections of lengths 50, 100, and 200, by randomly sampling fixed-length sequences from each of these sequences. In order to remove any further bias in ordering of EST fragments generated, we randomly sampled 10,000 queries in each set to form three query collections, hEST50, hEST100 and hEST200.

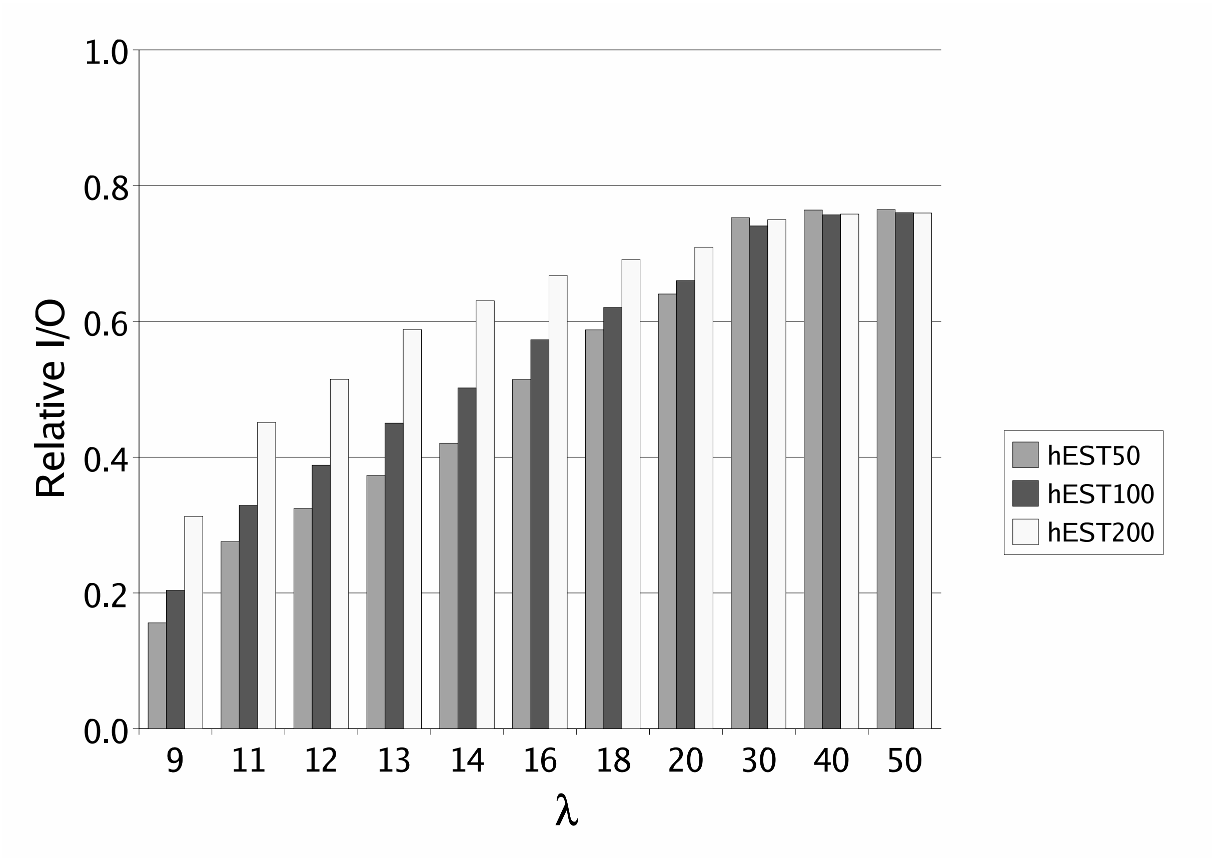


Figure 6.7: Stellar Vs. CO

6.5 Experimental Results

In this section, we present results of our empirical evaluation of various disk layout strategies for persistent suffix-trees during maximal substring search task. A buffer pool of 8MB, which forms approximately 1.5% of the total size of the suffix-tree, was used and managed using TOP-Q [8], a buffering policy specifically designed for use with suffix-trees.

6.5.1 Utility of Disk Layout

The relative performance of maximal substring search over persistent suffix-tree laid out using Stellar against the CO layout is shown in Figure 6.7.

As these results indicate, Stellar layout results in a small fraction of the disk I/O performed during search, when compared to the I/O incurred over suffix-tree in CO layout. For e.g., at λ set to 11, Stellar results in only 30-45% of the disk I/Os incurred by the CO layout. Although, with increasing value of λ, this performance differential reduces, Stellar never incurs more than 75% of disk I/O than CO layout.

When λ values are in the lower end of operational spectrum, e.g. set to 9, the overall I/O cost of search is dominated by the overhead due to reporting of all results. As a result of this, Stellar layout with larger fraction of local tree-edges clearly outperforms the CO layout which practically provides no tree-edge locality.

6.5.2 Performance of Stellar over SBFS

We now turn our attention towards comparison of disk I/O performance of suffix-tree layout schemes of Stellar and SBFS. In order to provide a normalized measure of performance for both the disk layout strategies, we measure their *relative performance gains* over the base disk I/O cost of searching over the suffix-tree in CO layout.

The relative performance of Stellar and SBFS with increasing values of λ is shown in Figure 6.8. As these graphs demonstrate, Stellar layout provides steadily increasing I/O gains with increasing values of λ. For example, at $\lambda = 11$, performance gain of Stellar over SBFS is close to 20%, which increases to more than 50% at $\lambda = 16$.

6.5.3 Cardinality Evaluations

In many uses of suffix-trees, it is enough to know the *cardinality* of matches rather than the identities of all the matches. For such uses of suffix-trees, it is interesting to see the behavior of disk-layout strategies when the I/O cost associated with the result reporting phase is neglected. The important point here is that the performance of maximal substring search algorithm *without* the result reporting phase is independent of the value of λ – this lower bound is used only to *decide* if the subtree below has to be traversed to report all the matches.

Figure 6.9 shows the α values for the three EST query collections we have considered. These results demonstrate that the performance of Stellar is significantly better than SBFS – with more than 2-folds improvement in I/O gains, when subtree traversals are not needed to report all the result identities.

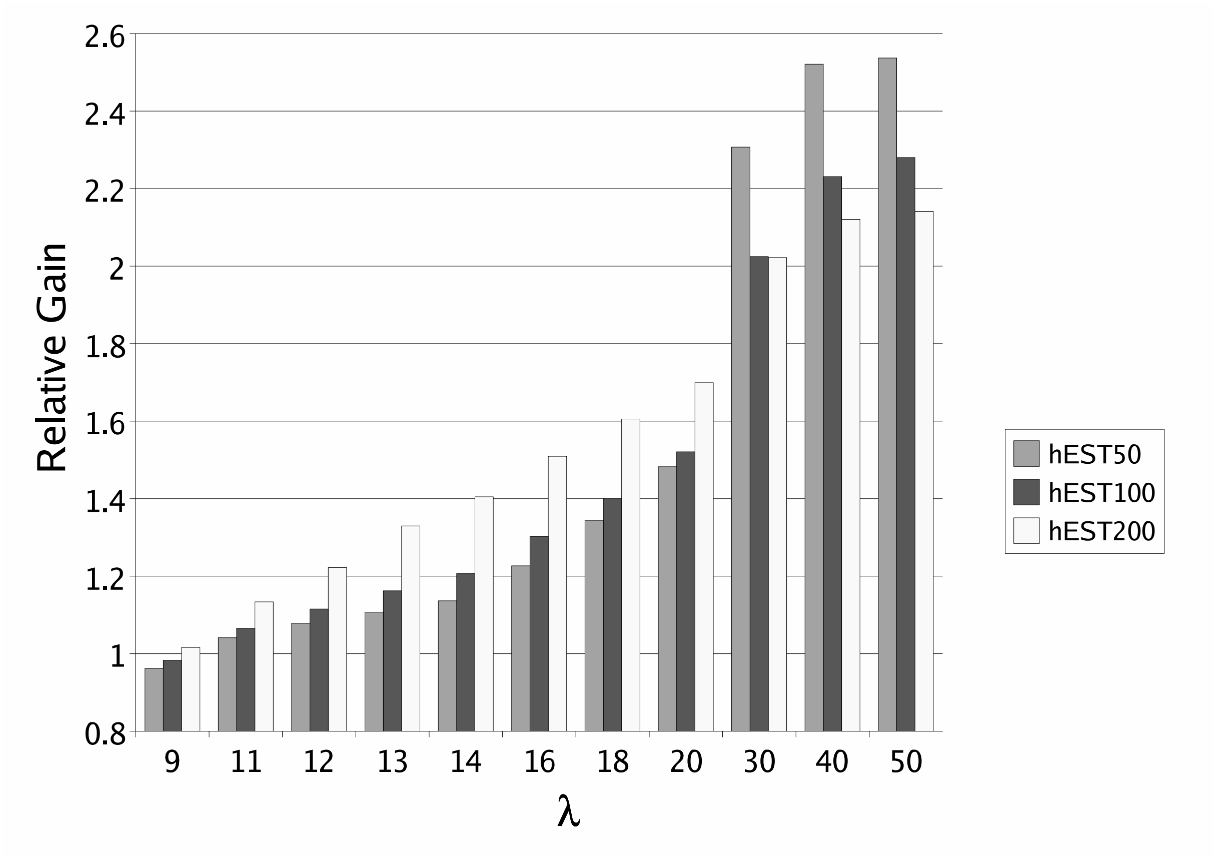


Figure 6.8: Stellar Vs. SBFS

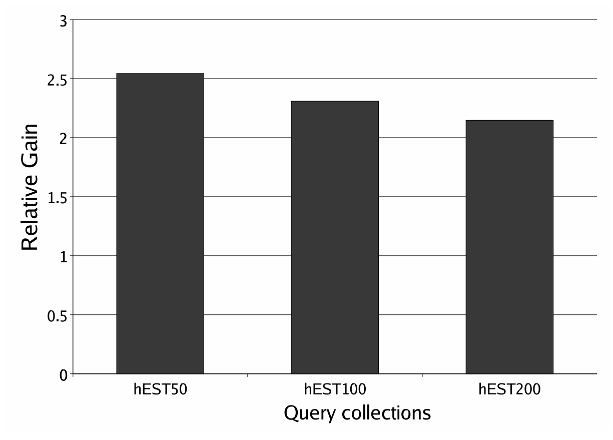


Figure 6.9: Stellar Cardinality Evaluation Performance

6.5.4 On-disk Search Performance

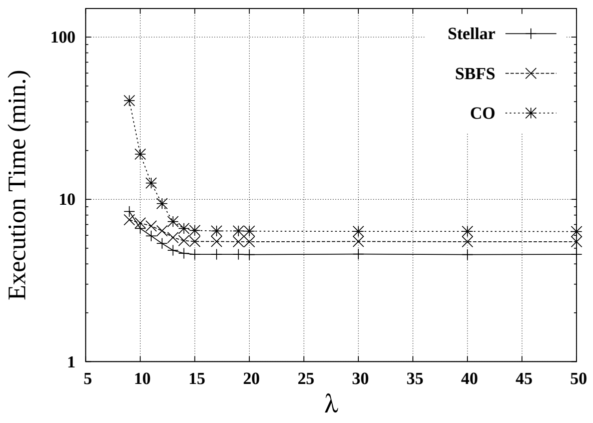
We now turn our attention to the impact of the suffix-tree layout on the time to execute a batch of queries on the persistent suffix-tree. Figure 6.10 plots the time taken, in minutes, for running a batch of 1000 maximal substring queries over a suffix-tree built on Human Chromosome 2 dataset. Note that the y-axis is plotted in logarithmic scale. These experiments were run on a HP-Compaq ES45 server running Tru64 Unix 5.1 with 6×72GB storage in RAID-0 configuration. We obtained the runtime profile, with 8MB buffer managed using the TOP-Q policy, after turning off the buffering induced by the operating system.

As these results indicate, persistent suffix-tree laid out through our Stellar organization can perform searches significantly faster than the other strategies.

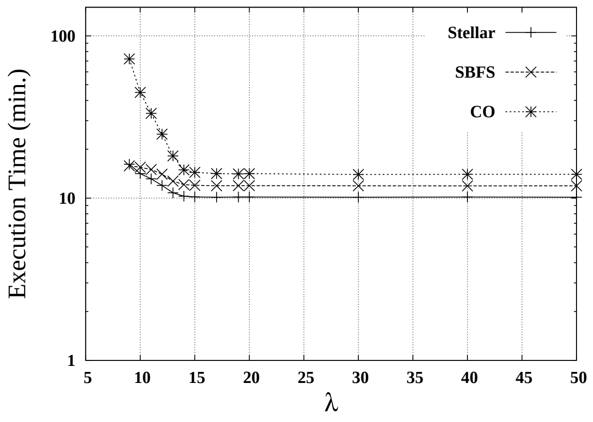
However, achieving this superior organization of LSTs comes with an associated, although necessary, computational work. We built an persistent LST using techniques presented in Chapter 5 over a 25Mbp fragment of Human Chromosome II with 32 MB of buffer. And this LST was reorganized into an Stellar ordered LST, with the same buffer size. Additionally, we utilized a transient translation table to map addresses of internal nodes in the original LST into corresponding nodes in the reorganized LST. This accounted for an additional additional $4 \times I$ bytes, where I is the number of internal nodes in the tree. In this experiment, the LST construction was completed in a little more than 4.5 hours, while Stellar reorganization needed additional 20 hours. Therefore, based on the characteristics of the application at hand, one should consider the layout strategies.

6.6 Search Performance over USTs

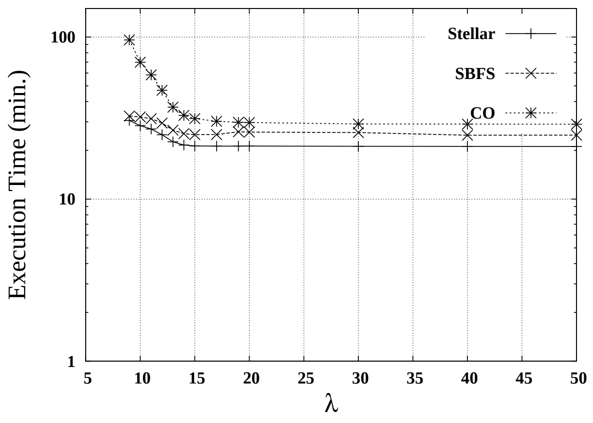
So far, we have studied the performance of maximal substring search over a persistent suffix-tree that provides suffix-links to traverse across the tree efficiently. However, it has been previously suggested that presence of the suffix-links in the tree results in poor performance of suffix-tree construction, and techniques for persistent UST (un-linked suffix-tree) construction have been proposed [61, 110, 123].



(a) hEST-50



(b) hEST-100



(c) hEST-200

Figure 6.10: Search Time Profile

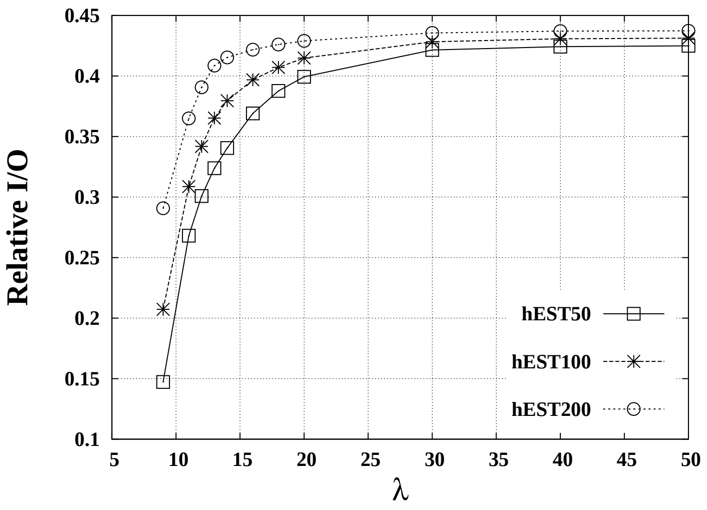


Figure 6.11: Gains due to SBFS Layout over UST

In this section, we first study the impact of disk layout on the search performance over USTs, and then compare their performance against persistent suffix-trees that have suffix-links, stored using Stellar disk layout. These experiments were conducted after incorporating additional index space optimization by removing the 4-byte suffix-link field in each internal node (bringing the indexing overhead to about 20.0 bytes per symbol). We used the persistent suffix-tree construction technique presented in [61], and the creation ordering of nodes generated during this construction comprises the baseline layout – similar to the CO-layout presented earlier.

6.6.1 Impact of Layout on UST Search Performance

In the absence of suffix links, we applied SBFS layout strategy, as it provides the highest tree-edge locality, and compared the performance gains obtained for $\mathsf{MSS}_{\mathsf{UST}}$ runs. Figure 6.11 plots the impact of disk layout, with increasing values of λ, over UST built over Human Chromosome II dataset.

These results show that a careful layout of persistent USTs saves more than 50% of disk accesses, and in the commonly used values of λ – ranging from 9 to 15, the I/O savings are in the range of 60-70%.

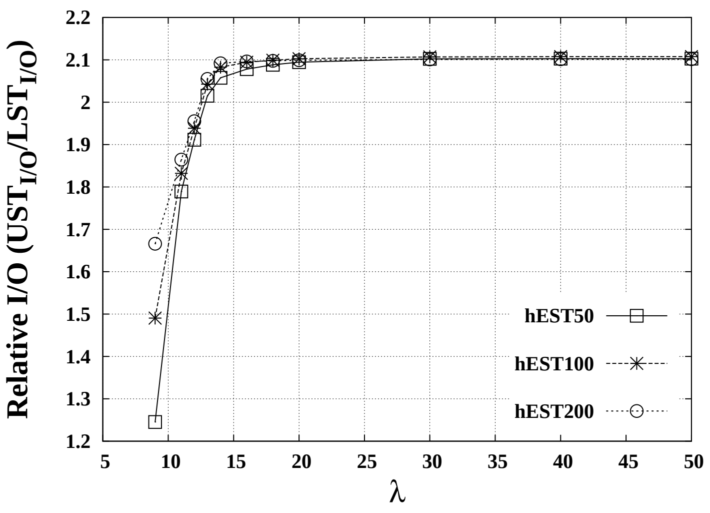


Figure 6.12: UST Vs. LST

6.6.2 Search Utility of Suffix-Links

We now turn our attention to relative I/O performance of search tasks over persistent UST and persistent suffix-trees with suffix-links (i.e., LSTs – linked suffix-trees), thereby quantifying the search utility of suffix-links. We compare the performance of UST laid out using SBFS strategy against the performance of LST laid out using Stellar layout strategy. Note that these layout strategies are optimized for the respective structural variants of suffix-trees.

Figure 6.12 presents the relative I/O incurred due to $\mathsf{MSS}_{\mathsf{UST}}$ as opposed to searching with LST, with increasing values of λ. As these graphs illustrate, searching over LST clearly provides distinct advantages over performing the same task with UST. Despite the superior space economy of USTs, it incurs more than 70% extra disk reads compared to LSTs. As the value of λ increases, the performance gap widens – at λ set to 20, UST incurs more than *2 times* the disk I/O than LST.

In order to confirm the wider applicability of these results, we performed similar experiments over UST and LST using the following two data-sequence and query-collection pairs:

Drosophila EST over Drosophila Genome fragment. In these experiments, the suffix-trees are built over a 25Mbp fragment of Drosophila genome (dataset D

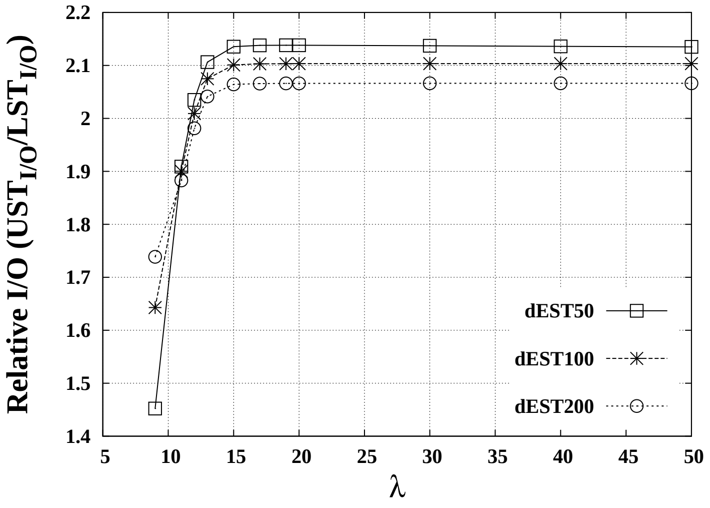


Figure 6.13: UST Vs. LST - Drosophila Melanogaster Genome

used in Chapter 5, and query collection was generated using the EST collection of Drosophila using the same method we used for generating hEST query collection. We distinguish this query collection as **dEST**. The results of the experiments are summarized in Figure 6.13. These graphs confirm our earlier finding that LSTs indeed provide significant I/O benefits over USTs which incur more than 2-times additional disk accesses during searching for practical values of λ.

Human EST over C. elegans Chromosome II. Both the previous experiments were conducted with both the query collection and the data-sequence drawn from the genome of the *same organism*. Now, we present the results of experiments when the query-set is drawn from an organism that is phylogenetically very distant from the organism whose genome fragment is indexed. We built suffix-trees over the Chromosome II genome fragment of C. elegans (dataset C of Chapter 5) and used hEST query collection over them. Figure 6.14 summarizes the results. These graphs show that even in this setting, I/O incurred by USTs is twice that of LSTs with Stellar layout for all practical values of λ.

These results show the need to retain suffix-links in the persistent suffix-trees, contrary to the persistent suffix-tree construction and maintenance recommended in [60, 61, 110, 123].

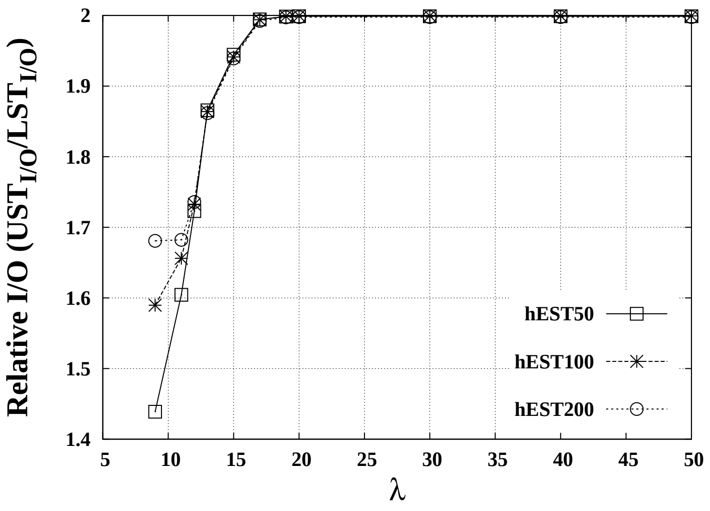


Figure 6.14: UST Vs. LST - over C.elegans Chromosome II

6.7 Conclusions

Developing suffix-trees as a persistent sequence index structure has been an active area in recent times, and many techniques have been proposed to significantly improve the construction time. However, there has been virtually no research on evaluating and optimizing the search performance of these persistent suffix-trees, the topic we have addressed here in detail.

Specifically, we have evaluated the impact of suffix-tree layout on disk on the I/O performance of common genomic search tasks, and shown through detailed empirical evidence that existing index layout algorithms are not effective for storing suffix-trees on disk. The layouts produced through these algorithms provide locality for only one of the two traversal paths used during suffix-tree searches, and practically zero locality for the other path.

Addressing this unsatisfactory state of affairs in persistent suffix tree layouts, we presented a layout strategy called Stellar, that optimizes the locality feature of both tree-edges and suffix-links in the suffix-tree. The layouts produced by Stellar show close to 40% suffix-link locality, and 60% tree-edge locality, thus combining the strengths of the two extreme layout schemes considered before.

Using real genomic DNA sequences drawn from GenBank repository, and querysets from Human-EST collection, we showed that Stellar incurs only about 30-40% of the disk I/O incurred by a suffix-tree stored in its creation order. Even in extreme cases, more than 25% disk costs are saved by laying out the persistent suffix-tree through Stellar. Furthermore, Stellar shows almost *2-fold improvement* over SBFS index layout strategy in terms of disk I/O saved. The relative performance of Stellar significantly improves with increasing values of λ (the minimum match length), thus highlighting the applicability of Stellar in full-genome alignment software such as MUMmer, where values of λ are typically in the range 20-50.

Finally, we presented results to show the utility of suffix-links in search tasks over persistent suffix-trees. Our experiments indicated that suffix-link based searching requires less than 50% of the disk I/O required for searching without suffix-links, *despite space overheads due to the presence of suffix-links* in every internal node. Contrary to the recent research in persistent suffix-tree construction where suffix-links have been abandoned to result in faster construction techniques, these results highlight that suffix-links be retained in the persistent suffix-trees in order to enable faster searches.

Chapter 7

Persistent Suffix-Trees for Proteins

7.1 Introduction

Previous chapters focused on the construction and storage organization of persistent suffix-trees over DNA sequence collections. Although it is one of the most important application domains for suffix-trees, there are many other areas where suffix-trees can be gainfully employed. There are proposals to use them to index a variety of sequence collections, such as protein sequences, textual data, time-series data [70] etc., as well as for 2-dimensional raster images [44]. In these applications, unlike DNA data, the underlying alphabet size could be rather large. For example, protein data comprises of an alphabet made of 20 amino-acid symbols.

In most of these applications, suffix-trees are used to accelerate similarity search algorithms similar to those used in DNA sequence processing. Therefore, the suffix-link based algorithms we considered in previous chapters are equally applicable in these domains as well. With steady growth of the underlying data collections in each of these domains, there is a clear motivation to study the persistent versions of suffix-trees in these applications.

Even though the algorithms for suffix-tree construction and searching continue to have linear time and space complexities for sequences with larger alphabet-size, their *absolute* performance is considerably affected due to the alphabet-size due to the following reasons:

1. In the case of array based implementations, effect of increased alphabet size appears

in the *space utilization* of the suffix-tree. Each internal node of the suffix-tree will now have a $|\Sigma|$-size array for outgoing child pointers. As the disk pagesize is fixed, this increase in the node size adversely affects the packing density. Furthermore, for a fixed length of the indexed sequence, the fraction of *null* pointers in this array increases as the size of the underlying alphabet is increased.

(An approach that suggests itself for reducing the fraction of null pointers due to large alphabets is to reduce the effective alphabet-size by splitting up the bits of the symbols in the original alphabet and then build the suffix-tree over the equivalent (but longer) sequence on the smaller alphabet. However, it is important to note that this approach will not yield an equivalent suffix-tree directly – a substring match in the lower alphabet space is not equivalent to a substring match in the original alphabet space due to missing information on the symbol boundaries. Additional steps are needed to ensure that the resulting substring match is indeed a true match in the original alphabet-space as well. There are atleast two alternative solutions based on this technique – *Sparse Suffix-trees* [68], and *Word Suffix-trees* [2]. Both these proposals result in a significantly complex construction algorithm, and additional steps in the substring searching. Hence, in this thesis we do not compare with these techniques.)

2. On the other hand, with linked-list representation, the $|\Sigma|$ factor shows up in the *time complexity* of the suffix-tree construction and search algorithms. Increasing the alphabet size results in increased length of the sibling linked-list that needs to be traversed for locating appropriate child node. This adversely affects the overall I/O efficiency of suffix-tree algorithms.

Further, the distribution of symbols in strings of larger alphabet could be very different from that of DNA, severely affecting the behavior of suffix-tree indexes. In addition, increase in the alphabet size typically alters the profile of access patterns observed during searching over the suffix-tree. The minimum match-length threshold, λ, is reduced as the alphabet size increases to account for the increased domain of matching substrings. As a result, TraverseSubtree needs to be performed much higher in the suffix-tree.

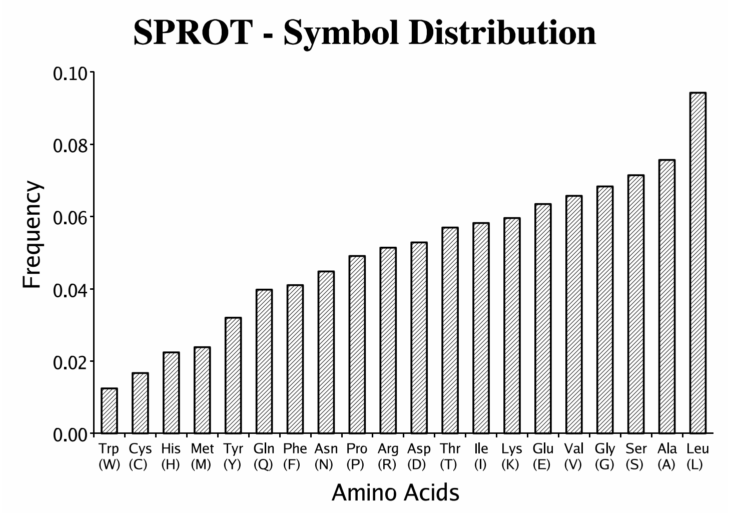


Figure 7.1: SPROT: Distribution of Symbols

In the context of BODHI, *protein sequences* are an important large alphabet sequence data that needs to be indexed for similarity searching. In this chapter, we present results of the performance study of the techniques presented in Chapter 5 and Chapter 6, for construction and searching over suffix-trees built over protein sequences. Specifically, we make the following contributions:

Firstly, we show that the TOP-Q buffering strategy outperforms the LRU and 2Q strategies, for suffix-tree building over protein sequences. These results demonstrate that TOP-Q is an effective strategy for suffix-trees independent of alphabet size, over a variety of datasets.

Next, we present performance numbers to further highlight the superiority of array representation of suffix-tree nodes over linked-list representation, even in the presence of large alphabets. Again, these results clearly show that, despite their reduced space-economy, array representation is the implementation choice for persistent suffix-trees.

Finally, moving on to the search aspect, we show that storage organization has considerable impact on the search performance of persistent suffix-trees on large alphabets.

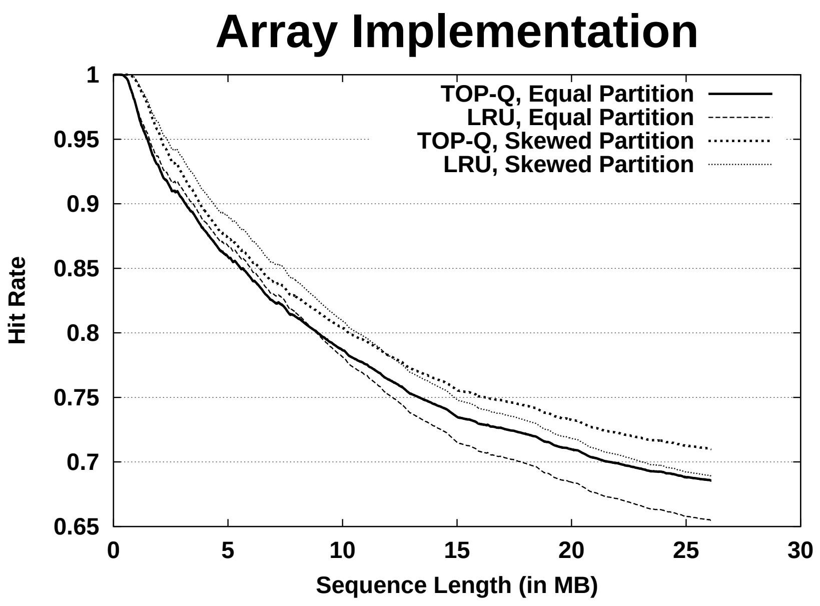
In our experiments, we used a 25 million length amino-acid sequence dataset, SPROT, derived from SwissPROT collection of protein sequences [120]. Although the individual protein sequences are short, it is possible to build and use a single suffix-tree index for the complete database by concatenating the individual sequences together. Such a suffix-tree is called a *Generalized Suffix Tree*, and commonly used for indexing a collection of short sequences [53]. Accordingly, we generated the SPROT dataset by concatenating the protein sequences. The distribution of symbols in the resulting data sequence is plotted in Figure 7.1. As these values show, the symbol-wise distribution in the SPROT dataset is highly skewed, in marked contrast to the pseudo-random nature of symbols in DNA sequences.

7.1.1 Organization

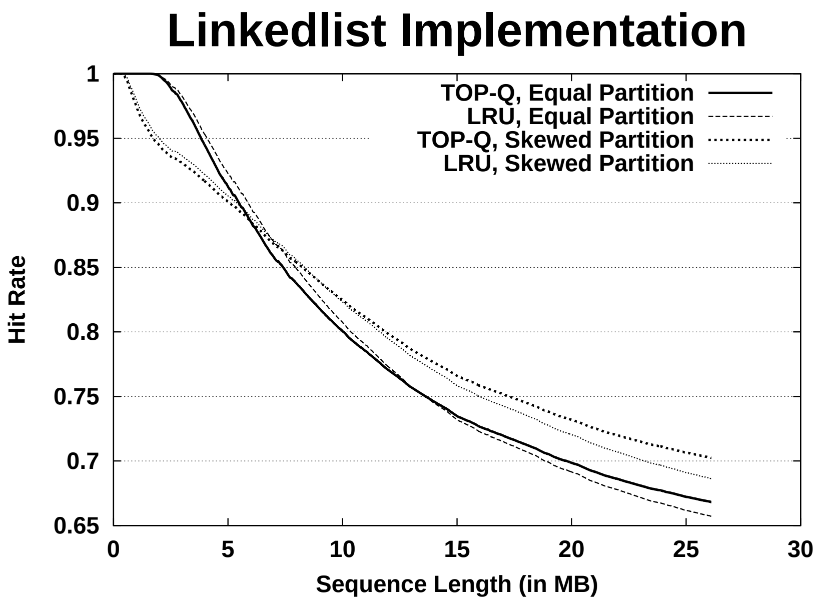
The remainder of the chapter is organized as follows: In Section 7.2, we present results for constructing persistent suffix-trees over protein sequences. Next, in Section 7.3, we evaluate the performance of various storage organization strategies and their impact on the search performance. Finally, we conclude in Section 7.4.

7.2 Suffix-Tree Construction over Protein Data

The results of our experiments with the SPROT dataset are summarized in Figure 7.2. As in Section 5.7.1, in these graphs we present results for two partitioning schemes of the available buffer-pool, namely, Equal-Partition (4000 buffers each for internal and leaf pages) and Skewed-Partiotioning (7950 buffers for internal pages and 50 buffers for the leaf pages). The graph does not include hitrates for 2Q, since they were very similar to LRU. As these graphs demonstrate, with increasing length of the indexed sequence, the hitrate of TOP-Q continues to gain over that of LRU. Further, TOP-Q responds favourably to the increased bias for internal nodes in the the buffer pool allocation. Thus, the benefits of TOP-Q are applicable not only with small-alphabet sequences such as DNA, but also with large-alphabet sequences.



(a) **Array Implementation**



(b) **Linkedlist Implementation**

Figure 7.2: SPROT: Hitrates during Construction

7.3 Storage Organizations over Protein Data

We now turn our attention towards the performance of suffix-tree *searching*, under different disk layout schemes. The locality results for the suffix tree over SPROT dataset, under different disk layout strategies is shown in Table 7.1. As these numbers demonstrate, Stellar shows a locality profile similar to that for DNA sequences. Similarly, the variation of the locality with the size of the diskpage is illustrated in Figure 7.3. The graphs in Figure 7.4 illustrate the depth-wise distribution of the locality profile throughout the suffix-tree. In contrast to the similar graphs for DNA sequences in Figure 6.5 earlier, the locality distribution displays a sharp variation at depths ranging from 0 (root of the suffix-tree) to 5.

Dataset	**Storage**	**Suffix-Links**	**Tree-Edges**
SPROT	CO	49.2%	0.2%
	SBFS	0.1%	56.1%
	Stellar	**31.6%**	**49.6%**

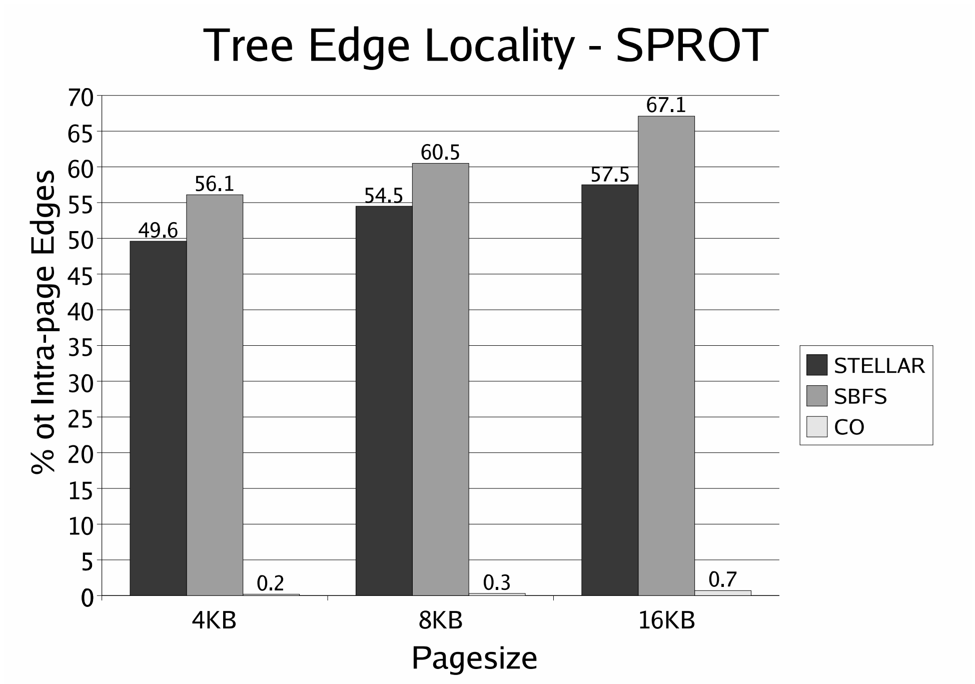
Table 7.1: SPROT: Static Edge and Link Locality

It should be noted that due to the choice of array-based representation of the suffix-tree, the size of the internal node is significantly larger (93 bytes as opposed to 29 bytes for DNA), leading to much smaller packing density of nodes. As a result, the scope for packing nodes related via either a tree-edge or a suffix-link into the same page is reduced, thus lowering the absolute value of locality.

7.3.1 Protein Substring Searches

For the protein sequence searches, an amino-acid query set of 10,000 randomly sampled sequences from translated Human UniGene non-redundant set of gene-oriented clusters was chosen [128].

Although the common search task over protein sequence databases is maximal substring location, the parameters used in these search tasks differ significantly from those



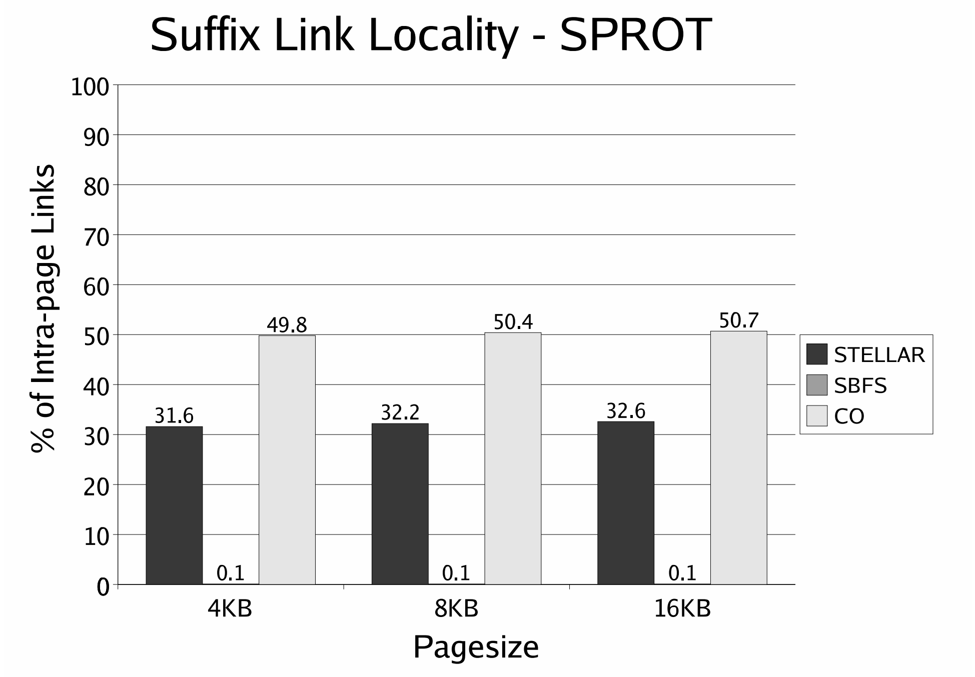
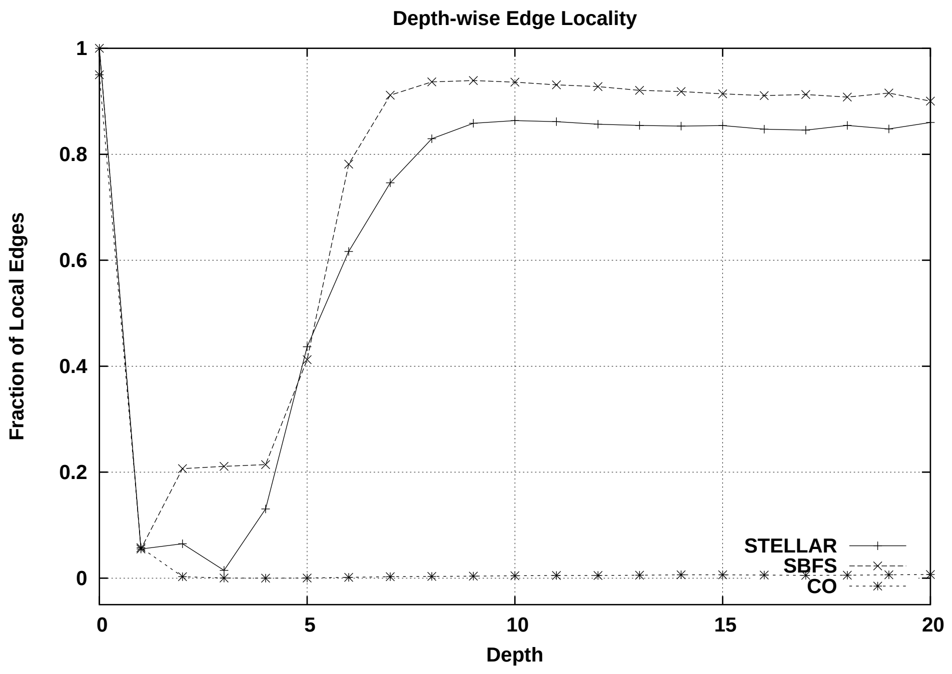
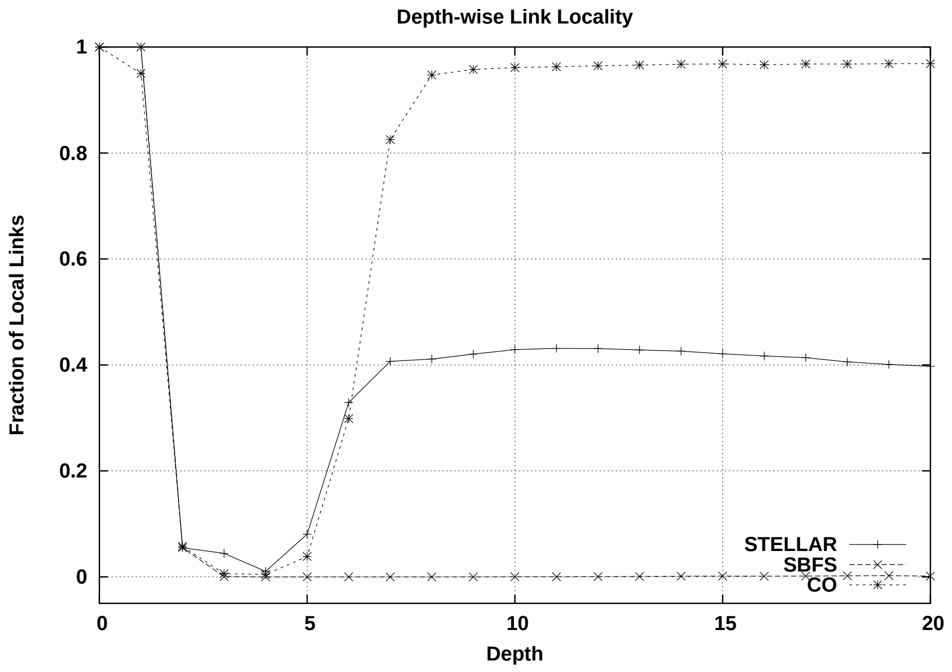


Figure 7.3: SPROT: Locality with varying Pagesize



(a) Edge Locality



(b) Link Locality

Figure 7.4: SPROT: Locality with varying Depth

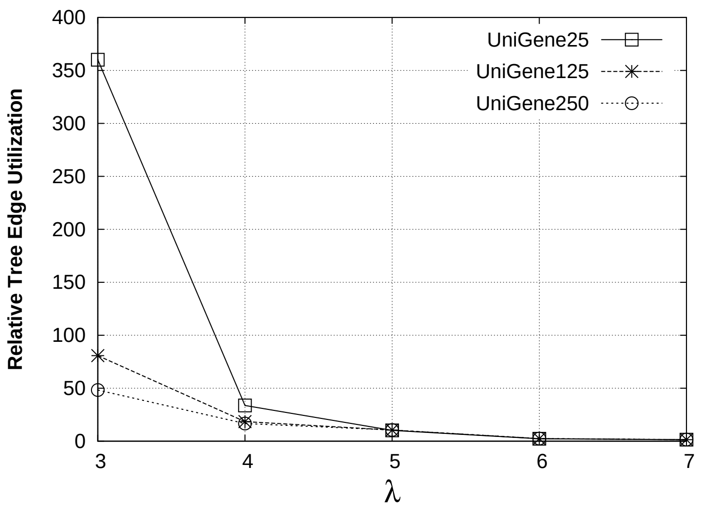


Figure 7.5: SPROT: Relative Edge Utilization

used in DNA database search, to account for the larger alphabet. Specifically, the λ values are much smaller – the BLASTP package uses default value of 3. As a result, the utilization of suffix-links during the search process is significantly less, and the search cost is dominated by the use of tree-edges for reporting of results.

The relative utilization of tree-edges with respect to suffix-links during the substring search task is illustrated in Figure 7.5. As these graphs indicate the tree-edge utility is significantly higher in the *typical operating range* of λ values. Thus, higher tree-edge locality is especially beneficial in the case of protein datasets.

Impact of Disk Layout

The relative performance of maximal substring search over persistent suffix-tree on SPROT dataset, laid out using Stellar against the CO layout is shown in Figure 7.6. For low values of λ, the I/O cost is dominated by the TraverseSubtree function. Due to the lack of locality of tree-edges in CO-layout it suffers from bad search performance in this range. Stellar on the other hand provides for good tree-edge locality, leading to significant I/O gains. Note that at $\lambda = 7$, the performance differential between CO and Stellar is very small – due to the fact that the average depth of the suffix-tree is ≈ 7.54, which results in negligible cost due to TraverseSubtree, even for CO layout. Thus, the main

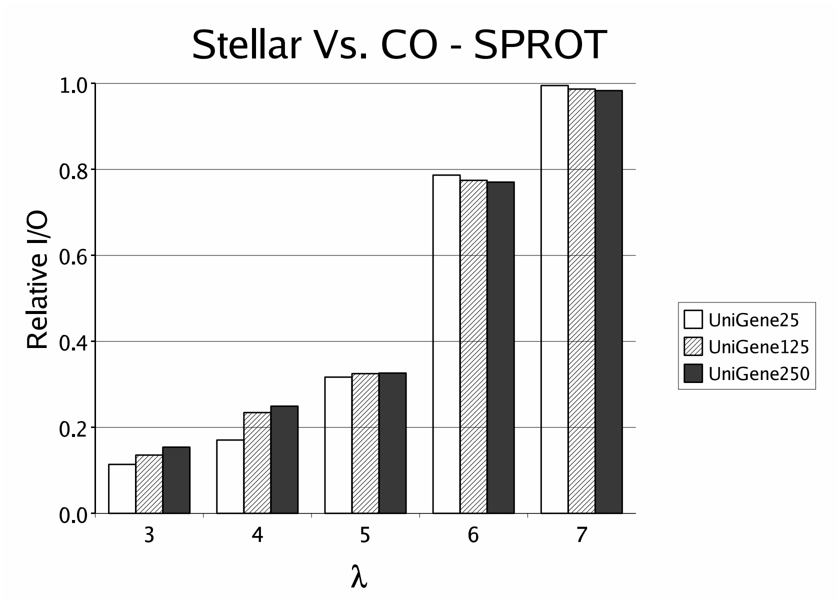


Figure 7.6: SPROT: Performance of Stellar over CO

component of I/O cost is that of the main search function in MaximalSubstringSearch, dominated by the suffix-link accesses. Since, CO-layout provides good suffix-link locality, the relative performance improvements due to Stellar are not very significant.

Stellar Vs. SBFS

Due to the increased tree-edge utilization in the operating range of λ values for protein substring similarity searches, it seems natural to expect SBFS which localizes only the tree-edges to perform much superior to Stellar.

The relative performance of Stellar over SBFS are shown in Figure 7.7. As expected, in the extreme low-value range of λ, the performance of SBFS is clearly superior to that of Stellar – however, with increasing length of the query sequence this performance gap decreases. As the value of λ increases, the cost of suffix-link traversals comes into prominence, and since Stellar optimizes these traversals, it starts to gain in performance over SBFS. For moderate values of λ (≥ 4), Stellar is quite competitive with SBFS.

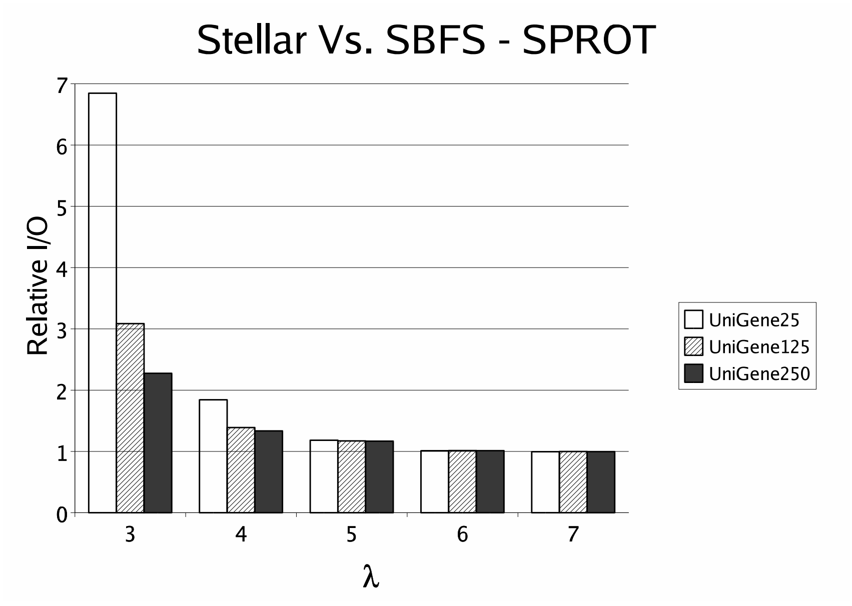


Figure 7.7: SPROT: Performance of Stellar over SBFS

7.3.2 Utility of Suffix-Links

Shifting our focus to the utility of suffix-links for large-alphabet sequences, it seems natural to expect that, due to the small value-range of λ, the UST-based methods hold advantage over LSTs. The USTs clearly provide better space economy (a reduction of 4-bytes per internal node), and TraverseSubtree being performed higher up in the tree the search cost is dominated by the tree-edge traversals.

Figure 7.8 plots the relative disk I/O performance of UST and LST, with increasing value of λ. Both the indexes, built over the SPROT data, are laid out using the SBFS strategy that is shown to optimize the performance. As these graphs indicate, the LST-based searches continue to provide improved disk I/O performance over searching using UST, despite small λ values and large alphabet-size. For example, with λ set to 3-4 (commonly used with BLASTP searches), the UST searching incurs about 80% extra disk accesses than LST. These results clearly indicate that benefits obtained due to the retention of suffix-links are independent of the alphabet-size.

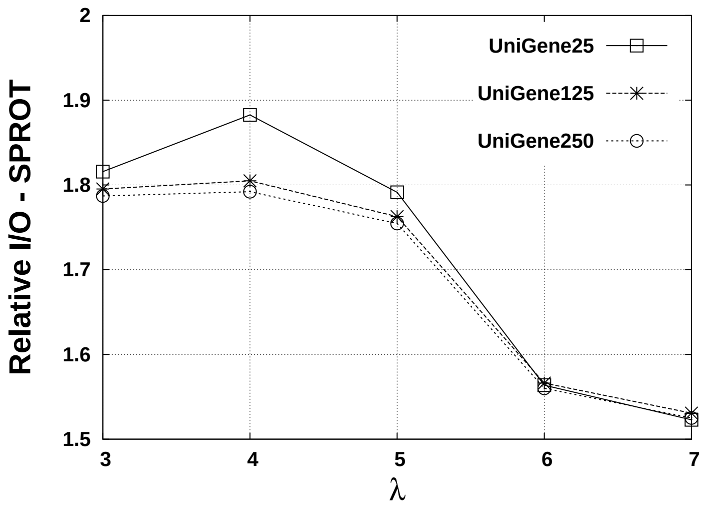


Figure 7.8: SPROT: UST Vs. LST

7.4 Conclusions

In this chapter, we investigated the applicability of techniques for speeding up persistent suffix-trees presented in the earlier chapters over *Protein sequences*, which have significantly different alphabet characteristics than DNA sequences.

We showed that the TOP-Q buffering strategy can be used also over protein sequences that have much larger alphabets. We also showed that even in the presence of skewed distribution of symbols in the sequence, the TOP-Q strategy is superior to other buffering strategies.

Next, we presented results to highlight the performance benefits gained through array representation of suffix-tree nodes over linked-list representation, in the presence of large alphabets, and with variable skew in the symbol distribution. These results showed that improved performance due to array representation is not limited to the small alphabets, thus justifying their use as a general physical implementation scheme for persistent suffix-trees.

Moving on to the evaluation of storage organizations, we showed that despite a significant skew in the utilization of tree-edges over that of suffix-links over protein sequence searches, Stellar organization is shown to very competitive to SBFS strategy. In comparison to the CO storage organization, Stellar saves close to 90% I/O during searching.

Again, these results bring out the applicability of our techniques across the spectrum of biological sequences.

Chapter 8

Performance Evaluation of BODHI

We have evaluated the performance of BODHI on a test-bed of typical queries in the biodiversity domain. These consist of queries over both *single-domains* (such as taxonomy, spatial or sequence domains) and *multiple domains* – i.e., queries similar to Query 1 presented in the Introduction. Moreover, since spatial data forms a large fraction of data and is traditionally considered by the biodiversity researchers to be the main component of the query processing time, we studied the performance of the spatial component in detail. In particular, we evaluated the spatial data handling capabilities of BODHI over the datasets and queries of the *SEQUOIA 2000* regional benchmark [118], a standard benchmark for spatial databases.

The performance numbers reported were generated on a Pentium-III 700MHz processor, with 512MB memory and an 18GB 10000-RPM SCSI hard disk (IBM DDYS-T18350M model), connected with Adaptec AIC-7896/7 Ultra2 SCSI host adapter. In order to reduce the effects of Linux's aggressive memory mapping of files, we flushed the benchmark database each time with an I/O over a large database.

The rest of the chapter is organized as follows: In Section 8.1, we describe the biodiversity datasets used in our experiments. Then, we present the performance profile of BODHI on these datasets for single and multi-domain queries in Section 8.2. The spatial data processing performance of BODHI, evaluated with SEQUOIA 2000 benchmark, is presented in Section 8.3.

8.1 Description of Datasets

Although there is no paucity of benchmarks for evaluating various functional and performance features of databases, many features required in biodiversity information systems are not handled by any benchmark in an unified manner. For example, the OO7 benchmark [20] is designed for solely evaluating object-oriented database performance. However, it does not have queries that take into account the effects of spatial and sequence operators in conjunction with object queries. Moreover, it does not exercise the *lengthy* sequence of joins due to long path-expression traversals that occur routinely in biodiversity workloads. As a result, we needed to develop a benchmark suite that closely models the range of data and query workload characteristics in biodiversity domain.

A serious hurdle that we faced in the design of the benchmark suite was the unavailability of large-scale digitized taxonomy data collections with the collaborating domain scientists, that can be effectively used in performance evaluation experiments. [1] This is because the domain experts we collaborated with have the bulk of their data in legacy formats – in many cases on "herbarium sheets"[2], and in text-books. While the digitization of this data is going on, we obtained the taxonomy data of about fifteen closely studied plant species that are marked endemic to the Western-Ghats region of Southern India, in a format that was amenable for loading. Table 8.1 lists the details of these select species, that form the basis of our benchmark data suite. This limited amount of data is scaled by boosting with synthetic data, generated with inputs from domain experts.

The data used in our experiments conforms to a biodiversity object model, which is presented in part as an object diagram in Figure 3.2. As shown in the object model, the schema is hierarchical in nature and consists of aggregation paths, inheritance structures over object types, spatial and genome sequence components. The well known taxonomy aggregation path of Order-Family-Genera-Species forms the backbone of the model. Each Species has a set of identifying characters (*IdentChar*), and there are many sub-

[1]Recently, a handful of web-based taxonomy data sources, that enable extracting the information stored in them, have come up. The domain scientists we collaborate with, are in the process of cleaning and curating this data, which can be populated into our system.

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

Order	**Family**	**Genera**	**Species**
Euphorbiales	Euphorbiaceae	Aporosa	bourdillonii
Euphorbiales	Euphorbiaceae	Aporosa	lindleyana
Laurales	Lauraceae	Actinodaphne	bourneae
Laurales	Lauraceae	Actinodaphne	lanata
Laurales	Lauraceae	Actinodaphne	lawsonii
Laurales	Lauraceae	Actinodaphne	malabarica
Laurales	Lauraceae	Appolonias	arnotti
Magnoliales	Magnoliaceae	Michelia	champa
Primulales	Myrsinaceae	Aridisia	sonchifolia
Polemoniales	Convolvulaceae	Ipomoea	campanulata
Rutales	Meliaceae	Aglaia	barberi
Rutales	Meliaceae	Aglaia	lawii
Rutales	Meliaceae	Aglaia	indica
Rutales	Meliaceae	Aglaia	jainii
Rutales	Meliaceae	Aglaia	simplicifolia
Rosales	Chrysobalanaceae	Atuna	travancorica

Table 8.1: Details of Endemic Plant-species

characteristics that are inherited from this. The spatial component of the model consists of a collection of reported habitat areas for each Species. Also associated with each Species is a collection of DNA sequences that are used to study the evolutionary pathways among the species by locating homologies (sequences which have a high likelihood of sharing a common ancestor). We describe the individual components of the benchmark dataset below:

Taxonomy Data: The real data available for about fifteen closely studied Plant species was scaled with synthetically generated data. The object relationships in taxonomy and characteristics hierarchies were generated through the use of heuristic probability of association at each optional relationship (uniform distribution in the range of 1-19 at each level of the hierarchy [87]). In case of collections in the aggregation path, the branch factor of the collection was also uniformly distributed.

Spatial Data We used the technique proposed in [67] to generate a synthetic 2-dimensional spatial data. The data consists of rectangular regions, whose centers are

Element	No. of Tuples	Overall Size(in KB)
Order	4	0.6
Family	46	7.1
Genera	496	76.0
Species	5155	1153.1
FlowerChar	5155	564.0
Habitats	5155	607.0
InfloChar	5	20.4
EMBLEntry	51550	2902
Compressed Genomic Sequence	51550	4743
Total		10073.2

Table 8.2: Statistics of the Synthetic Dataset

uniformly distributed over a unit square. The overlap between rectangular regions can be controlled by specifying the distribution of their height and width values. It should be noted that this dataset consists of only rectangular regions, while in reality we have to handle non-convex polygonal regions as well. The performance of spatial data handling over real dataset (involving non-convex polygonal regions) is evaluated separately through the SEQUOIA 2000 benchmark. Each species object generated above is associated with a synthetically generated polygon that represents the habitat of the species.

Genome Data In the case of Genome sequence data, we utilized publicly available data through the GenBank repository. In our experiments, we made use of a randomly selected sample of "expressed sequence tags" (ESTs) of various species available from the BLAST database of EMBL GenBank [41].

The statistics of the resulting benchmark dataset, which conforms to the schema illustrated in Figure 3.2, are summarized in Table 8.2. We consider a set of 5 queries over this dataset, spanning the domains of taxonomy, spatial and genome data, to illustrate the capabilities of BODHI in handling these domains. In addition, the performance numbers of these queries provide an indicator towards overall expected performance of the system. We use the *response time* as the metric of evaluation in our experiments.

Id	**Time**
Taxonomy Query-1	73 min. (Without Path-Dictionary)
	0.5 min. (With Path-Dictionary)
Genome Query-1	0.2 sec.
Genome Query-2	≈ 2.5 min. (Without Suffix-tree)
	4.5 sec. (With Suffix-tree)

Table 8.3: Performance Numbers for Single-domain queries

Id	**No Index**	**Path-Dictionary**	**Spatial& Path-Dictionary**	**Suffix-tree & Spatial & Path-Dictionary**
MDQ1	26.99 sec.	11.13 sec.	2.1 sec.	–
MDQ2	8275.66 sec.	8264.12 sec.	8252.2 sec.	135.2 sec.

Table 8.4: Performance Numbers for Multi-domain Queries

8.2 Biodiversity Queries

We now describe the set of queries considered to illustrate the capabilities of BODHI and present the performance numbers over each of these queries. The biodiversity query collection consists of two categories: (i) *Single-domain queries* – that are restricted to a single data domain (taxonomy, spatial or genome data), and (ii) *Multi-domain queries* – that combine multiple data domains in a single query. As we have explored the spatial query performance, in detail, with SEQUOIA 2000 benchmark in the subsequent section, we present only the taxonomy and genome sequence based queries under single-domain queries here.

The performance numbers for the queries are summarized in Tables 8.3 and 8.4.

8.2.1 Single-domain Queries

Taxonomy Query-1: *Find the names of all species that have the same Inflorescence characteristic in their Flowers as that of "Michelia-champa".*

With reference to the bio-diversity data model shown in Figure 3.2, this query performs a three level path traversal over the aggregation hierarchy of Species, Flower and Inflorescence Characteristics. The performance results in Table 8.3 for this query show that without any indexing strategy for accessing the aggregation paths, the query execution times are unacceptably high (73 minutes) – especially considering the modest size of the dataset. The reason for this high evaluation time of this query is due to the choice of a *nested loop join* used for evaluating this query. Clearly it would be more beneficial to choose either the sort-merge join or the hash join for the purpose. However, we were not able to use these due to the limitations imposed by the version of SHORE and λ-DB used in building BODHI. The hash join was not supported in λ-DB and the sorting of intermediate streams was not possible in the version of SHORE we used. The performance of the query execution improves by two orders of magnitude with the presence of a Path-Dictionary index over the queried path, taking only about 30 seconds. As discussed earlier in Section 3.4, the Path-Dictionary maintains a compact materialization of joins along the queried path, preventing the repeated computation of these expensive joins.

Genome Query-1: *Retrieve all DNA sequences of Michelia-champa.*

The DNA sequences are stored encoded, using context-free encoding, in a separate storage. This encoding increases the disk-memory bandwidth and enables the sequence similarity algorithms to operate in this encoded domain itself. At the same time, there is an overhead of decoding them before presenting to the user. The performance numbers for this query give an estimate of the delay involved in decoding these sequences.

Genome Query-2: *List names of all Species that have a DNA sequence within a BLAST score of 70 with any sequence of Michelia-champa.*

The computation of BLAST scores over a database could be a time consuming task, especially in the absence of any indexing strategy for speeding these queries. This is evident from the corresponding entries in the Table 8.3. The timing for this query – which results in 10 BLAST computations – is about 2.5 minutes! With a persistent suffix-tree index built on the sequence collection, this reduced dramatically to mere

4.5 seconds, an improvement of almost *30 times* in query execution speed.

8.2.2 Multi-domain Queries

Multi-domain Query-1: *Find the names of all Species sharing a common habitat and having the same Inflorescence characteristic as Michelia-Champa.*

This query, which is common among ecologists, is targeted at the combination of hierarchical data of Taxonomy domain, and associated Spatial data. The query evaluates the combined effectiveness of the Path-Dictionary index and R^*-Tree indexes available in BODHI. The performance numbers provided in Table 8.4 are for the optimal query plan which performs the spatial overlap before computing the joins over the aggregation paths. Since spatial overlap is highly selective in the existing dataset, the number of path aggregation traversals are reduced to a very small number. As a result, we see that even though this query is more complicated than *Taxonomy Query-1*, it takes less than 0.6% of time taken for *Taxonomy Query-1* even in the absence of the Path-Dictionary index. The presence of Path-Dictionary reduces the execution time further, from 26.99 seconds to 11.13 seconds – a reduction of 58%. In this case, the execution times are dominated by the spatial overlap computation. We can see this clearly by looking at the performance of the query when both R^*-Tree and Path-Dictionary indexes are present. The query time is just around 2 seconds, almost an 80% improvement. This clearly indicates that both indexing strategies are extremely useful for such queries.

Multi-domain Query-2: *Retrieve names of all pairs of Species sharing a common habitat, having same Inflorescence characteristic and having a DNA sequence within BLAST score of 70 of each other.*

This query, which extends the *Multi-domain Query-1* by adding an extra predicate for the BLAST score computation for each of the sequences in the target species, is similar to the “goal” query that we presented earlier as Query 1 in the Introduction.

Referring to Table 8.4, we see that the execution times without a suffix-tree index are close to *3 orders of magnitude* higher than those of *Multi-domain Query-1* – due to the additional 50 BLAST computations. On the other hand, the suffix-tree index based BLAST computation reduces the gap to within acceptable limits, evaluating the query in close to 2 minutes. The reduction in execution times due to other access-methods are approximately the same as in *Multi-domain Query-1*, about 11 seconds in presence of Path-Dictionary index and by a further 10 seconds in presence of both R^*-Tree and Path-Dictionary indices.

8.3 Evaluating Spatial Data Handling

The evaluation of queries over spatial data has traditionally been considered as a highly compute-intensive operation, and many indexing strategies have been proposed to improve the performance of these queries. The SEQUOIA benchmark has been quite popular for evaluating the performance and capabilities of spatial databases. It consists of a set of 10 queries over a schema involving spatial objects (such as polygons, points and graphs) and also bitmap (raster) objects. As we do not have support for bitmap data formats in BODHI, we have chosen to ignore the raster dataset and the queries (2),(3),(4) & (9), which involve these objects. The vector benchmark data consists of 62556 Point objects, 58585 Polygons and 201659 Graph objects. Table 8.5 summarizes the response times (in seconds) for the queries on this data. We have compared BODHI's performance with Paradise [30], a spatial database system also built on the SHORE storage manager, and Postgres [119], a popular free object-relational database. The numbers given for these two systems are taken from those reported in [30].

The SEQUOIA benchmark results in Table 8.5 show that BODHI is very close in performance to that of Paradise, which is a specialized and highly optimized spatial database system. Even though the hardware platform used by the two systems are difficult to compare, it should be noted that both Paradise and BODHI use the same underlying storage manager (Shore). In addition the following points regarding numbers reported under BODHI should be noted: (i) We use file-based storage management instead of using raw-disk as done by Paradise system; (ii) The optimal physical query plan is generated through a generic object-oriented query processor; (iii) The type-system is user-defined whereas in Paradise the basic type system of SHORE has been augmented; and, (iv) the size of the buffer pool used by SHORE is the default value – 320KB, whereas Paradise used 16MB.

Id	**BODHI (with R*-Tree)**	**BODHI (with Hil. R-Tree)**	**Paradise**
1	5742.0 (R*-Tree: 1342.0)	4662.0 (Hil. R-Tree: 262.0)	3613.0
5	0.12	0.11	0.2
6	8.0	8.0	7.0
7	0.66	0.7	0.6
8	9.7	9.6	9.4
10	11.0	10.8	*Not supported*

Table 8.5: SEQUOIA Benchmark numbers (in seconds)

We now present the chosen set of SEQUOIA queries and their performance statistics. We also explain a few of these queries and highlight their importance in a typical set of bio-diversity query workloads.

Sequoia 1 – Dataloading and Index creation. This query populates the database from a given set of datafiles, and is expected to exercise the bulk-loading facility in the database. At the time of writing, we do not a bulk-loading feature in BODHI, resulting in a transaction commit for each object hierarchy. Therefore, the table represents only an upper bound on the dataload and indexing times for the spatial component. Referring to Table 8.5, we see that this is the only benchmark query in which BODHI is far worse than Paradise which supports bulk-loading facility. However, we don't see it as a major bottleneck in BODHI, since the bio-diversity databases are not expected to have high rates of bulk data updates. Instead, these databases are highly query-intensive and hence it is important to have fast query processing speeds. In addition, we expect improvements in performance when a bulk-loading scheme is put in place for BODHI.

Sequoia 5 – Select a point based on its name.

Sequoia 6 – Select polygons overlapping a specified rectangle. This is one of the typical spatial queries asked in ecological studies where a geographic region is split into a set of grids and the researchers would want to identify the species whose previously recorded habitat boundaries overlap with the grid being studied. This could be important in identifying species whose co-existence in a region is to be targeted for study. The performance of spatial operators such as *overlap* depend directly on the performance of implementing these operators on a spatial index such as R^*-Tree or Hilbert R-Tree. Since the R^*-Tree implementation of BODHI is the same as that of Paradise (both use the index provided by the SHORE storage manager), we don't see much difference in the query execution performance.

Sequoia 7 – Select polygons greater than specified area, contained within a circle. We see similar queries occurring in bio-diversity studies with variations in the area selection clause of the query. The area of a polygon is provided through a derived attribute – computed based on the co-ordinates of the polygon. This is extendible to allow for selection over arbitrary derived attributes over which an index can be built. Thus, in ecological study databases, we get variations of the query that locate all the habitats that are near a study center, with a derived attribute value (such as bio-mass index of the habitat, etc.).

This query reflects the combination of B-Tree and spatial index based query processing. The order in which this query gets evaluated – whether the B-Tree lookup or the R^*-Tree based overlap selection is made as the first step – makes a big difference in the query answering times. The usage of query optimizer which maintains cost statistics and uses it to arrive at the final evaluation order is also tested in this query. The numbers presented in Table 8.5 are for the optimal plan generated by the query processor of BODHI, which is to perform the R^*-Tree based overlap selection first and then the B-Tree-based polygon area selection.

Sequoia 8 – Select polygons overlapping a rectangular region around a point.

Id	**Time**
11	3.36 sec.
12	51 sec.
13	66 min.

Table 8.6: Performance over Paradise Queries

Sequoia 10 – Select points contained in polygons with specific landuse type.

We also executed the above Sequoia benchmark queries with Hilbert R-Tree in place of R^*-Tree. The results obtained are shown in Table 8.5. The building times of Hilbert R-Tree were quite low in comparison to that of R^*-Tree, and at the same time provide almost the same performance. The numbers shown are for Hilbert R-Tree which employs s-to-(s+1) split policy on overflow, with $s = 2$. Even though the performance of the Hilbert R-Tree could be improved by increasing the value of s, the index creation times increase sharply with s. Hence, the current choice of split policy was chosen to optimize on the index building time and the performance of the index over benchmark queries.

In addition, BODHI also supports the spatial aggregate operator *Closest*, on the lines of Paradise spatial data management system. This operator was used in executing two spatial aggregate benchmark queries given by Paradise system, Query-11 and Query-12, in [96]. For completeness, we have also included Query-13, which is not an aggregation query, but is a spatial join in benchmark queries of Paradise. But we cannot compare the performance numbers obtained in BODHI with those reported in [96], as the benchmark datasets are completely different in both schema and the scale (they used 10 years of 8 Km. resolution AVHRR satellite images obtained from NASA, and DCW global data set containing information about roads, cities, land use, drainage properties etc.). Hence, we present the numbers in an absolute sense in Table 8.6.

Paradise 11 - Select closest graphs (polylines) to a given point. This query requires the evaluation of the spatial aggregate "Closest" using available index structures. This aggregation operator is implemented as an iterative searching for the closest polyline (*Graph* in Sequoia dataset). At each iteration step, a box is constructed around the given point, and all polylines that overlap with the box are located using the spatial index. If no polylines that satisfy this constraint are found, then dimensions of the box are increased and another iterative search is performed. When we obtain a non-null candidate set through this step, we compute exact distances between the point and the polylines in the candidate set, and the closest polyline is determined. The performance of this query depends heavily on the location of the point and the distribution of polylines in the region. If the polylines are densely packed, we obtain a non-null candidate set within a few iterations (most likely in the first iteration itself), thus getting the target polyline instantaneously. The performance numbers presented in Table 8.6 were obtained over a sample of 100 points from the Sequoia dataset.

Paradise 12 - Select closest graphs to every point. This query is an extension of the previous query, and performs a spatial aggregate on a cross product of two relations (in this case polylines and points). For each point in the Sequoia dataset, we compute the closest polyline, by running the previous query.

Paradise 13 - Select all polylines which intersect with each other. This query joins two large spatial relations and tests the efficiency of the system's spatial join algorithm. The cardinality of the polyline extents in Sequoia benchmark is very high, with 201659 graph objects in the dataset. In order to answer this query, we need to perform a self spatial-join of this extent, which is highly expensive. [3] This is clear from 66 minutes reported in Table 8.6, to answer this query.

8.4 Conclusions

In this chapter, we presented a detailed evaluation of the BODHI system, both in terms of the range of its querying capabilities as well as its performance profile. The complex multi-domain query, presented in the Introduction as Query 1 is shown to be computable

[3]Even in [96], the performance results, obtained with parallel disks and multiple processors, indicate that Query 13 takes an order of magnitude more time than Query 12.

in approximately *2 minutes*, in presence of the access-structures provided by the BODHI system – the Path-dictionary index for aggregation path traversals, Hilbert R-Tree for spatial data handling and the Persistent Suffix-tree over genomic DNA sequences. The same query would have taken more than 137 minutes, *without* utilizing any of the index structures. Further, we also presented a detailed evaluation of spatial data handling within BODHI, making use of the well-known SEQUOIA 2000 benchmark. These results showed that the BODHI system is comparable in performance and in the supported query repertoire to Paradise, a specialized spatial data management system.

Chapter 9

Conclusions and Future Research

9.1 Summary of Contributions

In this thesis, we have investigated the design and implementation of a holistic bio-diversity database, BODHI, that can be productively used by modern day biologists. This system addresses an urgent need for information management systems that can integrate a wide range of data associated with bio-diversity studies, including taxonomy information, spatial distributions, and genome sequence information. Focusing on accelerating the computationally expensive sequence processing capability of BODHI, we presented techniques to efficiently construct persistent suffix-trees using a combination of appropriate physical representation and a novel buffering strategy called TOP-Q that takes into account the behavior of traversals during suffix-tree construction. Further, we proposed a new storage organization, STELLAR, for persistent suffix-trees that caters to the combined traversal of tree-edges and suffix-links during searching over suffix-trees, and optimizes the disk I/O performance. We summarize our contributions in each of these areas below.

9.1.1 Design of BODHI

Modern bio-diversity studies generate and utilize a variety of inter-related data types forming deeply nested hierarchies. The queries that span these hierarchies need to perform multiple joins and, in many cases, the join-path could contain spatial or sequence similarity predicates, thus hindering efficient evaluation.

A multitude of tools have been deployed to handle each of the domains involved in isolation, and wrapper-based integration systems are being built to provide a functional integration. However, to the best of our knowledge, there has been no effort to support the diverse data-types efficiently under a single database platform. In this thesis, we presented the design and implementation of BODHI, a holistic approach to this problem of bio-diversity data unification.

BODHI is a native object-oriented database system that *seamlessly* integrates multiple types of data occurring in biodiversity studies. 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.

In addition to providing a functionally comprehensive query interface, BODHI achieves high performance by employing a variety of specialized access structures, such as *Multi-key Type Index*, *Path-dictionary Index*, *R***-Tree*, *Hilbert R-tree*, that are reported in the research literature for handling predicates over taxonomy hierarchies and spatial data. The Path-dictionary index was extended from its original proposal, to support N:M relationships as well as bags and sequences in the aggregation, a commonly found feature in the biodiversity schema. In addition, these indexes were implemented to satisfy the dual needs of efficiency and the ability to extend and improve the system. While these index structures are efficient in their respective domains, there are very few proposals for sequence indexing to accelerate a large class of biological sequence processing tasks.

In order to overcome the resulting performance bottleneck of sequence similarity queries, BODHI provides persistent version of suffix-trees, the ubiquitous main-memory sequence indexing structure. The persistent suffix-tree index is useful in a number of sequence querying applications, and provides an accurate indexing solution for biological sequences. We are not aware of any other database system that incorporates persistent suffix-trees as a first class sequence indexing strategy.

9.1.2 Efficient Construction of Persistent Suffix-tree Indexes

In the field of computational molecular biology, the suffix-tree has been considered a defacto index structure for biological sequence processing. However, its usage had been limited to small-sized datasets due to its large space requirements. With increase both in the volume of biological sequence data as well as the usage of sequence processing tasks, it has been considered essential to support suffix-tree indexes within persistent store – hitherto considered impractical using the standard construction algorithms.

We addressed this issue by designing a novel buffering policy called TOP-Q, that takes into account the nature of traversals during suffix-tree construction. In addition, we showed that the much preferred implementation of suffix-trees that uses a linked-list of sibling nodes is much more disk I/O intensive than a simpler array representation of suffix-trees – despite the increased space overhead due to the latter.

A significant advantage of our proposal is that all the existing suffix-tree based bioinformatics tools can be migrated to persistent store without having to reinvent or reimplement the algorithms. This is due to the fact that unlike alternate proposals for suffix-tree building [61, 123], we completely retain *all* the structural elements of suffix-tree. In particular, the *suffix-links* between internal nodes, which play an important role in linear time construction and subsequent querying over suffix-trees, are retained in our technique.

9.1.3 Storage Organization of Suffix-tree Indexes

Taking the next logical step in improving the utility of persistent suffix-trees, we addressed the issue of optimizing, in terms of disk I/O, the search tasks over the suffix-trees. We approached this issue in the same spirit as the previous work – i.e., to tune the parameters of the environment in which the suffix-trees are deployed, *without modifying either the structure or the algorithms over the index.*

Specifically, we presented a linear-time, top-down algorithm called Stellar, to reorder the persistent suffix-tree nodes such that the localities of both suffix-link and tree-edge based traversals during search is improved. We observed close to 60-70% reduction in I/O incurred during searches over DNA sequence collections when the suffix-tree is stored using Stellar strategy.

We also presented results to show that searching of persistent suffix-trees without utilizing suffix-links is far more expensive than the searches involving suffix-links. These results highlight the utility of retaining suffix-links in persistent suffix-trees.

9.2 Future Research

The work presented in this thesis can be extended in a number of ways, some of which are listed here:

1. **Multiple and Evolving Taxonomy.** The taxonomic organization is constantly evolving, due to the introduction of novel techniques for discovering the evolutionary and ecological relationships between organisms [99]. It is important to not only incorporate the ability to handle the resulting multiple taxonomies simultaneously, but also to track their lineage and discover inter-relationships between them. It would be necessary to expand the data modeling and querying capability of BODHI to be able to handle this requirement.

2. **Support for Distributed Data Repository.** With increasing volume and distribution of biodiversity information, it is inconceivable that a single data repository would be sufficient for supporting all the requirements of researchers. In order to cater to this need, distributed data handling capability can be added to the BODHI system. This support can be easily added since BODHI already provides shipping of OQL query results in XML format, the standard data interchange format over the Internet.

3. **Persistent Suffix-tree Support.** The techniques presented in this thesis for adding efficient suffix-tree index support in database kernels can be implemented in other popular database systems such as PostgreSQL [97] and MySQL [85].

References

[1] S. Altschul, W. Gish, W. Miller, E. W. Myers, and D. Lipman. A Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3), 1990.

[2] A. Andersson, N. J. Larsson, and K. Swansson. Suffix Trees on Words. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, 1996.

[3] ANZMETA DTD Version1.1. http://www.erin.gov.au/database/metadata/anzmeta/anzmeta-1.1.html.

[4] A. Apostolico, editor. *Combinatorial Algorithms on Words*, chapter The Myriad Virtues of Subword Trees. Springer Verlag, 1985.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[6] P. G. Baker, A. Brass, S. Bechhofer, C. Goble, N. Paton, and R. Stevens. TAMBIS—Transparent Access to Multiple Bioinformatics Information Sources. In *Proceedings of the Intelligent Systems for Molecular Biology*, 1998.

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

[8] S. Bedathur and J. Haritsa. Engineering a Fast Online Persistent Suffix Tree Construction. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004.

[9] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. GenBank: Update. *Nucleic Acids Research*, 32((Database Issue)), 2004.

[10] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 1(2), 1989.

[11] P. Bieganski. *Genetic Sequence Data Retrieval and Manipulation based on Generalized Suffix Trees*. PhD thesis, University of Minnesota, 1995.

[12] Biopolymer Markup Language - BIOML. http://bioinformatics.genomicsolutions.com/BioML.html.

[13] T. Boston and D. Stockwell. Interactive Species Distribution Reporting, Mapping and Modeling using the World Wide Web. In *Proceedings of the International WWW Conference (WWW)*, 1994.

[14] P. Buneman, S. B. Davidson, K. Hart, G. C. Overton, and L. Wong. A Data Transformation System for Biological Data Sources. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1995.

[15] X. Cao, S. C. Li, B. C. Ooi, and A. K. H. Tung. Piers: An Efficient Model for Similarity Search in DNA Sequence Databases. *SIGMOD Record*, 33(2), 2004.

[16] X. Cao, S. C. Li, and A. K. H. Tung. Indexing DNA Sequences Using q-grams. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005.

[17] X. Cao, B. C. Ooi, H. H. Pang, and K. L.Tan. DSIM: A Distance-based Indexing Method for Genomic Sequences. In *Proceedings of the IEEE International Conference on Bioinformatics and Bioengineering (BIBE)*, 2005.

[18] X. Cao, A. K. H. Tung, B. C. Ooi, K. L.Tan, and S. C. Li. String Join Using Precedence Count Matrix. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2004.

[19] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1994.

[20] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1993.

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

[22] W. I. Chang and E. L. Lawler. Approximate String Matching in Sublinear Expected Time. In *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 1990.

[23] H. Chou and D. Dewitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1985.

[24] R. Clifford and M. Sergot. Distributed and Paged Suffix Trees for Large Genetic Databases. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2003.

[25] A. Cobbs. Fast Approximate Matching using Suffix Trees. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, 1995.

[26] S. Davidson, C. Overton, and P. Buneman. Challenges in Integrating Biological Data Sources. *Journal of Computational Biology*, 2(4), 1995.

[27] S. B. Davidson, V. Tannen, J. Crabtree, G. C. Overton, B. P. Brunk, C. J. Stoeckert Jr., and J. Schug. K2/Kleisli and GUS: Experiments in Integrated Access to Genomic Data Sources. *IBM Systems Journal*, 40(2), 2001.

[28] M. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A Model of Evolutionary Change in Proteins. *Atlas of Protein Sequence and Structure*, 5, 1978.

[29] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27(11), 1999.

[30] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-Server Paradise. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1994.

[31] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering Techniques for Minimizing External Path Length. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, 1996.

[32] R. Durbin and J. Thierry-Mieg. A C.elegans Database Documentation. http://www.acedb.org/.

[33] W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems (TODS)*, 9(4), 1984.

[34] Environmental Information System. http://envis.nic.in/.

[35] T. Etzold and P. Argos. SRS: An Indexing and Retrieval Tool for Flat File Data Libraries. *Computer Applications in the Biosciences*, 9(1), 1993.

[36] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the Memory Bottleneck in Suffix Tree Construction. In *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 1998.

[37] M. Farach-Colton. Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 1997.

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

[39] P. Ferragina and R. Grossi. The String B-tree: a New Data Structure for String Search in External Memory and its Applications. *Journal of the ACM (JACM)*, 46(2), 1999.

[40] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[41] GenBank. http://www.ncbi.nlm.nih.gov/Genbank/.

[42] GenBank Statistics. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[43] *Getting to Know ArcView GIS for Version 3.1.* ESRI Press, 1999.

[44] R. Giancarlo. The Suffix-tree of a Square Matrix, with Applications. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[45] R. Giegerich and S. Kurtz. A Comparison of Imperative and purely Functional Suffix tree constructions. *Science of Programming*, 1995.

[46] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear Time Suffix Tree Construction. *Algorithmica*, 19(3), 1997.

[47] R. Giegerich, S. Kurtz, and J. Stoye. Efficient Implementation of Lazy Suffix Trees. In *Proceedings of the Third Workshop on Algorithmic Engineering (WAE 99)*, 1999.

[48] J. Gil and A. Itai. How to Pack Trees. *Journal of Algorithms*, 32(2), 1999.

[49] E. Giladi, M. G. Walker, J. Z. Wang, and W. Volkmuth. SST : An Algorithm for Searching sequence Databases in Time Proportional to the Logarithm of the Database Size. In *Proceedings of the International Conference on Research in Computational Molecular Biology (RECOMB)*, 2000.

[50] N. Goodman, S. Rozen, and L. Stein. A Glimpse at the DBMS Challenges Posed by the Human Genome Project. ftp://genome.wi.mit.edu/pub/papers/Y1994/challenges.ps.Z.

[51] N. Goodman, S. Rozen, and L. Stein. Building a Laboratory Information System around a C++-based Object oriented DBMS. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1994.

[52] G. Graefe, R. L. Cole, D. L. Davison, W. J. McKenna, and R. H. Wolniecwicz. Extensible Query Optimization and Parallel Execution in Volcano. In J.C. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Database Applications*. Morgan Kaufmann, 1993.

[53] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.

[54] D. Gusfield. Suffix Trees Come of Age in Bioinformatics (Invited Talk). In *IEEE Bioinformatics Conference (CSB)*, 2002.

[55] R. H. Güting. An Introduction to Spatial Database Systems. *VLDB Journal*, 3(4), 1994.

[56] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1984.

[57] L. M. Haas, J. E. Rice, P. M. Schwarz, W. C. Swope, P. Kodali, and E. Kotlar. DiscoveryLink: A system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2), 2001.

[58] J. Hammer and M. Schneider. Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[59] S. Henikoff and J. G. Henikoff. Amino Acid Substritution Matrics from Protein Blocks. *Proceedings of National Academy of Sciences USA*, 89, 1992.

[60] E. Hunt, M. Atkinson, and R. Irving. Database Indexing for Large DNA and Protein Sequence Collections. *VLDB Journal*, 7(3), 2001.

[61] E. Hunt, M. P. Atkinson, and R. W. Irving. A Database Index to Large Biological Sequences. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[62] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409, 2001.

[63] R. Japp. First Year Report. Master's thesis, University of Glasgow, July 2001.

[64] T. Johnson and D. Shasha. 2Q : A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1994.

[65] T. Kahveci and A. Singh. Progressive Searching of Biological Sequences. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 27(3), 2004.

[66] T. Kahveci and A. K. Singh. An Efficient Index Structure for String Databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[67] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1994.

[68] J. Kärkkaïnen and E. Ukkonen. Sparse Suffix Trees. In *Proceedings of the Annual International Conference on Computing and Combinatorics (COCOON)*, 1996.

[69] A. Kemper, C. Kilger, and G. Moerkotte. Function Materialization in Object bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1991.

[70] E. Keogh, S. Lonardi, and B. Chiu. Finding Surprising Patterns in a Time Series Database in Linear Time and Space. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.

[71] W. Kim, K. Kim, and A. Dale. Indexing Techniques for Object-oriented Databases. In W. Kim and F. Lochovsky, editors, *Object-oriented Concepts, Database and Applications.* Addison-Wesley Publishing Company (ACM Press), 1989.

[72] S. Kurtz. Reducing Space Requirement of Suffix Trees. *Software Practice and Experience*, 29(13), 1999.

[73] M. A. Lane, J. L. Edwards, and E. Nielsen. Biodiversity Informatics: The Challenge of Rapid Development, Large Databases, and Complex Data (keynote). In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[74] 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.

[75] D. J. Lipman and W. R. Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 227(4693), 1985.

[76] C. Low, B. Ooi, and H. Lu. H-trees: A Dynamic Associative Search Index for OODB. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1992.

[77] D. R. Maddison and W. P. Maddison. The Tree of Life: A multi-authored, distributed Internet project containing information about phylogeny and biodiversity. http://phylogeny.arizona.edu/tree/phylogeny.html, 1998.

[78] D. Maier and J. Stein. Indexing in an Object-oriented DBMS. In *Proceedings of the International Workshop on Object-oriented Database Systems*, 1986.

[79] U. Manber and G. Myers. Suffix Arrays: A New Method for On-line String Searches. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1990.

[80] E. M. McCreight. A Space-Efficient Suffix Tree Construction Algorithm. *Journal of the ACM (JACM)*, 23(2), 1976.

[81] C. Meek, J. M. Patel, and S. Kasetty. OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2003.

[82] T. Mitchellolds. Does Environmental Variation Maintain Genetic Variation - a Question of Scale. *Trends in Ecology & Evolution*, 7(12), 1992.

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

[84] W. E. G. Müller, F. Brümmer, R. Batel, I. M. Müller, and H. C. Schröder. Molecular biodiversity - case study: Porifera (sponges). *Naturwissenschaften*, 90(3), 2003.

[85] MySQL: The World's Most Popular Open Source Database.
http://www.mysql.org.

[86] L. Nakhleh, D. Miranker, F. Barbancon, W. H. Piel, and M. Donaghue. Requirements of Phylogenetic Databases. In *Proceedings of the IEEE International Conference on Bioinformatics and Bioengineering (BIBE)*, 2003.

[87] V. Nanjundiah and M. Gadgil. Personal communication, 1999.

[88] National Biodiversity Institute (INBio). http://www.inbio.ac.cr/en/default.html.

[89] G. Navarro and R. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. *Journal of Discrete Algorithms*, 1(1), 2000.

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

[91] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems (TODS)*, 9(1), 1984.

[92] Research Directions in Biodiversity and Ecosystem Informatics. Report of an NSF, USGS, NASA Workshop on Biodiversity and Ecosystem Informatics, 2001.

[93] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1993.

[94] R. J. Pankhurst. *Practical Taxonomic Computing.* Cambridge University Press, 1991.

[95] Paradise Team. Paradise: A Database System for GIS Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1995.

[96] J. M. Patel, J.-B. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. E. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, S. Guo, D. J. DeWitt, and J. F. Naughton. Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1997.

[97] PostgreSQL: The World's Most Advanced Open Source Database.
http://www.postgresql.org.

[98] Prometheus Project.
http://www.dcs.napier.ac.uk/ prometheus/.

[99] C. Raguenaud, M. Graham, and J. Kennedy. Two Approaches to Representing Multiple Overlapping Classifications: a Comparison. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2001.

[100] C. Raguenaud, J. Kennedy, and P. J. Barclay. The Prometheus Taxonomic Database. In *Proceedings of the IEEE International Conference on Bioinformatics and Bioengineering (BIBE)*, 2000.

[101] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 2nd edition, 2000.

[102] S. Ramaswamy and P. C. Kanellakis. OODB Indexing by Class Division. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1995.

[103] J. T. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1981.

[104] H. Saarenmaa. The Global Biodiversity Information Facility: Architectural and Implementation Issues. Technical Report TR-34, European Environment Agency, 1999.

[105] H. Saarenmaa, S. Leppäjärvi, J. Perttunen, and J. Saarikko. Object-oriented Taxonomic Biodiversity Databases on the World Wide Web. In A. Kempf and H. Saarenmaa, editors, *Internet Applications and Electronic Information Resources in Forestry and Environmental Sciences.* European Forest Institute, 1995.

[106] G. M. Sacco. Index Access with Finite Buffer. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1987.

[107] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, 1990.

[108] Scaleable Similarity Searching. http://www.research.ibm.com/compsci/compbio/scaleable.html.

[109] J. L. Schnase, J. Cushing, M. Frame, A. Frondorf, E. Landis, D. Maier, and A. Silberschatz. Information Technology Challenges of Biodiversity and Ecosystem Informatics. *Information Systems*, 28(4), 2003.

[110] K.-B. Schürman and J. Stoye. Suffix Tree Construction and Storage with Limited Main Memory. Technical Report 2003-06, Universität Bielefeld, 2003.

[111] P. H. Sellers. The Theory and Computation of Evolutionary Distances. *Journal of Algorithms*, 1, 1980.

[112] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[113] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 1(147), 1981.

[114] Species2000. http://www.species2000.org/.

[115] B. Sreenath and S. Seshadri. The hcC-Tree: An Efficient Index Structure for Object Oriented Databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1994.

[116] S. M. Stephens, J. Y. Chen, M. G. Davidson, S. Thomas, and B. M. Trute. Oracle Database 10g: a Platform for BLAST search and Regular Expression Pattern Matching in Life Sciences. *Nucleic Acids Research*, 33(Database issue):675–679, 2005.

[117] R. Stevens, C. Goble, P. Baker, and A. Brass. A Classfication of tasks in bioinformatics. *Bioinformatics Journal*, 17(2):180–188, 2001.

[118] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The SEQUOIA 2000 Storage Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1993.

[119] M. Stonebraker and L. A. Rowe. The Design of Postgres. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1986.

[120] SWISS-PROT Protein Knowledgebase. http://www.expasy.org/sprot/.

[121] W. Szpankowski. *Average Case Analysis of Algorithms on Sequences.* Wiley-Interscience, 2001.

[122] Z. Tan, X. Cao, B. C. Ooi, and A. K. H. Tung. The *ed-tree*: An Index for Large DNA Sequence Databases. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2003.

[123] S. Tata, R. A. Hankins, and J. M. Patel. Practical Suffix Tree Construction. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[124] M. A. Thomas and R. Klaper. Genomics for the Ecological Toolbox. *TRENDS in Ecology and Evolution*, 19(8), 2004.

[125] E. Ukkonen. Approximate String Matching over Suffix Trees. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 1993.

[126] E. Ukkonen. Online Construction of Suffix-trees. *Algorithmica*, 14(3), 1995.

[127] Unified Modeling Language (UML). http://www.uml.org/.

[128] UniGene: Organized view of the transcriptome. ftp://ftp.ncbi.nih.gov/repository/UniGene/.

[129] P. Valduriez. Join Indices. *ACM Transactions on Database Systems (TODS)*, 12(2), 1987.

[130] P. Weiner. Linear Pattern Matching algorithms. In *Proceedings of the IEEE Symposium on Switching and Automata Theory*, 1973.

[131] H. E. Williams. CAFE: an Indexed Approach to searching Genomic Databases. In *Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval (SIGIR)*, 1998.

[132] H. E. Williams and J. Zobel. Indexing and Retrieval for Genomic Databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1), 2002.

[133] World Conservation Monitoring Center. http://www.unep-wcmc.org/.

[134] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, 1990.