Efficient Index-based Methods for Processing Large Biological Databases

by

You Jung Kim

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Efficient Index-based Methods for Processing Large Biological Databases

by

You Jung Kim

Chair: Jignesh M. Patel

Over the last few decades, advances in life sciences have generated a vast amount of biological data. To cope with the rapid increase in data volume, there is a pressing need for efficient computational methods to query large biological datasets. This thesis develops efficient and scalable querying methods for biological data.

For an efficient sequence database search, we developed two q-gram index based algorithms, miBLAST and ProbeMatch. miBLAST is designed to expedite batch identification of statistically significant sequence alignments. ProbeMatch is designed for identifying sequence alignments based on a k-mismatch model. For an efficient protein structure database search, we also developed a multi-dimensional index based algorithm method called proCC, an automatic and efficient classification framework.

All these algorithms result in substantial performance improvements over existing methods.

When designing index-based methods, the right choice of indexing methods is essential. In addition to developing index-based methods for biological applications, we also investigated an essential database problem that reexamines the state-of-the-art indexing methods by experimental evaluation. Our experimental study provides a valuable insight for choosing the right indexing method and also motivates a careful consideration of index structures when designing index-based methods.

In the long run, index-based methods can lead to new and more efficient algorithms for querying and mining biological datasets. The examples above, which include query processing on biological sequence and geometrical structure datasets, employ index-based methods very effectively. While the database research community has long recognized the need for index-based query processing algorithms, the bioinformatics community has been slow to adopt such algorithms. However, since many biological datasets are growing very rapidly, database-style index-based algorithms are likely to play a crucial role in modern bioinformatics methods. The work proposed in this thesis lays the foundation for such methods.

CHAPTER I

Introduction

Over the last few decades, advances in life sciences have generated a vast amount of data. Even today, the volume of data is growing explosively, and in some cases, at a rate faster than the Moore's law. For instance, GenBank, a repository for biological sequences, has been increasing exponentially, doubling in size between 12 and 15 months. Protein structure and sequence databases such as PDB [2] and PIR [3] have also grown very rapidly (The number of structures in PDB today is roughly double the number of structures in the year 2000).

To cope with this rapid increase in data volume, there is a pressing and growing need for extremely efficient computational methods to analyze these large datasets. Existing algorithms designed for handling these datasets are already computationally intensive and they are increasingly having difficulties in processing a large volume of data. As a result, in many data intensive life science research applications, the computational methods are increasingly becoming a limiting factor in making research progress. For instance, SCOP and CATH are protein structure classification databases produced from raw data in PDB and they are updated based on various automated methods and manual interpretation. However, updates to SCOP and CATH are done only intermittently and the speed of updates does not keep up with the fast data deposition rate in PDB. This kind of problem is not limited to protein structure data. Rather, it is often common in many bioinformatics applications.

So far, computational tools and methods have been developed with a focus on functionality rather than on efficiency and scalability. However, with the amount of data increasing at a fast rate, such tools and methods do not scale to handle an ever growing database. Not surprisingly, we see that there is an increasing need for efficient and scalable methods for handling large biological datasets.

In this thesis, we describe our efforts in developing efficient and scalable querying methods for biological data. Since many biological datasets are large and are rapidly growing in size, it is natural to consider the use of index-based methods for efficient query processing. Therefore, the bulk of this thesis focuses on developing index-based methods for querying large biological datasets.

Index-based methods already have been used in several bioinformatics applications. Due to its efficiency in finding exact matches, q-gram index has been popularly used in sequence search applications [9, 51]. However, in those applications, the q-gram index has been used in a naive way based on an in-memory q-gram index and a simple look-up operation. The use of an in-memory index is easy to implement, but it has a central problem in that it cannot scale well for a large sequence data. With a better understanding on the q-gram index and the use of database-oriented indexing techniques, we show how to build a scalable and efficient algorithm using a disk-based q-gram index in Chapter II and III.

In Chapter II, we present an example of a scalable and efficient algorithm for a sequence database search. A common task in modern bioinformatics applications is to match a set of nucleotide query sequences against a large sequence database. Existing tools are designed to evaluate a single query at a time and can be unacceptably slow when the number of sequences in the query set is large. To solve this problem, we designed a new algorithm, called miBLAST, which evaluates such batch workloads efficiently. At the core, miBLAST employs a disk-based q-gram index and an index join for efficiently detecting similarity between query sequences and database sequences. This set-oriented technique, which indexes both the query and the database sets, results in substantial (about 22X) performance improvements over existing methods.

In Chapter III, we present another q-gram based sequence database search algorithm, called ProbeMatch. ProbeMatch is designed to handle a large amount of short query sequences generated by new generation sequencing technologies. Different from miBLAST, which finds statistically significant sequence alignments, ProbeMatch finds sequences alignments based on an exact k-mismatch model, and utilizes various gapped q-gram filtering and indexing techniques to speedup sequence alignments. Unlike existing tools which only performs ungapped alignments, ProbeMatch generates both ungapped and gapped alignments, and shows its efficiency and effectiveness by producing more alignments in a faster time.

As shown in Chapter II and III, biological data such as sequences can be represented using a set of short q-grams, and handled using a relatively simple q-gram index and an exact q-gram lookup operation. However, often a significant fraction of biological data needs to be modeled as a more complex multi-dimensional dataset, requiring approximate search operations.

In Chapter IV, we present an algorithm that models biological data as a multi-dimensional data set and utilizes an multi-dimensional index structure. Protein structure classification plays a central role in understanding the function of a protein molecule. With the rapid increase in the number of new protein structures, the need for automated and efficient methods for protein classification is increasingly important. Recognizing this need, we developed a method called proCC, an automatic and efficient classification framework, by transforming protein structures into multi-dimensional points and employing multi-dimensional index-based methods such as R*-tree to quickly retrieve similar protein structures. With no significant difference in overall accuracy, proCC provides a clear advantage over existing methods with respect to computational cost. It performs a classification by an order of magnitude (about 13X) faster than an existing method.

Although we have developed various index-based algorithms for searching biological databases, a fundamental question remains: What is a good choice for an index structure when we design an index-based application? Incidentally, this question is also interesting for non-biological data management problems. To answer the above question, we also investigate an essential database problem which reexamines the state-of-the-art multi-dimensional point index structures and challenges the conventional wisdom regarding the choice of index structures.

Chapter V and VI present an experimental study comparing the R*-tree and Quadtree. In particular, we extensively compare the performance of different indexing methods using a variety of query operations such as range query, k nearest neighbor query, and distance join query. Although a variety of query operations can be performed using these index structures, previous work has largely focused only on the range search operation. We go beyond this previous work and compare the performance of these index structures using various query operations. In addition, we also consider the impact of index construction methods in evaluating these index structures. Our study sheds light on how the choice of the underlying index structure affects the performance of different query operations, and shows that the method used for constructing the index and the dynamic nature of the dataset has a dramatic impact on the performance of these index structures.

CHAPTER II

miBLAST: scalable evaluation of a batch of nucleotide sequence queries

2.1 Introduction

A common query in a number of bioinformatics applications is to search a large nucleotide sequence database using a *set* of nucleotide sequence queries. For example, when validating the Affymetrix (oligonucleotide) probe set against the Unigene (EST) database, one needs to search the quarter million Affymetrix probes against the most recent Unigene release. Another example is using one animal model microarray against a different species, searching the chip probe set against the ESTs of the new species to validate the probes in the new species [58, 84]. A final example is in designing small interfering RNAs (siRNAs) libraries, where one needs to validate that the siRNAs only interfere with a single mRNA. A common characteristics of these types of applications is that a large *batch workload* of queries must be evaluated against a large database. Often the databases that being are searched in such scenarios are updated frequently (such as the periodic updates to GenBank), which requires periodic re-evaluation of these batch workloads.

One way of evaluating such batch workloads, is to execute each query in the workload one at a time using a local-sequence alignment tool such as BLAST [8, 9]. However in practice, with large batch sizes this method is computationally very expensive. Clearly, a tool that can significantly speed up the evaluation of such workloads is very valuable. The focus of this paper is on the design of such a tool called miBLAST (pronounced as “me-BLAST”)

We note that a number of previous research investigations have developed techniques for efficiently evaluating batch workloads. These tools[15, 51, 72] are designed for specialized biological applications, such as aligning ESTs to a genome of similar species, and improve performance by using an index of non-overlapping q-grams. However, these approaches often sacrifice some loss in sensitivity for performance gains. MegaBLAST [89] uses a greedy algorithm and can be an order of magnitude faster than regular BLAST. However, the use of MegaBLAST is limited in aligning highly similar sequences with large word sizes. MPBLAST [56] directly improves the speed of NCBI BLAST and WU-BLAST (http://blast.wustl.edu) by *multiplexing* query sequences, thus reducing the number of database searches. MPBLAST essentially concatenates the queries in the workload and sends a single long query to BLAST.

A new feature in NCBI BLAST essentially implements the MPBLAST-like multiplexing technique for batch queries, and produces alignments that are identical to running the queries in the batch one at a time. BLAST++ [86] exploits commonality of short words amongst queries and shares results with queries containing these common words. Consequently, the performance of BLAST++ is highly dependent on the level of commonality in queries. Another method for speeding up the processing of batch workloads is to make use of parallel processing techniques on a cluster of machines using the mpiBLAST [24] method. This method essentially parallelizes BLAST searches by segmenting a database and executing each segmented portion of the database in a node in the cluster. The drawback of this approach is that it requires access to a cluster.

In this work, we propose an efficient and practical method for evaluating a *batch workload* which consists of a large number of queries. Furthermore, given the popularity and familiarity of existing users with BLAST, we want our tool to exactly mimic the behavior and functionality of BLAST. In other words, we want our tool to have the same sensitivity as BLAST and employ the same statistical model and data formats for input and output.

While miBLAST is a general algorithm for sequence similarity matching between a batch workload and a database of sequences, it is especially useful in settings where the database and the workload consists of a large number of sequences (rather than a few very long sequences). Common examples of relevant biological applications of miBLAST include evaluating a large number of oligonucleotide probes, cDNA sequences, or ESTs against a large database of ESTs.

In order to develop a more suitable algorithm for efficiently evaluating batch workloads, it is first necessary to identify the reasons why the BLAST algorithm is not efficient for such batch workloads. The BLAST algorithm essentially scans the entire database for *each* query sequence. In each scan of the database, the algorithm checks each database sequence to find any common short words between the database and the query sequences. In case of a *word hit* the word alignment is extended to produce a complete alignment. Consequently, to evaluate a batch workload with n queries, the BLAST method will require n scans of the database, and will compare each sequence in the batch with each sequence in the database.

This observation enables us to design the miBLAST algorithm which speeds up the evaluation of batch workloads. At the core, the algorithm employs a q-gram filtering and an index join technique that processes two q-gram indices [70]. In a single scan of the two indices, the join method efficiently computes an initial word hit list for *all* the query sequences. The join also determines a set of *filtered* database sequences that have potential matches for the queries. Then, miBLAST only examines these filtered database sequences to produce the actual alignments. In practice, only a few database sequences match a given query, and consequently miBLAST doesn't have to examine the majority of sequences in the database.

The use of the q-gram index has also been explored in several sequence searching applications [15, 51, 72]. However, these applications are limited by the database or the main memory size, since they require that the whole index resides in main memory. In contrast, our work uses disk-based indices and can work with arbitrary large data sets.

We have evaluated miBLAST on an a number of actual batch workloads, including a workload to validate the labels of the Affymetrix probes (Human Genome U133A Set) with the current version of Unigene. The Affymetrix probe set consists of approximately two hundred and fifty thousand queries, which are on average 25 nucleotides long. Each sequence in this workload is searched against the UniGene Homo sapiens data set, which has 5,064,621 sequences and approximately 3.19 gigabases. Using the default parameters in the NCBI BLAST (including a word size of 11), this workload can be evaluated on a single machine in 27.27 days (9.50 seconds per query). In contrast, miBLAST processes this same workload on the same machine in 1.26 days (0.44 seconds per query). The resulting performance improvement achieved by miBLAST in this case is about a factor of 22. However, depending on the choice of parameters, miBLAST can outperform BLAST by even larger margins. For example, using a word size of 23 can result in a 45 fold performance improvement over regular BLAST. These results clearly demonstrate the effectiveness of our search tool.

Finally, miBLAST is built as an module that is integrated with the existing NCBI BLAST source code. This design allows miBLAST to reuse the alignment extension and result formatting components of the BLAST code base. As a result, miBLAST employs the traditional BLAST statistical model and outputting format. Since many users are very familiar with these aspects of BLAST, we expect that current BLAST users who need to evaluate batch workloads can easily transition to using miBLAST.

2.2 Algorithms and Methods

The key strategy employed by miBLAST is to use q-gram indices to quickly identify the set of database sequences that contain word hits for each query sequence. Since for any query sequence, only a small fraction of the database sequences actually

Table 2.1: An example of a q-gram index structure.

sequence ID	sequence	word(w)	sequence ID
1	ACAAAAA	AAA	1 2
2	AAAAAAAAC	AAC	2 3 4
3	CAACAACAA	ACA	1 3 4
4	CAACAACAA	CAA	1 3 4

The left column represents a sequence data set and the right column shows the index built on the data set using $l = 3$.

have a common word hit, miBLAST only has to examine the small set of sequences that are *filtered* through the index search. These filtered sequences are then retrieved and the word hits are expanded to produce the actual alignments. In the following sections, we first describe the q-gram index structure, and then we describe the miBLAST filtering algorithms in detail.

2.2.1 Notations

The miBLAST algorithm considers two sets of sequences: a query sequence set $Q = \{q_1, q_2, ..., q_i, ..., q_m\}$ and a database sequence set $D = \{d_1, d_2, ..., d_j, ..., d_n\}$, where q_i and d_j represent sequences. All sequences in Q and D are assigned a unique sequence ID. A word w is defined as a string having a fixed length l. A word hit is defined as an ordered pair (i, j) such that the query sequence q_i and a database sequence d_j share a word in common.

2.2.2 Index Structure and Construction

The structure of the miBLAST q-gram index [70] is as follows: The q-gram index over a set of sequences contains an entry for every unique *overlapping* word of length l in the set. Each index entry stores the list of sequence IDs that contain the corresponding index word. Note that the reference to the sequence is simply based on the sequence ID and does not include the offset in the sequence where the word is located. Consequently, if a word appears multiple times in a given sequence, the index entry only refers to the sequence once. This decision to not store the actual offset was made to reduce the size of the miBLAST indices. (The algorithm can be generalized to work with a q-gram index in which the offset position is also stored.) An example of a q-gram index is shown in Table 2.1.

A key advantage of this index structure is its efficiency. The index construction time is linear ($O(n)$) in the size of an input sequence set. In addition, searches for a word hit is very efficient as it takes constant time ($O(1)$).

Since the q-gram index has one index entry for every unique word of length q, the size of the index can be large, and can exceed the size of the available main memory. To overcome this problem, miBLAST employs a disk-based implementation of the q-gram index. The index is stored on disk and index entries are fetched from disk on demand. In addition, miBLAST is designed so that the index accesses are largely sequential (as opposed to the much more expensive random I/O). We note that the use of disk-based q-gram index in miBLAST is in contrast to other q-gram based approaches [51, 72], which employ an in-memory index scheme. Since the q-

gram index is typically larger than the actual database, the use of in-memory indices implies that the q-gram method of [51, 72] can not be used with very large databases.

We also note that the q-gram index used by miBLAST indexes all overlapping words. This scheme ensures that the miBLAST algorithm has no loss in sensitivity over the BLAST method. In contrast, the use of *non-overlapping* words in other approaches [51, 72] results in a loss in sensitivity, but produces a smaller index. We feel that the use of a larger index structure in miBLAST is justifiable since one of the design goals of miBLAST to provide the same sensitivity as BLAST.

Furthermore, to ensure that there is not a big performance penalty in using disk-based indexes, miBLAST makes careful use of sequential accesses. The index entries are sorted by the logical file offset positions provided by the operating system, and miBLAST accesses these logical index blocks sequentially. Such logical sequential access often closely corresponds to a physically sequential disk access since operating systems try to store the data in a file in physically contiguous disk blocks. This technique of using sequential access has important performance implications since scanning physically adjacent disk blocks is much more efficient than randomly accessing disk blocks that are spread across the disk. In addition, operating systems often prefetch adjacent blocks of data, which further improves the performance of sequential accesses.

Word length is a critical BLAST parameter, which specifies the size of a word that is used to detect a word hit. Since different queries may specify different word length, miBLAST must employ a strategy for dealing with different word lengths. One naive strategy that could be employed is to build a q-gram index for all possible word lengths. However, the cost of building all such indices can be prohibitively large. To avoid this high cost, miBLAST employs methods that allow an index to be reused even with queries that specify a different word lengths.

In fact this flexibility in the use of the q-gram index for different query word lengths is a key difference between the use of q-gram index in miBLAST and previous q-gram-based approaches [51, 72]. With miBLAST in practice one could store just a single q-gram index based on the smallest supported word length parameter. Alternatively, we could choose to maintain a small number of q-gram indices and for a given query pick the q-gram index with the q value that is closest to the specified query word size.

2.2.3 The miBLAST Algorithm

The miBLAST algorithm consists of three primary steps. First, two q-gram indices are constructed - one on the query set and the other on the database set. Second, using these indexes, the filtering algorithm selects database sequences that contain potential word hits for each query sequence. Finally, the BLAST alignment module is invoked on the filtered database sequences to generate the actual alignment.

The miBLAST method is outlined in Algorithm 1. miBLAST takes as input a set of query sequences, a set of database sequences, and a word length to be used in a BLAST search. Based on this input, miBLAST starts by building a database index. In practice, the database index is often pre-built and is stored on disk so that the same index can be reused for many searches.

Notice that in lines 3-7 of Algorithm 1 miBLAST runs different filtering algorithms

Algorithm 1 miBLAST(Q, D, l)

```
INPUT:
 • $Q = \{q_1, q_2, ..., q_i, ..., q_m\}$ is a set of query sequences
 • $D = \{d_1, d_2, ..., d_j, ..., d_n\}$ is a set of database sequences
 • $ID$ is a unique number assigned to each sequence, $q_i$ and $d_j$ in $Q$ and $D$
 • $m$ is the word length used in database index construction
 • $l$ is a word length in a BLAST search
VARIABLES:
 • $D_{ID} = \{1, 2, ....., n\}$ is a set of database sequence IDs in D
 • $F_i \subseteq D_{ID}$ is a set of sequence IDs that have word hits with $q_i$
 • $Filtered = \{F_1, F_2, ....., F_m\}$ is a set of $F$ for all $q_i$ in $Q$
 • $DIndex$ is a $q$-gram index for $D$
 • $QIndex$ is a $q$-gram index for $Q$
 • $DIndex.L(w)$ is an operation returning the list of sequences
   containing a word $w$ from DIndex.

1: Build a $DIndex$ on $D$ with a word length $m$, if the index doesn't already exist.
2:
3: if $l \leq m$ then
4:    $Filtered$=INDEX-JOIN FILTER($DIndex, Q, l, m$)
5: else if $l > m$ then
6:    $Filtered$=SLIDING-WINDOW FILTER($DIndex, Q, l, m$)
7: end if
8:
9: for all sequence $q_i$ in $Q$ do
10:    for all sequenceID $j$ in $F_i$ of $Filtered$ do
11:       Find alignments with a $q_i$ in $Q$ and a $d_j$ in $D$
12:    end for
13: end for
```

depending on the word length l. If the query word length l is smaller or equal to the index word length m, then miBLAST uses an index join algorithm for sequence filtering (shown in Algorithm 2). However, if $l > m$, then miBLAST uses the sliding window filtering method which is shown in Algorithm 3.

Algorithm 2 INDEX-JOIN FILTER($DIndex, Q, l, m$)

```
1: Build a $QIndex$ on Q with a word length $l$
2: Create a $bitmap[|Q|, |D|]$
3:
4: for all word $w$ in $QIndex$ do
5:    if $QIndex.L(w) \neq$NULL & $DIndex.L(w) \neq$ NULL then
6:       $L_q \leftarrow QIndex.L(w)$
7:       $L_d \leftarrow DIndex.L(w)$
8:       for $i \leftarrow 1$ to $|L_q|$ do
9:          for $j \leftarrow 1$ to $|L_d|$ do
10:            $bitmap[L_q[i], L_d[j]] =$ TRUE
11:         end for
12:      end for
13:   end if
14: end for
15:
16: for $i \leftarrow 1$ to $|Q|$ do
17:    for $j \leftarrow 1$ to $|D|$ do
18:       if $bitmap[i, j] =$ TRUE then
19:          $F_i = F_i \bigcup \{j\}$
20:       end if
21:    end for
22: end for
23:
24: $Filtered = \{F_1\} \cup \{F_2\} \cup ... \cup \{F_m\}$
25: return $Filtered$
```

Index Lookup

The index lookup operation, $L(w)$, returns a list of sequence IDs containing a query word w of length l. When the query word length is the same as the index word length, i.e. $l = m$, the list can be retrieved by a single index lookup. However, when $l < m$, the index lookup function performs multiple index lookups. These lookups search for all index words whose prefix matches the entire query word. The final result list is constructed by simply computing a union of the sequence lists returned by each lookup. Since our q-gram index is lexicographically sorted, these multiple

Algorithm 3 SLIDING-WINDOW FILTER($DIndex, Q, l, m$)

```
1: Create a $bitmap[|Q|, |D|]$
2: Create a $counter[|D|]$
3:
4: for all sequences $q_i$ in $Q$ do
5:    for all words $w$ of length $l$ in $q_i$ do
6:       Initialize the $counter$
7:       for all all substrings $w'$ of length $m$ in $w$ do
8:          if $DIndex.L(w') \neq$ NULL then
9:             $L_d \leftarrow DIndex.L(w')$
10:            for $j \leftarrow 1$ to $|L_d|$ do
11:               $counter[L_d[j]] + +$
12:            end for
13:         end if
14:      end for
15:      for $j \leftarrow 1$ to $|D|$ do
16:         if $counter[j] = l - m + 1$ then
17:            $bitmap[i, j] =$ TRUE
18:         end if
19:      end for
20:   end for
21: end for
22:
23: for $i \leftarrow 1$ to $|Q|$ do
24:    for $j \leftarrow 1$ to $|D|$ do
25:       if $bitmap[i, j]$ =TRUE then
26:          $F_i = F_i \bigcup \{j\}$
27:       end if
28:    end for
29: end for
30:
31: $Filtered = \{F_1\} \cup \{F_2\} \cup ... \cup \{F_m\}$
32: return $Filtered$
```

index lookups essentially result in a sequential index scan, making the multiple index lookup step very efficient.

Index Join Filtering

In order to find all word hits between the query sequence set Q and the set of database sequences D, miBLAST employs an index join technique. Two indexes, one for each D and Q, are built. After the index construction is completed, miBLAST joins these indexes based on words. In our implementation of the q-gram index, our index function simply uses the least-significant bits of each character in the word, which essentially produces index entries that are in lexicographically ordered.

Starting with the first word in the query index, for each word w we probe the query index to obtain a list $L_q(w) = (x, y, z, ...)$ such that q_x, q_y and q_z contain a word w as its substring. Next, the database index is probed with the same word to obtain a list $L_d(w) = (p, q, r, ...)$. Then, we take the cartesian product of the entries in these two lists, to generate all the potential word hit pairs, $(x, p), (x, q), (x, r), (y, p), (y, q), (y, r), ...$ Each pair indicates a query sequence that has a potential word match with a database sequence. These sequences are then examined for actual alignments using the BLAST alignment algorithm. This process continues until we have examined all word hit pairs in Q and D.

Notice that the join of the two indices produces a complete list of database and query sequence pairs that share a common word hit. There are two interesting properties associated with the entries in this complete list. First, this list can have a number of duplicate pairs since a single database sequence and a query sequence may share a number of common words. Second, this list is not sorted by the query sequence number. A naive way of producing actual alignment would be to simply expand each sequence pair and output the result as alignments are produced. However, because of the first property of this list, a single database sequence may have to be fetched multiple times for generating alignments for the same query, which is inefficient. Second, the output that is produced will have alignments that are not grouped by the query sequence numbers. Consequently, the resulting output will have to be sorted on the query sequence number before presenting the results to the user. This sort operation can be expensive, especially with a large batch workload.

To address these issues, we employ an efficient bitmap approach. We employ a two-dimensional binary bitmap data structure, which has $|Q| \times |D|$ entries, where $|Q|$ and $|D|$ represent the number of query and database sequences each. All bit entries are initially set to FALSE. Then, whenever a word hit pair, (i, j), is found, the corresponding entry (i, j) in the bitmap is set to TRUE. At the end of the index join, the bitmap entries that are set to TRUE represent database and query sequence pairs that have at least one common word. By sequentially scanning the rows of this bitmap, for each query sequence we can generate a sorted list of database sequence IDs that should be extended in the alignment phase. The results that we produce are naturally grouped by the query sequences (a local sort is still needed to order the results for each query by the scores), and a database entry is fetched at most once for each query sequence.

Sliding-window Filtering

Since the cost of constructing a q-gram index for a large database can be expensive, it is desirable to consider methods that allow reusing an existing index whenever possible. However, since the word length for a search is a user-defined parameter, reusing an index with a word length other than the specified word length parameter is challenging. In this section, we discuss the sliding-window technique which allows miBLAST to reuse an index when the query word length is greater than the index word length.

To explain the sliding-window method, we will use the following notations: let $s[1..k]$ denote a word of length k in a sequence s, and let m and l denote the lengths of the index and query words respectively.

The sliding-window method is based on the observation that if a query word $s[1..l]$ exists in a database sequence d_i, then all of its substrings of length m, namely $s[1..m], s[2..m+1], s[3..m+2], ..., s[l-m+1..l]$, must also exist in the index of the sequence d_i. Consequently, the index entry for each of these substrings must contain the sequence ID i. This property provides a necessary condition for the word $s[1..l]$ to be found in a sequence d_i. Note that this condition is not a sufficient condition, which implies that this technique may produce *false positives.* However, the false positives can be easily eliminated by actually checking for the precise word hits when the database sequence is retrieved. We note that a similar technique is also used in A/G BLAST[55] to find word matches of length l between a query and a database using a lookup table for words of length m in a query sequence.

The algorithm for the sliding-window method is shown in Algorithm 3.

2.2.4 Implementation

The miBLAST implementation consists of three main components: q-gram index construction, filtering, and alignment generation components. The index construction component takes a formatted database as its input, which is generated by the NCBI formatdb utility, and then builds a q-gram index on the database. The filtering component takes as inputs the database index and a batch of query sequences and computes a set of database sequences containing word hits for the query sequences. Finally, the alignment generation component computes the actual sequence alignments. The alignment generation component uses the standard NCBI BLAST alignment generation component, which is modified so that it only needs to examine a subset of database sequences produced by the filtering component.

miBLAST is written in C using the NCBI toolkit. The current miBLAST implementation uses NCBI BLAST v.2.2.8, and supports querying on nucleotide data sets. These modifications have added less than one hundred lines of source code, and these modifications are clearly marked in our public release.

2.3 Experiments

In this section, we present results of an empirical evaluation of miBLAST. The database that we used for our empirical evaluation is the NCBI Human UniGene build #177. This database contains 5,064,621 sequences and roughly 3.19 gigabases.

For the empirical evaluation, we used a number of query workloads to test the impact of the following parameters: the batch workload size, word size parameter, and the query sequence length. The first two query sets are drawn from the Human Genome U133A probe sets containing oligonucleotide sequences from Affymetrix and the RefSet Oligos for the human genome from Illumina Inc. The Affymetrix probe set contain 247,965 sequences with an average length of 25. The Illumina probe set contains 22,740 sequences with an average length of 70. These query sets are used to see if the probe labels given by the company are consistent with current Unigene clusters, since the clusters change over time. In addition, we also extracted query sequences of various lengths (from 16 bp to 512 bp) from the EST human database. We used these EST query sets to measure the impact of query lengths on the performance of various algorithms.

In evaluating the miBLAST performance, we considered comparing miBLAST to other tools such as MegaBLAST [89], BLAST++ [86], NCBI-BLAST, and MP-BLAST [56]. However, as described below a number of these methods are not directly comparable with miBLAST.

We did not compare miBLAST with MegaBLAST, as it is known that the sensitivity of MegaBLAST can be less than the sensitivity of standard BLAST.

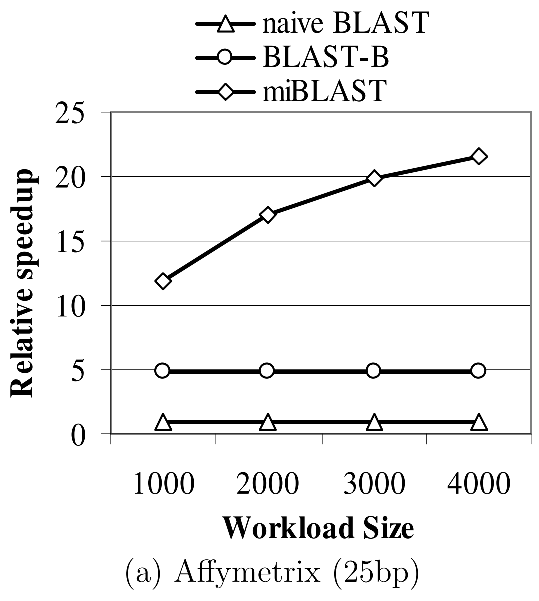
For BLAST++, the current version can not handle the UniGene database build #177 due to its large size, so we used the first half of the UniGene database to compare miBLAST with BLAST++.

In this paper we extensively compare miBLAST with NCBI BLAST v.2.2.8. For batch workloads, NCBI BLAST can be used in two ways. The first approach, which we call naive BLAST, iteratively runs BLAST for each query in the workload. The second approach uses the relatively new “-B” option in BLAST. This approach, which we call

BLAST-B, essentially implements the multiplexing method used in MPBLAST [56]. In this approach, a specified number of queries in the batch are multiplexed (i.e. concatenated) to produce a single large query string. Then the traditional BLAST method is invoked on this concatenated query string. While BLAST-B reports all the alignments that are found with naive BLAST, it produces slightly different output than naive BLAST. BLAST-B does not produce summary statistics for each query, but only produces a single summary statistics for the entire concatenated query. In contrast, miBLAST produces the same output as naive BLAST.

To run BLAST-B, the user must specify the number of queries to be concatenated. In the current version of BLAST, the upper limit on the number of concatenated queries is 255. However, although a user can specify a batch size from anywhere between 2 and 255, we have noticed that there is often an optimal batch size. In general performance initially improves as the degree of concatenation is increased, but starts dropping gradually after a certain point. This *optimal* point can change depending on query sizes, data sets, and BLAST search parameters. For BLAST-B, we ran several experiments to manually find the optimal batch size for each workload, and used the optimal batch size for each BLAST-B run.

Default BLAST parameters were used for running the three different methods. All experiments were run on a machine with a 2.2GHz AMD Opteron processor and 4GB RAM running the Linux 2.6.9 kernel. All measurements are based on the performance with a *cold cache*, which essentially means that before each experimental run the system cache has no pre-cached data. All times reported in the following experiments are actual wall-clock time taken to run the queries.



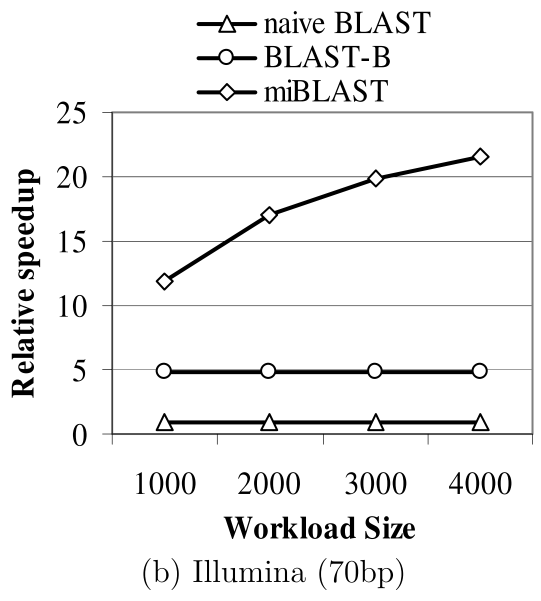


Figure 2.1: The relative speedup of each method compared to naive BLAST

Effect of Batch Workload Size

To examine the effect of batch workload size on miBLAST, naive BLAST, and BLAST-B, we ran experiments by increasing the batch workload size (the number of queries in the batch) from 1000 to 4000. Figures 2.1 (a) and 2.1 (b) show the relative speedup to naive BLAST for the Affymetrix (25bp) and Illumina (70bp) workloads respectively.

Table 2.2: Effect of Batch Workload Size

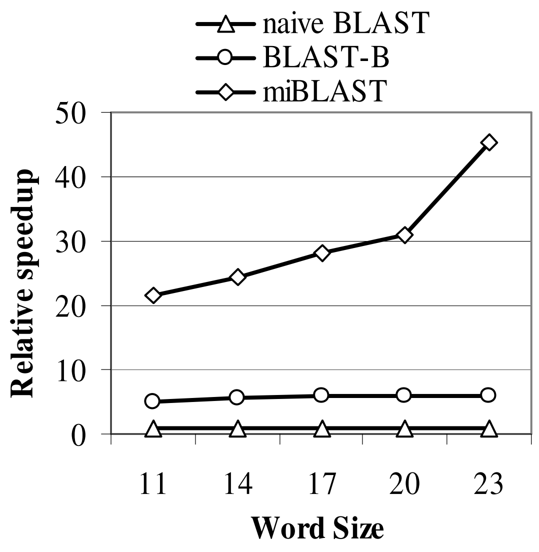
Workload size	Filtering Cost per Query (25 bp)	Alignment Cost per Query (25 bp)	Filtering Cost per Query (70 bp)	Alignment Cost per Query (70 bp)
1000	0.44 (54%)	0.37 (46%)	0.46 (31%)	1.02 (69%)
2000	0.23 (42%)	0.33 (58%)	0.25 (21%)	0.96 (79%)
3000	0.17 (36%)	0.31 (64%)	0.18 (16%)	0.94 (84%)
4000	0.14 (31%)	0.30 (69%)	0.17 (15%)	0.94 (85%)

The detailed execution time of miBLAST for the experimental results shown in Figure 2.1. All times reported here are in seconds

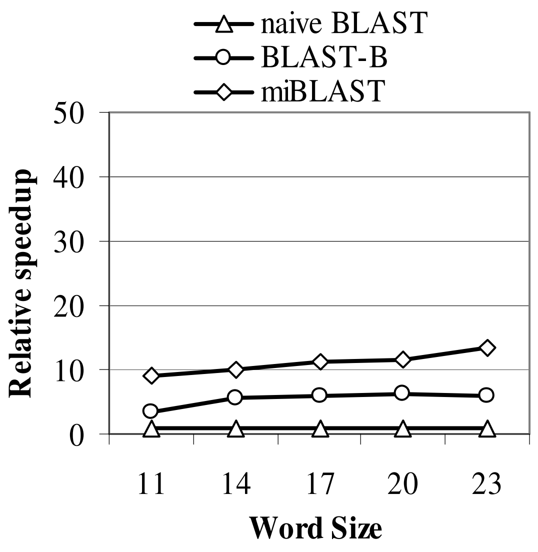
As can be seen in Figure 2.1, miBLAST is significantly faster than the other two methods. The performance of miBLAST improves as the batch size increases, and for a batch size of 4000 queries, miBLAST is 21.6 and 9.9 times faster than naive BLAST for the 25 bp and 70 bp queries respectively, and 4.5 and 2.7 times faster than BLAST-B for the 25 bp and 70 bp queries respectively.

To understand why the performance of miBLAST improves as the batch size increases (as seen in Figure 2.1), consider Table 2.2, which shows the breakdown of the filtering cost (the index join component), and the cost of calling the BLAST alignment method for different workloads. As can be seen from this table, both the filtering and alignment costs per query decrease as the workload size increases. For the filtering cost, with a larger workload size the cost of the index join is amortized over a larger number of queries, resulting a reduction in the per query filtering cost. For the alignment cost, the reduction in per query cost come from the benefits of using the operating system cache. For the initial sequences in the batch, fetching a database sequence often results in an actual disk I/O. However, as the batch size increases the chances of a single database sequence matching more than one query sequence increases. Repeated accesses to the database sequence are likely to find the database sequence in the operating system cache, and does not have to incur an expensive disk I/O. Consequently, as the batch size increases, the alignment costs per query also reduce.

It is also notable that the database caching effect further improves the relative performance of miBLAST. For instance, when we ran five workloads of 4000 queries consecutively, the relative speedup in processing the last workload increases up to



(a) Affymetrix (25bp)



(b) Illumina (70bp)

Figure 2.2: The effect of the BLAST word size parameter on the query performance for each method

25.4 times, while the relative speedup of the first batch workload is 21.6 times. The reason for this improvement is that the runs 2-4 benefit from seeing data in the cache that has been retrieved by the processing of previous runs. miBLAST benefits more from this caching as it is more disk I/O intensive, as has a much larger data structure (the index). In fact, we can expect that if there was enough space to hold the entire miBLAST index in memory, its relative performance would be even greater.

2.3.1 Effect of the BLAST Word Size Parameter

In this section, we examine the effect of the query word size parameter, which is a commonly tuned BLAST parameter.

For this experiment, we use the same data set and query workload as used in the previous experiment, but increase the word size gradually from 11 to 23. In each case miBLAST used an index that had a word size of 11. The results of this experiment

Table 2.3: Word size and filtration ratio

Word size	Average Filtration Ratio	Alignment Cost per Query (in seconds)
11	0.54964 %	0.30
14	0.03924 %	0.13
17	0.02231 %	0.11
20	0.02115 %	0.09
23	0.02009 %	0.08

The effect of the BLAST word size parameter on the filtration ratio and the alignment cost in miBLAST. The query set used for collecting this data is 4000 queries from the Affymetrix data set.

are shown in Figure 2.2. As seen from this figure, the performance of miBLAST improves significantly as the word size increases. For a word size of 23, miBLAST is 45.2(25 bp) and 13.6 (70 bp) times faster than naive BLAST, and 7.5 (25 bp) and 2.2 (70 bp) times faster than BLAST-B.

To understand why a larger word sizes benefits miBLAST, we measured a metric called *filtration ratio*. Filtration ratio is defined as the ratio of the number of sequences that are identified as potential results using the word hit over the total number of sequences in the database [15]. The filtration ratio measures the proportion of the database that must to be examined to generate actual alignments. With miBLAST a lower filtration ratio leads to better performance as a smaller fraction of the database is searched during the alignment phase. As the word size increases, the probability of finding a word hit decreases exponentially, leading to a exponential decrease in the filtration ratio. This filtering behavior with respective to word size and its effect on the performance of miBLAST is shown in Table 2.3.

As can be seen in Figure 2.2, with miBLAST a larger word size does lead to a

Table 2.4: Execution time per query for various word sizes

Word Size	naive BLAST	BLAST-B	miBLAST
11	9.50	1.95	0.44
14	9.26	1.61	0.38
17	9.46	1.58	0.37
20	9.45	1.58	0.30
23	9.44	1.56	0.21

The average execution time per query for the results shown in Figure 2.2 (a), for a batch size of 4000. All times reported here are in seconds.

reduction in the the number of retrieved sequences. However both naive BLAST and BLAST-B scan the entire database (during the word hit generation phase), so the lower filtration ratio does not reduce the number of database sequences that are retrieved. There is a reduction in the number of alignments that are computed, but both naive BLAST and BLAST-B spend most of their execution time in the word hit generation phase. Consequently, the overall reduction in execution time with increasing word size is very small for these two methods.

An astute reader may have noted that in Table 2.4 increasing the word size does not have a significant impact on the performance of naive BLAST and BLAST-B. While this effect may seem contrary to the intuition of improved performance for larger word sizes, it turns out that in the case of *nucleotide* sequence searches, increasing the word size does not have a significant impact on the performance of both naive BLAST and BLAST-B due to the way the database sequence representation database is packed into bytes and interactions of this packing with the processor word length. Increasing the BLAST word size parameter may have negligible effect on performances as the processor may still be doing equivalent work, because it is fetching and process-

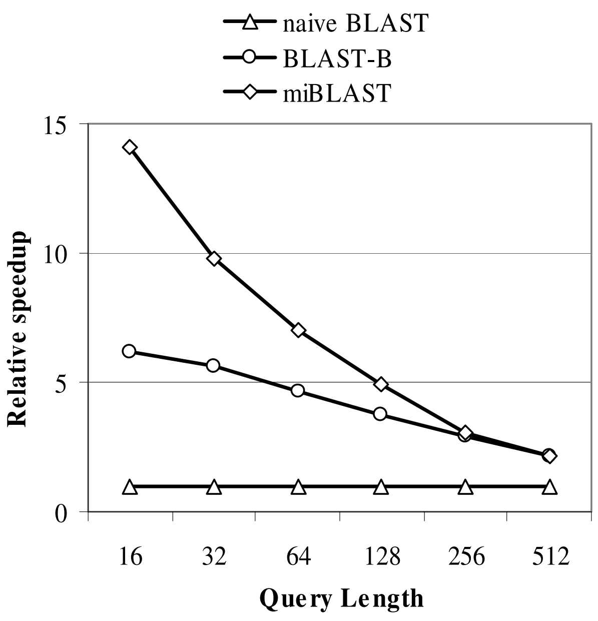


Figure 2.3: The relative speedup to naive BLAST for various query lengths

ing data in block sizes that are set by the underlying computer architecture. In fact, in some cases increasing the word size may actually result in a small decrease in performance (for example when the larger BLAST word parameter requires the processor to operate on a larger number of memory blocks). This effect of increasing word size has also been reported for WU-BLAST (see http://blast.wustl.edu/blast/TO-FLY.html).

2.3.2 Effect of Query Length

To measure the effect of query length on the performance of miBLAST, we ran the following experiments. We generated a number of query sets from the EST human database. Each query set contained 1000 queries of a fixed length, that was randomly picked from the EST database. We generated query sets with query lengths ranging from 16 to 512bp and each query set is run against UniGene database. The results of this experiment (see Figure 2.3) show that the performance speedup of miBLAST decreases as query size increases. The reason for this behavior is that miBLAST's performance is highly dependent on filtration ratio, as it primarily speeds up the filtering component of BLAST searches. In general, a lower filtration ratio leads to better relative performance for miBLAST. As the query length increases, it is likely that the query will have more word hits with sequences in the database, increasing its filtration ratio, and resulting in a relative reduced performance for miBLAST.

We also note that the comparisons with BLAST-B in Figure 2.3 are based on the most optimal batch size for BLAST-B, which we picked by manually trying various batch sizes for each query set. For different workloads, the optimal batch size changes and to get the best performance using BLAST-B the user has to manually determine the optimal batch size. For instance, the optimal batch sizes are approximately 100, 50, 50, 25, 25, and 12 for queries of lengths 16, 32, 64, 128, 256, and 512 respectively. In contrast, with miBLAST there is no such manual tuning requirement. The optimal batch size can have a significant impact on performance for BLAST-B; for instance, with 256bp queries, using a batch size 200 instead of 25 increases the total execution time by 50%.

2.3.3 Performance Comparison with BLAST++

In this last experiment, we compare miBLAST with BLAST++. As noted earlier, the current version of BLAST++ can not handle the size of the current Human Unigene data set. Hence, for this experiment we used only half of the Human Unigene

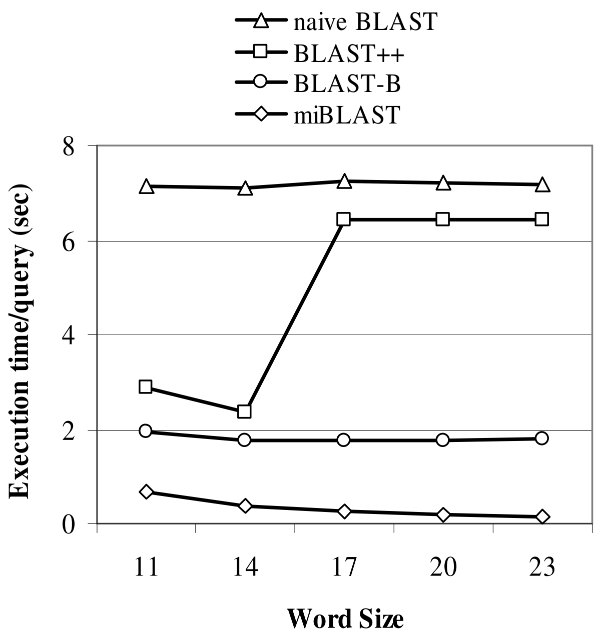


Figure 2.4: The execution time of each method for various word sizes

data set (for all the the methods). The query set that we use in this experiment is the Affymetrix oligonucleotide sequences (25bp). The BLAST++ configuration is similar to BLAST-B, and there is an optimal batch size, which in this case is about 200 queries in a batch. The results of this experiment are shown in Figure 2.4. As can be seen in this Figure, BLAST++ only outperforms the naive BLAST and is worse than miBLAST and BLAST-B in all cases. miBLAST is 5 and 48 times faster than BLAST++ at a word size of 11 and 23 respectively. Also, note that as the word size increases, the performance of BLAST++ generally degrades. The performance improvement for BLAST++ comes from sharing database sequence information for common words in the queries. However, when the word size is large, the number of such common words is reduced, and BLAST++ ends up accessing larger number of database sequences for each query sequence.

2.4 Discussion

2.4.1 Index Storage and Construction Costs

The index used in miBLAST requires $(8 \times S^m + 4 \times N)$ bytes, where S is the size of the alphabet for the symbols (typically 4 for nucleotide sequences), m is the index word length, and N is the total number of symbols in the database. $8 \times S^m$ bytes are used for index header, and $4 \times N$ bytes are used for saving sequence ID information. However, since we don't save a duplicate sequence ID when the same word occurs more than once in the same sequence, the actual index size is significantly smaller than indicated by the above formula. For the human UniGene database containing 3.19 gigabases, using a word length 11, the index for the database is 11.94 GB. Only 32MB of this space is used for the index header, and the remaining portion of this index space is used for storing the sequence ID information.

Constructing a q-gram index can be expensive for large databases. However, the index construction cost is a one time cost and this cost is amortized over all the batch workloads that use this index. For example, the index on the Human UniGene takes 38 minutes on our test machine. Assuming that we process the Affy query set consisting of 247,965 query sequences, the index construction cost per query is 0.0092 seconds, and this cost can be reduced further when we have a larger batch workload or when the index is used to process multiple batch workloads.

2.4.2 Biological Applications of miBLAST

In this section, we discuss the biological applications that can benefit from miBLAST. The characteristics of miBLAST make it immediately applicable when evaluating a large number of oligonucleotide probes, cDNA sequences, or ESTs against databases of ESTs. Next we elaborate on some applications with these characteristics.

One important application is the validation of a probe set, such as the Affymetrix probe set, against the UniGene database. The Affymetrix probe sets are searched against the most recent UniGene, and the search is often conducted periodically triggered by updates to the UniGene dataset. In this search, if one is only looking for labels, then a regular expression search is sufficient. However, the mismatches found in BLAST are often also important. For example, a step in evaluating the Affymetrix probes is an hybridization energy calculation for each probe that has sequence similarity with an EST. Mismatches may not bind with the same efficiency as a perfect match, but knowing the number of mismatch probes and the hybridization energy helps determine the specificity of the individual probes[59]. This task requires a sensitive search tool. BLAST, which uses overlapping words in its search technique, when run with a small word size parameter results in a sensitive search that is suitable for this application. Since this task needs periodic evaluation of a large number of probes against a large EST database, miBLAST is a good alternative to BLAST.

A similar application is when designing new probe sets for DNA microarrays. miBLAST can be used to search new probe sets against the EST library of a species as a first pass for sensitivity. The search result of mismatch and perfect matches can be used to calculate hybridization energy for new microarray design. This task has been a major step in programs developed for probe design[75, 88] and miBLAST can help speed up the basic computation task.

Another biological application is when using one animal model microarray against a similar species. While the creation of a new chip set for every species is technically feasible, the cost involved and skills necessary are not widespread. Until that time, using a similar species chip set will be an inexpensive solution. In this case, searching the chip probe set against the collection of ESTs or cDNA of the related species are often needed to validate the probes in the new species [60].

While the focus of the applications in the discussion above has been on microarray studies, miBLAST can also be used in other applications. As more individuals create siRNAs, to silence genes within cells, short nucleotide sequence searches against EST databases will increase to look for cross hybrid activities with other ESTs to narrow the specificity of the siRNAs. miBLAST can decrease the computation demands of this task.

We note that miBLAST improves the efficiency of sequence searches for short nucleotides against ESTs. If one is concerned with EST-to-genome or genome-to-genome alignment, other existing methods [29, 51, 81] are likely to be more practical for such tasks.

Finally we note that our miBLAST implementation, like WU-BLAST[34], also allows the user to specify a scoring matrix, which can be used to model complex scoring models, such as discriminating between the scoring of transitions and transversions.

2.5 Conclusion

In this paper, we have presented miBLAST, a fast BLAST-like search algorithm for efficiently evaluating batch workloads which consist of a large number of nucleotide query sequences that must be matched against a nucleotide sequence database. Current methods for evaluating such workloads essentially employ a “nested-loops” paradigm in which each query sequence is individually evaluated using the BLAST search tool. This existing approach can be very expensive, especially for large batch sizes. Using a combination of q-gram indexes and an index join algorithm, miBLAST can dramatically speed up the evaluations of such workloads. miBLAST is particularly effective for workloads which consist of short queries, such as oligonucleotide probe sets.

The miBLAST search tool is implemented using the NCBI toolkit, and employs the same statistical model and output format that is familiar to BLAST users. Consequently, we expect that existing BLAST users can make a seamless transition to miBLAST. The source code and executable for miBLAST are freely available.

CHAPTER III

ProbeMatch: an efficient and effective tool for mapping oligonucleotides

3.1 Introduction

New high throughput DNA sequence technologies play an important role in modern life science research. These high throughput methods, such as the Solexa and 454 Life Sciences technologies, produce a large volume of sequence data, which can be used for a variety of tasks including genome re-sequencing and genome-wide polymorphism discovery [39]. These methods produce a large set (thousands or millions) of short sequences that often must be mapped to a genome, allowing for only a few errors. Traditional sequence alignment tools such as BLAST [9] provide the necessary functionality, but they are computationally prohibitively expensive for this task.

To address this computational problem, several programs have been developed. ELAND [6], which is a part of the data analysis pipeline for the Illumina-Solexa analyzer, is designed to search DNA databases for a large number of short sequences.

To speedup the data processing, ELAND performs only ungapped alignments allowing up to 2 mismatches. MAQ [38] is another alignment program designed for Illumina-Solexa analyzer, which also performs only ungapped alignments allowing up to 3 mismatches. In addition, using sequence quality information, MAQ measures the error probability of alignments. SOAP [62] is another tool, which allows for ungapped alignments, and alignments with one continuous gap (1 opening gap and up to 2 extended gaps) and no mismatch. For example, SOAP will find an alignment with 1 continuous gap of size 2, but won't find an alignment with 1 gap and 1 mismatch.

Compared to traditional programs such as BLAST, these newer programs are often faster by an order of magnitude or more, but these programs usually map only 60-80% of the query sequences to genomes, leaving a significant fraction of the sequence workload for further processing using computationally expensive but sensitive alignment methods [39]. Unfortunately, as predicted by Amdahl's law [10], the overall gain using these faster methods is limited and does not adequately address the pressing end-to-end computational problem of rapidly and accurately mapping a large number of oligonucleotide sequences against entire genomes. In this work, we address this challenge and present an efficient and highly sensitive sequence alignment technique for a large set of short sequences. Unlike existing methods, our program allows for a richer match model and finds gapped and ungapped alignments with up to 3 errors of any error combinations (mismatch, insertion, and deletion). The ability to identify both gapped and ungapped alignments has the added advantage of being able to detect multiple classes of mutations: single nucleotide variations (SNVs) and insertions or deletions (indels). The genetic variation provided by each class of mutations can occur in coding or regulatory regions thereby altering the function of important proteins which have a major impact on human health and susceptibility to disease. Similar to other programs, ProbeMatch only reports the top scoring alignments for each input query sequence. However, since ProbeMatch performs gapped alignments, top scoring alignments can be different depending on the match, mismatch, and gap scores. In our program, by default, an alignment with a single mismatch has a precedence over an alignment with a single gap. Except for this constraint, other constraints can be adjusted. For example, a user can give a higher precedence to an alignment with 1 gap over an alignment with 3 mismatches by setting appropriate scores for match, mismatch, gap opening, and gap extension. However, these scores must be set such that the score of an alignment with 1 gap is higher than the score of an alignment with 3 mismatches. For ease of use, our program provides a list of options for selecting precedence constraints over alignments and it automatically select the appropriate scores for a selected option. ProbeMatch has been used to align 169,095 Illumina-Solexa reads against a human genome, which were not mapped by ELAND, and found alignments for 26,416 reads of these 169,095 sequences in 2 hours. It is interesting to note that a large number of reads are not mapped. A likely reason, based on our experience and other labs that are working with the relatively new Solexa technology, is that these may be due to adaptor sequences, poor quality sequence which can no longer map to anything, contaminants, etc. It is also likely that more indels/mismatches could be responsible for these if they have large gain/loss of nucleotide. We plan on exploring these issues further in the future as we gain more experience with this technology.

3.2 Methods

ProbeMatch takes as input a query sequence set and a database of sequences. Since the database is often large and cannot fit in memory, ProbeMatch divides the database into small segments. ProbeMatch loads each database segment and builds a q-gram index. For each query sequence, ProbeMatch searches against the q-gram index to find potential hits and extends hits to find longer alignments. This process is repeated until all database segments are processed.

To speedup sequence alignment, we use a filtering technique based on the following lemma: If two sequences, Q and T, match within k errors and j non-overlapping fragments are taken from Q, then the matching sequence T contains at least one of the fragments with at most $\lfloor k/j \rfloor$ errors (Error includes not only mismatches but also insertions and deletions) [11]. Based on this lemma, to find matching sequences with at most 3 errors ($k = 3$), first, the query sequence is split into into two fragments ($j = 2$). Next, matching target sequence that can be aligned to one of the fragments within 1 error ($\lfloor k/j \rfloor = 1$) are found. Finally, the matched hits are extended to check if the entire query sequence and the target sequence can be aligned within 3 errors. The ProbeMatch index is a *gapped* q-gram index. A gapped q-gram is a non-contiguous alphabet sequence with q alphabet matches and don't cares at specified positions [78]. For example, one gapped 3-gram is a non-contiguous sequence of length 4, which has 3 exact alphabet matches and 1 don't care at a specified position. There exist several patterns of a gapped q-gram depending on the don't care position. For instance, one gapped 3-gram has two different patterns, ##_# and #_##, where #

and _ corresponding to an alphabet match and a don't care respectively.

The gapped q-gram has an advantage over an ungapped q-gram as a carefully selected gapped q-gram pattern provides a more efficient filtering than an ungapped q-gram. For example, consider finding all candidate sequences that match a query of length 18-nt, with upto 1 mismatch. Such candidate sequences can be identified using an ungapped q-gram of length 9 (#########). However, using the following one gapped q-gram, ######_##### (or #####_######), we can also identify all candidate sequences. Using this gapped q-gram requires two more alphabet matches than an ungapped q-gram, which leads to a more selective (hence efficient) filter than an ungapped q-gram. Choosing an optimal gapped q-gram pattern is important, and the optimal pattern is determined by the query sequence size and the number of mismatches allowed. For a list of sequence sizes and number of mismatch pairs, we pre-compute optimal patterns by exhaustive search and use this information at runtime. In the above example, using the gapped q-gram pattern, ######_#####, we can detect candidate sequences having a mismatch, but we cannot detect candidate sequences having an indel. To handle indels, we need to use multiple q-gram patterns. In addition to the basic pattern, ######_#####, we use two other patterns, ######__##### and ###########. These patterns are generated from the basic pattern by increasing and decreasing the length of the gap by 1 [77]. For a query sequence, we generate q-grams using all three patterns. For a database sequence, we generate q-grams only using the basic pattern and build a q-gram index. Even if there is an indel, any matching query and target sequences within 1 error should share at least one q-gram. Using q-grams from the query sequence, we look up the q-gram index to find candidate database sequences and extend them to check whether the query and target sequences can be actually aligned within 1 error. The techniques described above can be applied to find matching sequences with any k errors. In our implementation, to find matching sequences with 3 errors, we split a query sequence into 2 fragments. To find matching sequences with k errors, we need to split the query sequence into $k-1$ fragments. For each query sequence fragment, we then need to identify matching database sequences with at most 1 error (including indels), using the gapped q-grams technique described above. Then, we can extend this query fragement match for an entire query to check if the entire query and matching database sequences can be aligned within k errors. With an increasing k, each query fragment size becomes smaller, requiring a smaller gapped q-gram to find matching sequences for each fragment. Such smaller q-gram lead to less efficient q-gram based, which in turn increases the overall execution time. Our current code is optimized to handled only two query fragments.

3.3 Experiments

We evaluated the performance of ProbeMatch using transcriptome data from a prostate cell line (RWPE), generated by the Illumina Genome Analyzer. As a query dataset, we selected 169,095 transcriptome short queries (36nt), that were not mapped to the human genome by ELAND. This query dataset is run against the human genome using various alignment programs on a machine with a 2.2GHz AMD Opteron processor and 4GB RAM running the Linux Fedora 2.6.9. Results are shown in

Table 3.1: Comparison of execution times and sensitivity

Program	Time (hh:mm:ss)	Reads aligned
ELAND	32:18	0
BLAST (-F F -W 9)	34:45:19	21,684
BLAST (-F F -W 11)	22:57:53	15,230
SOAP	19:28	16,024
SOAP gapped	37:24	18,681
ProbeMatch	2:16:06	26,467

We aligned 169,095 sequences against the entire human genome using multiple programs. For BLAST, we ran two experiments, one using default BLAST parameters which uses a word size of 11, and another using a smaller word of size 9. In the BLAST experiments, the DUST filter option is disabled since other programs do not mask low complexity sequence regions (The DUST filter masks low complexity regions and is enabled in BLAST by default. While masking helps increase speed and reduce false positives, it takes away some regions of the genome that could be important for this dataset). For SOAP, we also ran two experiments that perform ungapped and gapped alignments respectively. We report the number of unique queries that having alignments with no more than 3 errors as the number of aligned queries.

Table 3.1 and Figure 3.1.

Table 3.1 compares sensitivity and speed of various programs. It shows the number of queries mapped to the human genome and total execution times of each program. As shown in Table 3.1, ProbeMatch is 10-15 times faster than BLAST, while aligning more queries. Compared to SOAP, ProbeMatch is 3.6-7.0 times slower, but it aligns 30-40% more queries: ProbeMatch and SOAP align 26,467 and 18,681 queries respectively. Both programs found ungapped alignments with 3 mismatches for 16,024 queries. They also found gapped alignments with 1 continuous gap for 2,657 queries. In addition to these alignments, ProbeMatch found alignments for 7,786 queries, which cannot be found by SOAP. These queries are the ones with more than 1 gap or the ones that have both gaps and mismatches.

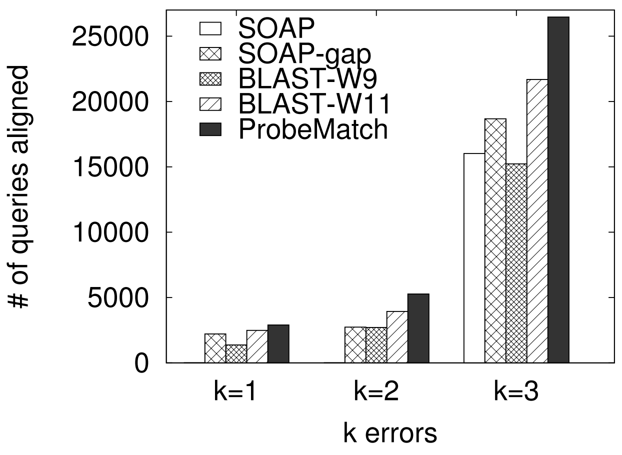


Figure 3.1: Aligned queries for different k-error values

Figure 3.1 shows detailed information about the number of queries mapped to the human genome when varying the permitted error (i.e. varying the parameter k). Although it is not shown in this figure, the result from ProbeMatch includes all the queries aligned with SOAP and BLAST for $k = 3$. With BLAST output, in some cases, we cannot determine whether a query sequence can be aligned to a target sequence within 3 errors. BLAST performs local sequence alignments, which may not align an entire query to a target sequence. Due to this, BLAST output shows alignments containing only aligned regions of query and target sequences. Without aligning an entire query sequence to a target sequence, we cannot figure out the number of errors between two sequences.

To illustrate this point, consider the following two sequences having 3 errors between them (There are 3 mismatches in 33-35th base pairs).

Q=GAATGTTTCATTAATCAAGAACGAAAGTCGGAGGTT

D=GAATGTTTCATTAATCAAGAACGAAAGTCGGACCCT

For these two sequences, BLAST finds and shows the following alignment.

```
Q:GAATGTTTCATTAATCAAGAACGAAAGTCGGA
  ||||||||||||||||||||||||||||||||
D:GAATGTTTCATTAATCAAGAACGAAAGTCGGA
```

Note that the above alignment aligns the first 32 base pairs of the query sequence Q to the target sequence D, and does not align GGTT in Q (the last 4 base pairs) to D. This is because aligning GGTT to D will decrease an overall similarity score. From this BLAST output, we speculate that there are at least 32 exact matches and 1-4 errors between Q and D. However, we don't know exactly how many errors are. For an entire query Q to be aligned to the target sequence D within 3 errors, there should be at least one more exact match in the last 4 base pairs (and at least 33 exact matches between Q and D). To check this, we need to extend the alignment for the unaligned query sequence region (the last 4 base pairs), and count how many errors are in the region. Essentially, we have to generate the following alignment, which aligns the entire query sequence to the target sequence.

```
Q:GAATGTTTCATTAATCAAGAACGAAAGTCGGAGGTT
  ||||||||||||||||||||||||||||||||   |
D:GAATGTTTCATTAATCAAGAACGAAAGTCGGACCCT
```

With BLAST output, post-processing such as extending unaligned sequence regions is required. ProbeMatch does it by default.

3.4 Conclusion

In this chapter, we described another sequence alignment algorithm, called ProbeMatch, for matching a large number of oligonucleotides against a genome database. To speedup sequence alignment, ProbeMatch uses gapped q-grams and q-grams of various patterns. This approach results in fewer initial sequences to examine with no loss in sensitivity under a k mismatch model. Unlike existing tools which only performs ungapped alignments, ProbeMatch generates both ungapped and gapped alignments, and shows its efficiency and effectiveness by producing more alignments in a faster time.

CHAPTER IV

ProCC: an efficient index-based method for protein structure classification

4.1 Introduction

Classification of protein domains based on their tertiary structure provides a valuable resource that can be used to understand protein function and evolutionary relationships [69]. As a result, several classification databases [47, 69, 73] have been developed, of which SCOP [69] and CATH [73] are the most widely used databases. Both databases are hierarchically organized and use protein domains as a basic unit of classification.

While SCOP and CATH provide a valuable resource for biologists, these databases are updated only intermittently – for example, over the past three years, SCOP has been updated roughly every six months, and CATH has been updated annually. Updates to these databases require varying degrees of semi-automated methods and manual interpretation. As a result, newly deposited protein structures only show up in the classification hierarchy in the next release cycle of these databases. At the same time, the number of newly determined protein structures has been growing rapidly. For instance, during the past year, more than 5000 structures were deposited in PDB. Also, the number of structures in PDB today is roughly double the number of structures in the year 2000[2]. This rapid increase in the number of new protein structures makes the need for *automated* classification tools even more important.

We recognize that the manual and semi-automated methods used in SCOP and CATH produce classification hierarchies that are of high quality, and automated methods are unlikely to incorporate the nuanced judgment that an experienced biologist brings to the classification task. Nevertheless, automated methods, if they are highly accurate, can provide a valuable complementary approach in producing classification hierarchies. With the rapid increase in the number of protein structures, automated methods can (and currently do) play an important role as a pre-processing step for producing manually-tuned classification hierarchies.

Recognizing this need, several automatic domain classification methods [17, 20, 33, 35, 74] have recently been developed. Superfamily [35] is purely based on sequence comparison criteria. It is efficient, but often fails in correctly classifying remote homologs of structurally similar proteins. Methods such as F2CS [33] and SGM [74] are based purely on structure comparison. They are computationally very efficient and accurate for classifying at the fold level, but not necessarily at the superfamily and family levels. Recent methods [17, 20] combine sequence and structure information for classification, and make a classification decision based on a consensus of several sequence and structure comparisons. In general, these methods are more accurate than previous methods, though they are computationally more expensive.

An important issue in automatic protein classification is the ability of a tool to detect new classes (i.e. detecting novel folds). Detecting such new classes is important as novel domain structures are constantly found in newly determined protein structures, and the information about new classes can be effectively used to better understand the new structures and can also be used to assist humans in organizing the new structures into the next version of the classification hierarchy. While many of these existing protein classification methods are very good at classifying new domains into *existing* classes, the effectiveness of these methods in detecting *new* classes is modest. Of the existing classification tools, both SGM [74] and SCOPmap [20] can be used to detect new classes, and as we show in this paper our method is much more effective compared to these two methods for new class detection.

In this paper, we present *proCC* – an automatic, accurate, and efficient classification framework, which consists of three components. Given an unclassified query domain, a structure comparison component employs an index-based method to quickly find domains with similar structures. Then, based on these results, a classification component assigns the query to an existing class label, or marks the query as *unclassified* to indicate that the query domain is potentially a new fold. Finally, a clustering component takes all domains marked as unclassified and runs a clustering method to detect potentially novel folds. Collectively, these components provide a unified and automated protein domain classification tool.

To demonstrate the capabilities of our methods, we have tested our method to predict the classification for new domains in SCOP 1.69 based on prior knowledge of the previous version of SCOP (version 1.67). Our experimental results show that the precision of our method is 86.0%, 87.7%, and 90.5% at the family, superfamily, and fold levels. We also compare our method with SGM and SCOPmap, and show that our methods are about 15-19% more accurate than SGM and comparable to SCOPmap. However, SCOPmap only classifies at the superfamily and the fold levels, whereas our tool also provides a classification at the the family level. More significantly for new fold detection, the predications made by proCC is 20% better than SCOPmap. Our experimental evaluation also shows that our method produces clusters which closely correspond to the new families in SCOP 1.69.

4.2 Algorithms and Methods

Our proCC method consists of a pipeline of the following three modules: structure comparison, structure classification, and clustering. Given a new query protein domain, the structure comparison module finds the top k structures that are similar to the query. Then, based on these results, the classification of the query domain is performed using the class label information from the k nearest structural neighbors, and a support vector machine (SVM) [23]. This second step may label a query domain as *unclassified* if the classification module cannot assign a class label with enough confidence. Finally, for the domains that are labeled as unclassified by the previous step, a clustering module identifies cluster boundaries as a way of suggesting groups of domains that are potentially in novel folds. Each of these three steps is described in more detail below.

4.2.1 Structure comparison

The protein domain structure comparison module employs an index structure to rapidly find structures that are similar to the query. The basic unit for comparing structure similarity is a triplet of secondary structure elements (SSEs). Comparing protein structures using SSEs has been used before [47, 64, 66] as it is more efficient for computing structure similarity, compared to using the actual atomic coordinates of the C_α atoms, as is done in DALI [46] and CE [82]. In addition, since domains are classified according to the composition and the spatial arrangement of SSEs in common classification databases such as SCOP and CATH, structure comparison based on SSEs is more natural for the purpose of structure classification.

To find the top k domains that are similar to a given domain, the following steps are performed in order.

1. Each domain in the database is decomposed into a set of SSE triplets. A 10-dimensional vector is used to represent the SSE triplet, and an index is constructed over all the SSE triplets in the database. The query protein domain is also decomposed into SSE triplets.

2. For each SSE triplet in the query, matching SSE triplets are retrieved using the index. Based on the hits from this index probe, a similarity score between the matching triplets is calculated.

3. For each target domain in the database, a weighted bipartite graph is generated based on the SSE triplet matching results. For each target graph, a maximum

weighted bipartite graph matching algorithm is run to compute an overall similarity score between the query and the target. Finally, the top k scoring targets are returned as the result of the search.

Each of these three steps is described in detail in the following three subsections.

We note that our method finds all the domains in the database that have at least one or more SSE triplet matches to the query, and k is the number of such domains in the database. Therefore, the value for k varies depending on the query structure and is automatically determined by our method.

Structure representation and indexing

We model each protein domain as a set of SSEs, and represent each SSE using its associated type, length, and a direction vector. Given a SSE S_i, its type, denoted as T_i, is either an α helix or a β strand. For a concise representation, loops and turns are excluded. The length of the SSE, denoted as L_i, is the number of residues contributing to the formation of that SSE. The direction vector, denoted as V_i, is a unit vector, $V_i = (X_s - X_e)/||X_s - X_e||$, where X_s and X_e represent the two end points of the SSE. X_s and X_e are calculated using the following equations defined in [83].

For an α helix, X_s and X_e are calculated as:

$$\begin{aligned} X_s &= (0.74X_i + X_{i+1} + X_{i+2} + 0.74X_{i+3})/3.48 \\ X_e &= (0.74X_j + X_{j-1} + X_{j-2} + 0.74X_{j-3})/3.48 \end{aligned}$$

where X_i and X_j represent the beginning and ending residues of the SSE.

For a β strand, X_s and X_e are calculated as:

$$\begin{aligned} X_s &= (X_i + X_{i+1})/2 \\ X_e &= (X_j + X_{j-1})/2 \end{aligned}$$

Since we are interested in indexing SSE triplets, we use the above representation of a single SSE to develop a representation for an SSE triplet. Given three SSEs, S_i, S_j, and S_k, the triplet containing these three SSE contains the following information:

- SSE types: T_i, T_j, T_k.
- SSE lengths: L_i, L_j, L_k.
- Angles between each pair of SSEs: θ_{ij}, θ_{ik}, θ_{jk}, where θ_{ij} is the angle formed by S_i and S_j, and it is calculated as: $\cos^{-1}(V_i \cdot V_j) \bmod 180$. (Note that the mod 180 component of the equation is used to allow for similarity matching under coordinate inversion.)
- Distances between each pair of SSEs: D_{ij}, D_{ik}, D_{jk}, where D_{ij} is the average of the minimum distances between residues in S_i and S_j. To calculate D_{ij}, the smaller SSE (between S_i and S_j) is selected. Then, the minimum distances from every residue in S_i to every residue in S_j are calculated ($S_i \leq S_j$), and the average of these minimum distances is used as the SSE distance D_{ij}. Intuitively, this measure aims to concisely capture the distance between two SSEs. The index search (described below) will use these distances to remove pairs of SSEs that have very different inter-SSE distances.

The information describing an SSE triplet is encoded into a compact 10-dimensional vector, which serves as the actual representation of the SSE triplet in an index. This

10-dimensional vector is:

$$(TC, X_i, Y_i, X_j, Y_j, X_k, Y_k, D_{jk}, D_{ik}, D_{ij})$$

In this vector representation, amongst the three SSEs, the i^{th} SSE is closest to the N-terminal, and the k^{th} SSE is closest to the C-terminal. TC is a three bit value that encodes the types of the three SSEs. The next six values, X_i, Y_i, X_j, Y_j, X_k, and Y_k, represent the lengths and angles of S_i, S_j, and S_k. Each SSE, for instance S_i, is mapped to a point (X_i, Y_i) in a 2D Euclidean space where $X_i = L_i cos\theta_{jk}$ and $Y_i = L_i sin\theta_{jk}$. This transformation to a 2-D Euclidean coordinate allows us to use a conventional spatial index for efficiently locating close neighbors. The last three values, D_{jk}, D_{ik}, and D_{ij}, are the pairwise SSE distance values as defined before.

The 10-dimensional vector representation serves as the key for indexing the SSE triplets. For a given protein domain, rather than inserting an index entry for every SSE triplet, we only insert SSE triplets that have *all* inter-SSE distances less than 20 A°. This cutoff value is based on a similar cutoff that is used in DALI [46].

In our method, a SSE triplet is used as a basic search unit. While different cardinality for SSE can also be considered (for example a quadruplet or a pair instead of a triplet), the SSE cardinality directly affects the efficiency and the sensitivity of the searches. Using a SSE pair increases the sensitivity of searches, but degrades the search efficiency, especially when searching a large database of domains. Using a SSE quadruplet is more efficient, but may be too conservative in detecting distantly related structures, such as domains in the superfamily or fold levels. We use a SSE triplet to strike a balance between sensitivity and search efficiency. We also note that the use of SSE triplet has been made for similar reasons in previous works [18].

SSE triplet matching and index probing

To match a query against a database of proteins, we first decompose the query into all SSE triplets with inter-SSE distances less than 20 A°. We then probe the index with each query triplet and retrieve *target* triplets in the database that are "similar" to the query triplet. Similarity between a query triplet and a target triplet is defined using the scoring model described below.

SSE triplet similarity scoring: Given two matching triplets, T^q in a query and T^t in a target (database), the SSE triplet similarity, denoted by $SC_{triplet}(T^q, T^t)$, is computed using the following equation.

$$SC_{triplet}(T^q, T^t) = SC_{pair}(S^q_{ij}, S^t_{ij}) + SC_{pair}(S^q_{jk}, S^t_{jk}) + SC_{pair}(S^q_{ik}, S^t_{ik})$$

where S^q_{ij} and S^t_{ij} denote the equivalent SSE pairs of S_i and S_j.

The score, $SC_{pair}(S^q_{ij}, S^t_{ij})$, is the SSE pair similarity score, and is computed as:

$$SC_{pair}(S^q_{ij}, S^t_{ij}) = \left\{\theta^E - \frac{|D^q_{ij} - D^t_{ij}|}{D^*_{ij}}\right\} \times w(D^*_{ij}) \times (l_i \times l_j)^p$$

where D^*_{ij} is the average of D^q_{ij} and D^t_{ij}, $w(r)$=exp(-$(r/20)^2$),$l_i = \min(L^q_i, L^t_i)$, $l_j = \min(L^q_j, L^t_j)$, $p = 0.6$, and $\theta^E = 0.2$.

In the above equation, the first term measures the distance deviation between two SSE pairs. The second term de-emphasizes the significance of matches between two distant SSE pairs, since “distant SSE pairs are abundant and less discriminate” [46]. The last term scales the score by the maximum aligned portion between the SSE pairs and a parameter p. The parameter p is set to 0.6, which was empirically determined by randomly choosing 400 domains from ASTRAL and computing our SC_{pair} score and DALI score for each SSE pair. The value of $p = 0.6$ produced the maximum correlation between the two scores (correlation coefficient of 0.6).

We note that our scoring equation has a strong similarity to the DALI scoring model. In the DALI model, a similarity score is calculated using all pairwise residue distances, whereas in our model the basic unit of comparison is an SSE rather than individual residues. The scoring using the SSE uses only the information in the index, and is computationally much faster than the scoring function used in DALI (which costs $O(N^2)$, where N is the number of residues).

SSE triplet index search: When matching a query triplet, rather than scanning all the SSE triplets in the database (which can be slow), we use an index search to find all database triplets that are similar to the query triplet. For each database triplet, we then compute the similarity score with the query triplet using the $SC_{triplet}$ equation described below.

The index probe retrieves all matching entries using the following criteria: Given a query triplet T^q and a target triplet T^t, which are defined as below,

$$T^q = (TC^q, X_i^q, Y_i^q, X_j^q, Y_j^q, X_k^q, Y_k^q, D_{jk}^q, D_{ik}^q, D_{ij}^q)$$
$$T^t = (TC^t, X_i^t, Y_i^t, X_j^t, Y_j^t, X_k^t, Y_k^t, D_{jk}^t, D_{ik}^t, D_{ij}^t)$$

T^t is a match for T^q when the following conditions are met.

1.$TC^q = TC^t$,

2.$\forall r = i, j, k(\sqrt{(X_r^q - X_r^t)^2 + (Y_r^q - Y_r^t)^2} \leq sin(\theta) \times \sqrt{X_r^q + Y_r^q})$, and

3.$|D_{jk}^q - D_{jk}^t| \leq d_{\varepsilon 1} \wedge |D_{ik}^q - D_{ik}^t| \leq d_{\varepsilon 2} \wedge |D_{ij}^q - D_{ij}^t| \leq d_{\varepsilon 3}$

The first condition checks to ensure that the two SSE triplets have the same SSE types and the same order for the SSEs. The second condition checks to see if the three SSEs in T^t are within in a small distance $(sin(\theta) \times \sqrt{X_r^q + Y_r^q})$ of the corresponding SSEs in T^q. In our implementation, θ is set to 30°. The final condition checks if the distance between each matching SSE pair is within a small threshold, d_ε. As in the DALI scoring model [46], the exact value for this threshold depends on the types of the SSEs being compared. The distance cutoff is set to 3 A° for a β-strand pair, 4 A° for an α-helix and β-strand pair, and 5 A° for an α helix pair. We note that these cutoff values are higher than the ones used in the DALI model as we are matching SSEs in a triplet, rather than just individual pair without considering a triplet configuration (as is done in DALI). The original DALI cutoffs would be too strict for matching SSE triplets.

Protein structure matching

The previous step produces matching target triplets in the database for every triplet in the query, and the associated matching score ($SC_{triplet}$). Next, we need to assemble these triplet hits into matches for the entire protein domain. For this step, we construct a weighted bipartite graph for *every* target protein domain that has some triplet matches. In each graph, nodes on one side of the bipartite graph represent triplets in the query and nodes on the other side represent triplets in the database entry. An edge between two nodes indicates that the two triplets were matched by the previous step, and the weight of the edge represents the $SC_{triplet}$ score. A maximum weighted bipartite graph matching algorithm is run on this graph to produce an injective (one-to-one) mapping from the query SSE triplets to the triplets in the target. Then, using this mapping, an overall structure similarity score is computed as:

$$SC_{raw}(q,t) = \sum_{i=1}^{M} SC_{triplet}(T_i^q, T_i^t)$$

where T_i^q and T_i^t are equivalent SSE triplets in the one-to-one mapping, M is the total number of equivalent SSE triplet pairs, and $SC_{triplet}(T_i^q, T_i^t)$ is the triplet similarity score between T_i^q and T_i^t.

This raw similarity score depends on the sizes of the query and target protein domains, and is normalized as follows:

$$SC_{norm}(q,t) = (SC_{raw}(q,t) \times R^*)/SC_{raw}(q,q) \text{ , where}$$

$$R^* = \frac{R_{Total}(q,t) + R_{SSE}(q,t)}{2} \text{ ,}$$

$$R_{Total}(q,t) = 1 + \max\left(-1, \log_{10} \frac{N_q}{\max(N_q, N_t)}\right) \text{ , and}$$

$$R_{SSE}(q,t) = 1 + \max\left(-1, \log_{10} \frac{SN_t}{\max(SN_q, SN_t)}\right)$$

In the equations above, N_q and N_t are the total number of residues in the query and the target respectively. SN_q and SN_t are the number of residues contributing to the formation of α-helices and β-strands in the query and the target respectively.

The ratios, R_{Total} and R_{SSE}, scale down the raw score inversely proportional to the size difference between the two proteins, and produce a score that is less sensitive to the size differences between the query and the target. The division by the self-similarity score, $SC_{raw}(q, q)$, produces a normalized similarity score between 0 and 1, which represents how similar the query protein is to a target, compared to itself. The normalized score is reported as the final structure similarity score.

4.2.2 Structure classification

Existing automatic classification methods [17, 20, 35, 74] employ a nearest neighbor classification strategy. Given a query protein domain, they find the structurally closest neighbor that has a known classification label. Then the query is assigned the same label as its nearest neighbor. Although the nearest classification strategy is effective in many cases, it has a significant limitation as proteins with novel folds are guaranteed to be misclassified.

To resolve this problem, the SGM method [74], which employs a modified nearest-neighbor approach, reports a label of *unknown and/or possibly new* when it cannot classify proteins with high confidence. To detect the boundary between classification and non-classification, it uses an inter to intra cluster distance ratio, based on the observation that "chains that are equidistant to several clusters are hard to classify and chains that are far away from any known clusters are probably new folds" [74]. The distance ratio can be effectively used to detect whether a protein is *relatively* closer to a specific cluster than to the remaining clusters. However, it cannot be used

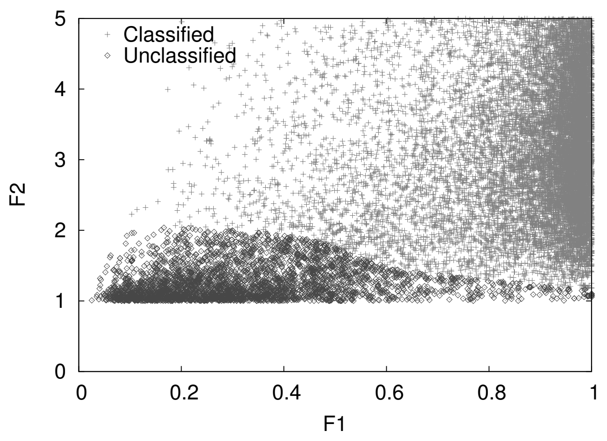


Figure 4.1: Visualization of the classification decision boundary

This figure shows the classification boundary created for entries in SCOP 1.67 using SCOP 1.65 as the database. The SVM is used to detect the boundary between "Classified" and "Unclassified" entries. This trained SVM will then be used to predict class labels for SCOP 1.69.

to detect whether a protein is *absolutely* close to a specific cluster.

Our classification method is also based on the nearest neighbor classification, and adopts the same observation as is used in SGM to detect unknown and/or possibly new folds. However, our method improves classification accuracy by using additional measures and a more sophisticated class boundary detection method, namely an SVM [23]. Furthermore, instead of reporting proteins labels as "unknown and/or possibly new", our method also identifies clusters among unclassified proteins to further automate the classification process.

Classification using an SVM

Our method for assigning a class label uses three pieces of information, namely: an absolute similarity ratio ($F1$), a relative similarity ratio ($F2$), and the nearest cluster classification label ($C1$). This information is collected using the following procedure:

First, given a protein domain q, the structure comparison method, described in the structure comparison section, is used to find the top k structure neighbors in the database. From this top k list, we remove any hits to the query itself.

Then, we pick the top structure, n_1, as the nearest neighbor. Let $C1$ denote n_1's classification label. We then go down the list and find the next structure that has a different label from $C1$. Let us call this entry n_2, and let $C2$ denote the label for n_2. Next, we compute the scores, $SC_{norm}(q, n_1)$ and $SC_{norm}(q, n_2)$. Then we use these scores to compute $F1$ and $F2$ as: $F1 = SC_{norm}(q, n_1)$, and $F2 = SC_{norm}(q, n_1)/SC_{norm}(q, n_2)$. Finally, we return the values $F1$, $F2$, and $C1$.

Intuitively, high $F1$ and $F2$ values indicate that the query is structurally similar to its nearest neighbor, and is also relatively closer to its nearest neighbor than to any other domain, which in turn suggests a high confidence in the assignment of the classification label. On the other hand, low $F1$ and $F2$ values imply that the domain is not particularly similar to any existing domains, which suggests that the domain is potentially a new fold.

To automate the classification process, we need a classification decision model which defines clear boundaries between classification and non-classification. Using SCOP as the gold standard, we generated a classification decision model that reflects the rules used for creating new folds in SCOP. In order to create such classification decision model, we used a support vector machine (SVM) to capture nonlinear classification decision boundaries in SCOP. As a training set for the decision model, we picked SCOP version 1.65 and version 1.67 and trained the model as follows: Using domains in SCOP 1.67 as the queries, and domains in SCOP 1.65 as the database, we perform structure comparison using the method described in the structure comparison section. For each query, we calculate the $F1$ and $F2$ scores. If the SCOP label of a query protein domain is the same as its nearest neighbor's SCOP label, the query with its $F1$ and $F2$ is used as a positive example, otherwise, it is used as a negative example in the training set for the SVM. The resulting training dataset is shown in Figure **??**.

The classification label assignment step simply uses the trained SVM to determine if a query should be labeled as *unclassified*. For queries that the SVM determines can be classified, the label of the nearest-neighbor in the database is used as the predicted class label.

4.2.3 Identification and clustering of novel structures

Our classification method takes the approach of assigning an "unclassified" label to protein domains that have novel folds or have subjective and fuzzy classification boundaries. Assigning an actual class label to such domains often requires additional biological information and manual interpretation [48, 69]. Since such manual intervention is likely to continue to be unavoidable even in the foreseeable future, it is useful if additional information is provided to make a more informed (and potentially faster) manual assignment. In this section, we outline our method for aiding this manual assignment process by employing a clustering method for grouping the protein domains that are labeled as "unclassified" with our classification method.

The basic intuition behind using clustering is that protein domains that are in the same cluster are likely to have stronger similarities to each other, sharing similar protein structures, compared to domains in different clusters. In addition, it is often likely that well-segregated clusters correspond to novel folds. To detect these novel folds, we first perform an all-to-all comparison using all the protein domains that are labeled as unclassified by the previous structure classification step. Then, we construct a graph that has a node for every unclassified domain. In this graph two nodes are connected by an edge if the similarity score between the protein domains corresponding to the nodes is above a certain threshold. Each edge has a weight, which is equal to the similarity score. Once this graph is constructed, the MCL [85] algorithm is run on the graph to detect clusters. (MCL is a clustering algorithm that is specifically designed to work with graphs.) The computed clusters are then reported as groups that potentially correspond to novel folds.

In addition, for each computed cluster we also produce a representative structure, which is simply the graph center for that cluster (if there are more than one centers, we randomly select one of the centers as the representative structure).

4.3 Experiments

4.3.1 Experimental setup and datasets

In this section, we present results measuring the effectiveness of our classification methods. For the empirical evaluation, we employed the experimental strategies used in previous studies [17, 20]: namely, domains in an older version of SCOP are used as the set of database domains with known class labels, and domains in a newer version of SCOP are used as the query set. Classification accuracy is measured by comparing the predicted labels with the (known) labels in the newer version of SCOP. In our experiments, SCOP 1.67 and SCOP 1.69 are used as the database and the query set respectively.

SCOP 1.67 and 1.69 contain 65122 and 70859 domains, which are grouped into 2630 and 2845 families respectively. However, in our evaluation theoretical domains and domains with less than 3 SSEs are excluded. After these exclusions, we end up with 58456 and 63745 domains in SCOP 1.67 and 1.69 respectively. Our database is the set of 58456 domains in SCOP 1.67, and our query set is the 5289 newly added domains in SCOP 1.69. We used the ASTRAL Compendium [19] for the PDB-style coordinate information for these SCOP domains. In addition, we used the STRIDE program [30] to generate secondary structure assignments for each domain.

Our implementation is written in C++, and uses the LEDA 3.2R package for the maximum bipartite graph matching, and the SVM^{light} [5] package. The SVM model was trained using SCOP 1.65 and SCOP 1.67 (see the structure classification section in Methods). We used a radial basis function as the kernel with a weight cost set to the ratio of the number of negative examples to the number of positive examples.

All experiments were run on a 2.2 GHz Opteron machine, with 4 GB of RAM, and running the Linux 2.6.9 kernel. Throughout this section, we will use the term *class* to refer to a class in the classification scheme.

4.3.2 Experimental evaluation

Precision and computational cost

To measure the effectiveness of our classification method, we compare the predicted classification label (at the fold, superfamily, and family levels) with the actual label in SCOP 1.69 using the following metrics:

Overall precision	=	(CC + UN) / (TE + TN)
Classification error ratio	=	CI / (CC + CI)
New class detection ratio	=	UN / TN

In the above equations, CC is the number of correctly classified domains, and CI is the number of incorrectly classified domains. UN represents the number of domains of new structures which are not in existing classes and therefore are correctly marked as *unclassified*. UE is the number of domains which should have been classified into existing classes, but which are marked as *unclassified* by our method. (Note $CC+CI$ is the total number of domains that are assigned some labels by our method, and $UN + UE$ is the total number of domains that are tagged as *unclassified* by our method.) TE represents the total number of domains in common classes in SCOP 1.67 and SCOP 1.69, and TN represents the total number of domains in new classes in SCOP 1.69.

Overall precision measures how many proteins are correctly classified or correctly labeled as unclassified. The *classification error ratio* measures how many errors are made when query domains are assigned actual labels. *A new class detection ratio* measures how effectively a method can detect domains that are in new classification

Table 4.1: Classification result for proCC using new domains in SCOP 1.69

	Classified domains		**Unclassified domains**		**Total domains**	
			New classes	Existing classes	New classes	Existing classes
	Correct CC	Incorrect CI	UN	UE	TN	TE
Family	4008	347	555	379	726	4563
Superfamily	4321	154	292	522	353	4936
Fold	4597	159	153	380	209	5080

	Overall precision $\frac{(CC+UN)}{(TN+TE)}$	**Classification error** $\frac{CI}{(CC+CI)}$	**New class detection ratio** $\frac{UN}{TN}$
Family	86.3%	8.0%	76.5%
Superfamily	87.2%	3.4%	82.7%
Fold	90.1%	3.3%	75.0%

This table shows the result of classifying 5298 new domains in SCOP 1.69 using proCC.

classes.

The results for this experiment are shown in Table 4.1. As can be seen from this table, our classification method is highly accurate and is fairly effective in detecting domains that are in new classification classes.

With respect to the computation time for classification, the computation cost is linearly proportional to the number of SSE triplets in the query. The average number of SSEs per domain is about 77, and for queries of this size, our method requires about 30 seconds of execution time. Of this computation time, the index matching component takes about 38% of the time (This index search time is about 8 times faster than a full scan of the file that has all the SSE triplets). About 56% of the

Table 4.2: The comparison between SGM and proCC

	Overall precision		**Classification error ratio**		**New class detection ratio**	
	SGM	proCC	SGM	proCC	SGM	proCC
Family	71.3%	86.3%	19.7%	8.0%	77.4%	76.5%
Superfamily	69.6%	87.2%	17.0%	3.4%	82.2%	82.7%
Fold	71.3%	90.1%	15.7%	3.3%	76.6%	75.0%

This table shows the result of comparing SGM and proCC for classifying 5298 new domains in SCOP 1.69.

computation time is spent on the overall structure matching component (the bipartite graph matching method), and the remaining 6% of the time is spent for program setup, input and output processing, and SVM classification (see the Methods section for description of these components).

Comparison with other methods

A number of methods have previously been proposed for automatic classification [17, 20, 35, 74]. In evaluating performance, we considered comparing our method with each of these methods. However, some of these methods are not suitable for comparison because of the following reasons. Currently, a fair comparison with Superfamily [35] is not possible since a SCOP 1.67 Hidden Markov Model is required for comparison, and this model is currently not available (Personal Communication, Derek Wilson, 2006). Comparison with [17] is not possible since its implementation or result data sets are not available.

Therefore, in this section, we compare our method with SGM [74] and SCOPmap [20]. The SGM method is a classification method based on 30-dimensional Gaussian integrals of protein structures, and nearest neighbor classification. The SGM method has been shown to be very fast and effective for classifying CATH. SCOPmap is a consensus-based method that uses seven different sequence and structure comparison methods. SCOPmap has been extensively compared with Superfamily, and has been shown to be more accurate than Superfamily [20].

Comparison with SGM: Before presenting the results with SGM, we note that the performance of SGM can change depending on adjustable parameters, such as the distance ratio cutoff in SGM. We experimented with a variety of parameter settings for SGM and found that settings that increase the new class detection ratio (or decrease the classification error ratio), degrade the overall precision. To select a reasonable baseline for comparison, we picked parameter values for SGM which produce a new class detection ratio similar to our method. With this method, we end up with distance ratio cutoff values of 1.22, 1.23, and 1.23 at the family, superfamily, and fold levels respectively.

The results comparing SGM and proCC for the 5289 new domains in SCOP 1.69 are shown in Table 4.2. Although SGM was very effective for classifying CATH, this method is less successful with SCOP. As these result shows, our method is 15-19% more accurate than SGM at the family, superfamily, and fold levels, and makes fewer misclassification mistakes.

We have also evaluated proCC, and compared it with SGM, using CATH (SGM was originally only tested against CATH). We used CATH 2.0 and CATH 2.4 as the database and query domains. The overall precision of the SGM method in classifying

Table 4.3: The comparison between SCOPmap and proCC using the predicted superfamily SCOP labels

	Classified with correct domain boundary		**Unclassified with correct domain boundary**		**Incorrect domain boundary**
	Correct CC	Incorrect CI	New classes UN	Existing classes UE	ID
SCOPmap	2069	65	190	212	237
proCC	2025	75	246	275	152

	Overall precision $\frac{(CC+UN)}{2773}$	**New class detection ratio** $\frac{UN}{307}$	**Estimated average execution time**
SCOPmap	81.5%	61.9%	2-3 hours per query
proCC	81.9%	80.1%	9 minutes per query

This table shows the result of classifying 2773 single domain chains in SCOP 1.69. All numbers reported in column 2-6 are in terms of the number of chains (or domains due to the fact that we used single domain chains). Column 2-3 show the number of single domain chains which are correctly identified as single domain chains and are classified to known superfamilies. Column 4-5 show the number of single domain chains which are correctly identified as single domain chains and are labeled as unclassified. Column 6 shows the number of single chain domains which are incorrectly identified as multi-domain chains.

CATH is 93.9%, 94.5%, 94.7%, and 97.1% at the H, T, A, and C levels, whereas the overall precision of our method is 94.1%, 95.6%, 95.6% and 97.2% at the H, T, A, and C levels. Compared to SCOP, both methods generate more accurate results with CATH. However, the higher precision with CATH is expected since CATH uses a broader definition of fold, i.e. there are fewer folds in the CATH classification compared to SCOP [25].

Comparison with SCOPmap: In this section, we present results comparing

SCOPmap and our proCC method. In comparison with SCOPmap, we note that SCOPmap takes as input a query protein chain, identifies domains by aligning the query protein chain to sequences and structures in its database, and assigns a classification label to each identified domain. On the other hand, the input to proCC is a domain rather than a protein chain. So for comparison with SCOPmap, we first ran a domain prediction method with query protein chains to identify the domain boundaries. Then, we ran our classification method on the identified domains. For domain boundary prediction, we used the SSEP-domain method [26], which was shown to be very accurate in the CAFASP4-DP competition [1].

We compared SCOPmap and proCC using 2773 new single domain chains in SCOP 1.69. For this experiment, multi-domain chains are excluded, due to the difficulty in measuring effectiveness objectively (In the case of multi-domain chains, the number of predicted domains, predicted domain boundaries, and the number of correct domain classification assignments all need to be considered, and there is no systematic way of differentiating these effects from the actual classification effectiveness which we aim to evaluate).

Initially, we attempted to run SCOPmap on the 2773 chains. However, running SCOPmap on these 2773 chains takes an enormous amount of computational resource requiring approximately 2-3 hours to process each individual chain (Personal Communication, Sara Cheek, 2006). Due to this high computation cost, new proteins are typically classified using large clusters and classification results are posted at ftp://iole.swmed.edu/pub/scopmap. Therefore, we compared our method with SCOPmap based on the latest result posted on the SCOPmap ftp site.

Finally, while our method can predict the family, superfamily, and fold labels, SCOPmap primarily predicts the superfamily label, and only predicts the fold label for queries that it cannot assign a superfamily label. SCOPmap never predicts a family label. Since the main classification prediction made by SCOPmap is at the superfamily level, for this evaluation we compared the classification effectiveness only at this level. The results of this evaluation are shown in Table 4.3. From this table we can make the following observations:

(1) Overall precision: By examining column 5 in Table 4.3, we observe that the overall precision of SCOPmap is marginally lower than proCC with the SSEP-domain prediction method. From column 4, we also observe that the SSEP-domain prediction method performs better than SCOPmap in identifying single domain chains. To isolate the effect of domain prediction from the classification accuracy, we also measured overall precision as (CC + CI) / (2773 - ID). This adjusted overall precision is 89.1% and 86.7% for SCOPmap and proCC respectively. We note that SCOPmap is tightly coupled with its domain prediction method, and considered as an entire package, proCC coupled with SSEP provides slightly higher overall precision than SCOPmap. Furthermore, the added advantage of our approach is that it can be coupled with any domain prediction method allowing our approach to easily leverage future improvements in domain predication methods.

(2) Detection of novel structures: From column 6 in Table 4.3, we observe that with respect to detecting novel structures, our method is about 20% more accurate than SCOPmap. The reason for this difference is that SCOPmap aggressively classifies a query into a known classification class if at least one of the 7 sequence and structure comparison methods can find a significant match to the query. This approach can be effective when the query belongs to a known class, but is vulnerable to making false predications for queries that have novel structures, especially when classification boundaries for those structures are ambiguous. On the other hand, our method makes a classification decision based on a sophisticated decision model, which distinguishes novel protein structures from known protein structures based on knowledge learned from a prior classification database.

(3) Computational cost:

With respect to computation time (see the last column in Table 4.3), our method has a clear advantage over SCOPmap. While SCOPmap takes on average 2-3 hours per query, our method can classify a query on average in 9 minutes. Of these 9 minutes, on average 8 minutes are spent on the SSEP domain prediction web service, and on average only 1 minute is spent in our classification method. We recognize that a technique to address the significantly higher computational cost of SCOPmap is to employ a large cluster. While this solution is practical in some cases (although very costly), with the increasing rate of production of new structures it may be more practical to employ a much cheaper solution like proCC which has comparable precision and offers more flexibility as it can be coupled with any domain prediction tool.

Finally, we note that in contrast of SCOPmap, proCC also provides classification predictions for the SCOP family level. Such predictions are useful as it is known that several domains in the same superfamily can be functionally divergent, and a more fine-grained family level classification is more useful for predicting domain

Table 4.4: The clustering effectiveness at the SCOP family, superfamily, and fold levels

	SCOP Classes (A)	**MCL Clusters (B)**	**# of common clusters/classes (C)**	**# of correctly labeled domains in (C)**
Family	320	358	301	822 (88%)
Superfamily	260	327	234	731 (78%)
Fold	200	318	191	670 (72%)

This table shows the result of clustering 934 unclassified domains at the SCOP family, superfamily, and fold levels. Column 2 shows the number of SCOP families, superfamilies, and folds that these 934 domains are spread across. Column 3 shows the number of automatically generated clusters at each SCOP level. Column 4 shows the number of common clusters/SCOP classes that were correctly mapped. Column 5 shows the number of actual domains in the cluster that had the same label as the corresponding SCOP class.

functions [65].

Detection and clustering of novel families, superfamilies, and folds

From the query set of 5289 domains, our classification method labels 934 domains as *unclassified*. As a way of identifying and describing novel families, superfamilies, and folds among these unclassified domains, we ran the MCL clustering algorithm on a graph constructed using these unclassified domains. To construct a graph for clustering, a threshold value for structure similarity is required (see the identification and clustering of novel structures section in Methods). In addition, for the clustering at the different SCOP levels, different threshold values are needed. For this experiment, we set the threshold value to 0.4, 0.32, and 0.3 for the family, superfamily, and fold levels respectively, based on the observation that more than 90% of correctly classified proteins have a similarity score above these values with their nearest structure

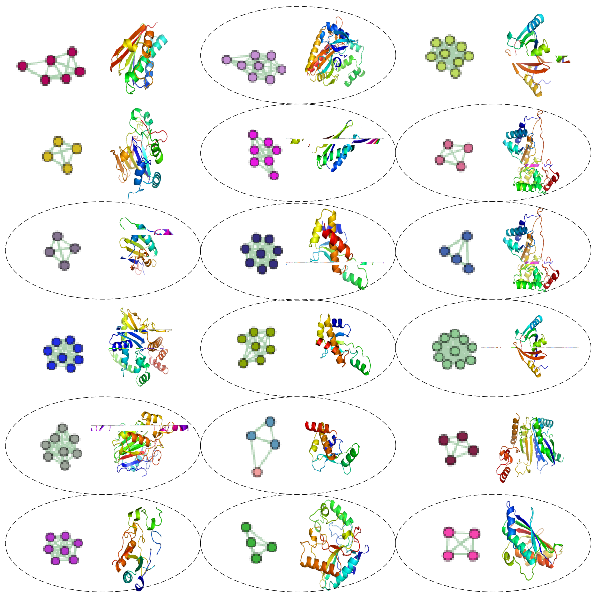


Figure 4.2: Assessing the quality of the automatically generated clusters

This figure shows the automatically generated family-level clusters for the unclassified domains in the SCOP 1.69 "d" class (i.e. the alpha and beta proteins (a+b)). This figure also shows the representative domain structures for each cluster. A connected graph corresponds to an automatically detected MCL cluster. The ellipses indicate the novel families in SCOP 1.69. The MCL clusters are assigned a family-level label based on the most common family-level label in the cluster. Within a cluster, the nodes with the same color indicate that all these nodes have the same family-level label. To keep this figure simple, only clusters with more than four domains are shown. There are an additional of 79 clusters that matched the SCOP family label, and of these 30 clusters correspond to new families in SCOP 1.69. This figure was generated using BioLayout [27] and PyMol [4].

neighbor in the same SCOP family, superfamily, and fold.

To measure the capability of the automated method in identifying novel SCOP families, we compared the automatically produced clusters with the family level classes in SCOP 1.69. The 934 unclassified domains are spread across 320 families in SCOP 1.69. For these domains, the automated method produced 358 clusters. To check the agreement between SCOP and the automatically generated clusters, we generated class labels for the clusters based on the most common family label in a cluster. Based on this class label assignment, each SCOP family is paired with one or zero cluster having the same class label. When more than one cluster maps to the same SCOP family, we count only the assignment of one of the automatically generated cluster; this cluster is the one in which the number of domains that correctly match the SCOP family label is highest amongst the set of clusters that also have the same SCOP family label. We then counted the number of common clusters/families that were "correctly" mapped, and found that there are 301 common clusters between the two classifications. Then, for each correctly mapped cluster, we counted the number of actual domains in the cluster that had the same label as the corresponding SCOP family. This total is 822, which is 88% of the total number of unclassified domains.

Using the same method, we also computed the clustering effectiveness at the superfamily and fold levels. These results are shown in Table 4.4.

In Table 4.4, of the 358 identified clusters at the family level, 159 clusters actually correspond to 159 novel families in SCOP 1.69, which is 74% of the 215 total number of novel families introduced in SCOP 1.69. At the superfamily level, out of 327 identified clusters, 62 clusters actually correspond to 62 novel superfamilies in SCOP 1.69, which is 65% of the 95 total number of novel superfamilies introduced in SCOP 1.69. At the fold level, out of 318 identified clusters, 46 clusters actually correspond to 46 novel families in SCOP 1.69, which is 75% of the 61 total number of novel families introduced in SCOP 1.69.

In addition, to measure the extent of homogeneity in automatically generated clusters, we also evaluated the quality of clusters using a measure called "cluster purity" [36]. It is 1 when all domains in the same cluster have perfect agreement in their class labels, and it is defined as:

$$ClusterPurity(\mathbb{C}, \mathbb{S}) = \frac{1}{N} \sum_{C \in \mathbb{C}} \max_{S \in \mathbb{S}} |C \cap S|$$

In the above equation, C is a cluster in the set of MCL clusters $\mathbb{C}$, S is a family in the set of SCOP families $\mathbb{S}$, and N is the total number of domains in $\mathbb{S}$.

Using this measure, the cluster purity of the MCL clusters is 0.96, 0.95, and 0.96 at the SCOP family, superfamily, and fold levels respectively. This high cluster purity value shows that our clustering method produces clusters that have a high degree of agreement with the SCOP classes. An example of automatically clustered novel SCOP families is shown in Figure 4.2.

4.4 Discussion

4.4.1 Applications for efficient structure comparison

In general, existing protein classification methods have focused on classifying new domains into existing classification hierarchies. However, it has been observed that in SCOP previously classified domains are often rearranged in subsequent releases, as new structures sometimes reveal more relationship amongst new and existing domains [22]. Therefore, in addition to classifying new structures, it is to automatically detect such potential rearrangements.

One way of approaching this problem is to perform an all-to-all comparison with existing and new domains, and then generate clusters using a clustering method. For instance, if the introduction of a new domain provides evidence connecting previously unrelated domains, a cluster that consist of these domains can be found, suggesting some potential rearrangements involving these domains.

In performing this task, along with clustering techniques, an efficient and accurate structure comparison method is crucial since one has to compare each pair of structures ($O(n^2)$ comparisons). Our structure comparison method (see structure comparison section in Methods), is very efficient and could be a suitable choice for this task.

Incorporating this functionality to continually detect rearrangements of the classification hierarchy into our classification framework will be part of our future work.

4.4.2 Integration with domain prediction methods

To make a domain classification truly automatic, given a protein structure, first the domain boundaries must be identified. The domain boundary prediction problem is well recognized as a crucial component for functional classification and structure prediction [26], and there are a number of competing domain prediction methods [79]. The proCC method provides a framework which allows us to couple our classification method with any domain prediction tool. While we have used the SSEP-domain method in our current study, other domain prediction methods, for instance, Rosseta-Ginzu [21], which is more accurate but slower, can be used to potentially produce even better classification results. In addition, the loose coupling between the classification and domain prediction components will easily allow us to leverage future advances that are likely to be made in domain boundary prediction methods.

4.5 Conclusion

In this paper we have described a method called proCC for automatically classifying proteins. Using extensive experimental evaluation, we have demonstrated that our method often has higher accuracy compared to existing automated methods. Our method is also very effective in predicting new folds, and is very efficient. While our method cannot completely remove the need for manual intervention that is invariably needed in producing high-quality classification hierarchies such as SCOP and CATH, it can provide a valuable complimentary method for classifying new domains that have not been incorporated into the latest releases of these databases. In addition, our method can also help the curators of these databases in reorganizing the existing classification hierarchies to accommodate new protein structures.

CHAPTER V

Evaluation of Multi-dimensional Indexing Structures for Range Query

5.1 Introduction

Due to the demanding need for efficient multi-dimensional indexing methods in many database applications, significant research effort has been invested towards developing new multi-dimensional indexing methods. There are more than a few dozen indexing structures for multi-dimensional data [12, 13, 14, 31, 63, 80], and each year a few more indexing structures are proposed.

Of these indexing methods, the R*-tree [12], one of the R-tree [37] variants, is most widely used. Since it is well-known that the performance of the R*-tree deteriorates rapidly with increasing data dimensionality [14], the R*-tree is not used for handling very high-dimensional datasets. High-dimensional datasets are plagued with the curse of dimensionality, and general high performance high-dimensional indexing methods that work across a variety of applications still remains a legitimate research goal (and is likely to continue to be an open research problem for at least the near future). However, for many high-dimensional applications, a practical method is to apply a dimensionality reduction method such as principal components analysis (PCA) to transform the high-dimensional dataset into a low or medium dimensional datasets (usually with 8 or fewer dimensions). The use of R*-tree for low and medium dimensional point indexing has been advocated for many years and many applications have been built using R*-trees. A few examples of such applications are similarity search in sequence databases [7], image retrievals in multimedia databases [28, 71], subsequence matching in time-series databases [67, 68], and similarity searches in protein databases [87]. A common characteristic of these examples is that they all rely on R*-trees for indexing multi-dimensional point datasets, which usually have less than 8 dimensions.

However, as shown in several previous comparative studies [31], there is no single index structure that works efficiently across a variety of applications. In addition, choosing a right index structure for a given application requires considering various factors such as the type of query operations and data update characteristics. The relative importance of each criteria and consideration of these factors will lead to a different choice of an index structure. Previous comparative studies have not adequately considered these factors.

In this work, we benchmark these popular indices, R*-tree, Quadtree, and Pyramid-Technique, for low and medium dimensional point data, using various criteria. Previous works [44, 45, 57] have also carried out performance comparison of R*-tree and Quadtree using 2 dimensional spatial data. What makes our work different from previous studies is that our study compares index structures using higher dimensional point data, which is required in many modern database applications [7, 28, 67, 68, 71, 87]. Furthermore, our study considers various important factors that must be considered when choosing an index structure. These factors include the characteristic of data update, such as whether dataset is static or dynamic.

Among several index structures supporting for multi-dimensional point data [13, 14, 31, 63], we choose and compare the R*-tree, Quadtree, and Pyramid-Technique for the following reasons. The R*-tree has been chosen since it is widely used in many applications.

The reason for choosing the Quadtree is that it has been around for many decades even within the context of database systems [32, 43], but it is still not as widely accepted as the R*-tree. We speculate that the reasons for largely ignoring the Quadtree for database indexing are: 1) the unbalanced nature of the index structure, especially for skewed data, and 2) the mismatch between the size of a non-leaf node and a disk block size. Nevertheless, the Quadtree has some nice properties as it employs a disjoint and regular space partitioning strategy. This partitioning method has an advantage over the R*-tree which suffers from rapidly increasing overlap of MBRs even for relatively low dimensionality. In addition, drawbacks related to the large fan-out and low node occupancy in the Quadtree can be alleviated to some extent with efficient node *packing* techniques which we employ in our implementation.

The Pyramid-Technique [13] is selected for the comparison since it was shown to be far superior to the Hilbert R-tree [50] and the X-tree [14]. In addition, the Pyramid-Technique is based on the disjoint space partitioning similar to the Quadtree.

However, instead of using a regular space decomposition strategy, it adopts a non-regular space decomposition strategy. Including the Pyramid-Technique allows us to examine how different space partitioning strategies affect the performance of the indices.

We note that there are a very large number of multi-dimensional indexing structures (see [31] for the long list 10 years ago) and every year we see a number of new indexing structures being proposed, usually for specific applications. Invariably, in our selection of these three indexing structures, we potentially will have missed the readers' favorite indexing structure. However, it is virtually impossible to thoroughly evaluate all these indexing structures, and the purpose of this study is not to conduct a thorough evaluation of all existing multi-dimensional indexing structures. Rather we choose the R*-tree that is most commonly employed spatial indexing structure in the database community, the Quadtree which is widely used in graphics applications and has a very different partitioning philosophy compared to the R*-tree, and the Pyramid-Technique which provides a novel way of using an B^+-tree for multi-dimensional indexing.

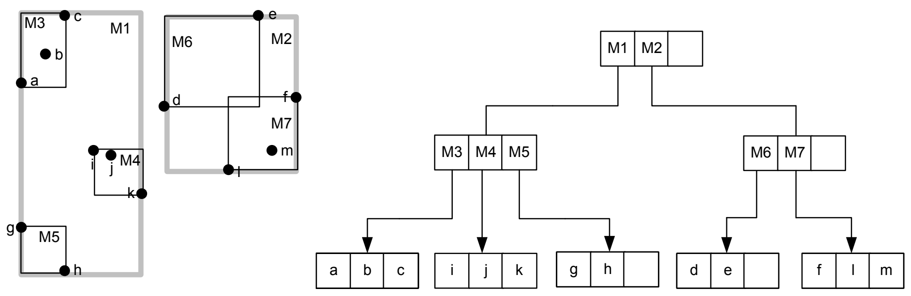
Having chosen the above indices for comparison, we use the following criteria for benchmarking the indices.

Data update characteristic - whether data is static or dynamic: Databases deal with various types of data which can be static, dynamic, or partially dynamic. With respect to data update, GIS databases are mostly static. However, time-series databases such as stock data or moving object databases can be highly dynamic. Such data update characteristics are very important in selecting the right index *con-*

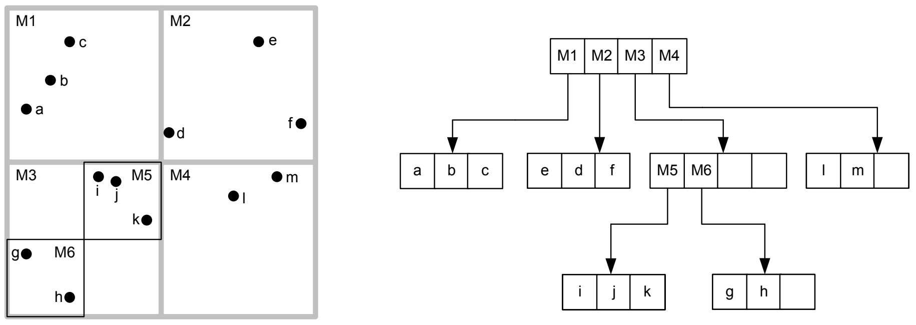
struction method. For instance, when data is static, constructing an index using bulkloading algorithms [42, 49, 61, 76] is the best choice since bulkloading algorithms build a good index structure, and also reduce index construction time. When data is highly dynamic, an index needs to be constructed using a dynamic algorithm (by inserting one entry at a time). Although this data update characteristics and applicable index creation methods need to be considered in selecting an index structure, the issues that arise in each situation and the impact on the index performance have not been systematically examined. Therefore, in this work, we also investigate the index performance associated with index construction methods.

Our experiments based on the above criteria reveal an interesting point. Depending on whether data characteristic is static or dynamic, the best performing index structure can change. For instance, when data is dynamic, the Quadtree begins to outperform the R*-tree. The primary reason for the better performance of the R*-tree with static data is due to the use of a STR bulkloading algorithm [61]. However, once the dynamic R*-tree algorithm [12] is used, overlaps amongst MBRs increase with increasing data dimensionality, and it results in degrading the R*-tree performance.

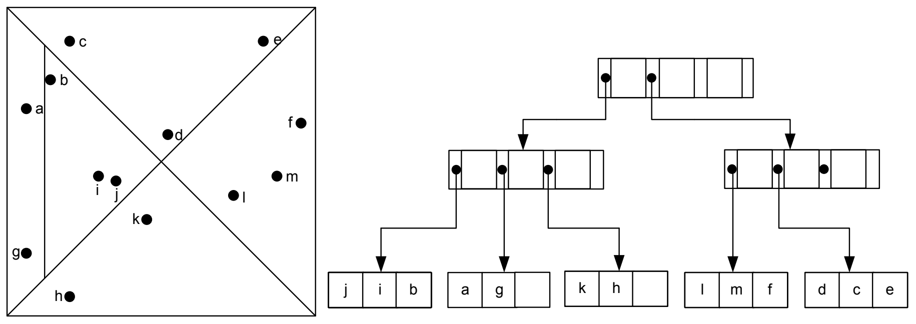
The rest of this chapter is organized as follows. Section 5.2 describes index structures evaluated in this work. Section 6.2 describes the implementation details of each index structure. Section 5.5 presents our experimental results. Section 6.6 summarizes our results.



(a) R* tree



(b) Quadtree



(c) Pyramid-Technique

Figure 5.1: The index structures of the R*-tree, Quadtree, and Pyramid-Technique for an example dataset.

5.2 Background

In this section, we give a brief overview of three indices, the R*-tree, Quadtree, and Pyramid-Technique. Figure 5.1 shows an example of the three index structures constructed for the same set of data points (In the example, it is assumed that a maximum leaf node capacity is equal to 3). As shown in Figure 5.1, these three index structures are distinguished from each other in the following ways:

1. The R*-tree index is based on the decomposition of *data* into partitions, while the Quadtree and Pyramid-Technique indices are based on the decomposition of the underlying *space*.

2. While the R*-tree index does not result in disjoint partitions, the Quadtree and Pyramid-Technique indices generate disjoint partitions (i.e. the partitions do not have any overlap).

3. The Quadtree index is based on a regular and balanced space partitioning, but the Pyramid-Technique index is based on a irregular and unbalanced space partitioning.

4. The R*-tree and Pyramid-Technique indices generate balanced index structures, while the Quadtree index can be an unbalanced structure.

5. As an index structure, the Pyramid-Technique utilizes the 1-dimensional B^+ tree while the R*-tree and Quadtree indices use d-dimensional tree structures.

5.2.1 R*-tree

The R*-tree [12] is the most popular variant of the R-tree [37]. The R-tree is an index structure based on a hierarchical decomposition of data. In the R-tree, data entries close in space are grouped together using minimum bounding rectangles (MBRs). Then the MBRs are stored as internal nodes and data entries are stored in leaf nodes of a balanced tree structure. In the R-tree, an index structure is not unique for a given set of data point; i.e. a different order of data insertions can result in a different index structure for the same dataset. Also, in an R-tree, the underlying partitions imposed by the index can overlap. The basic index structure of the R*-tree is the same as the R-tree, but it uses more sophisticated node insertion and splitting algorithms to reduce the size of the MBRs and the overlaps amongst MBRs. An example of the R*-tree for a set of 2 dimensional point dataset is shown in Figure 5.1 (a).

5.2.2 Quadtree

The Quadtree is an index structure based on a hierarchical and *regular* decomposition of space. The Quadtree partitions the d-dimensional data space by recursively dividing it into 2^d nodes, each of which corresponds to a d-dimensional hyper-rectangle of the equal size. Each leaf node in the Quadtree has a maximum capacity, so a leaf node is split when it reaches the maximum capacity. When a leaf node is split, the Quadtree creates 2^d new child nodes and distributes data into the 2^d new child nodes. An example of the Quadtree for a set of 2 dimensional point dataset is shown in

Figure 5.1 (b).

5.2.3 Pyramid-Technique

The Pyramid-Technique [13] is based on transforming high dimensional points to one dimensional points, and then using the B^+ tree to index the one dimensional points. The transformation is performed based on the following space partitioning strategy. Given a d-dimensional dataset, first the Pyramid-Technique divides the data space into $2d$ pyramids, each of which has a center point in the data space as their top and $(d-1)$-dimensional surface of the data space as their base. For each pyramid, a unique integer p is assigned. Next, each pyramid is subsequently divided into several partitions where each partition corresponds to a data page in the B^+ tree.

A d-dimensional data point, v, is transformed into a one dimensional point using a pyramid number p and a height h of the point in that pyramid. The pyramid p is the pyramid in which the data point v is located. The height h is the distance between x-th dimensional value in v and the center of x-th dimensional data space. Here, x is defined as the dimension, (p mod d), and the center of x-th dimensional data space is 0.5 (assuming that the data space is $[0..1]^d$).

$$h = |0.5 - v_x| \text{ where } x = p \bmod d$$

Once the pyramid value p and the height h of the point is computed, the d-dimensional point v is mapped to a one dimensional pyramid value, $p + h$. Using the pyramid value as a key, the point is stored in a B^+ tree. An example of the Pyramid-Technique for a set of 2 dimensional point dataset is shown in Figure 5.1

(c).

5.3 Implementation

In this section, we first describe our implementation of the R* tree, Pyramid-Technique, and Quadtree indices as disk-based index structures in the SHORE [16] storage manager. Then, we describe our implementation of range query algorithm that use these index structures.

5.3.1 Index Construction

5.3.1.1 R*-tree

The R*-tree [12] is the most popular variant of the R-tree [37]. The R-tree is an index structure based on a hierarchical decomposition of data. In the R-tree, data entries close in space are grouped together using minimum bounding rectangles (MBRs). Then the MBRs are stored as internal nodes and data entries are stored in leaf nodes of a balanced tree structure. In the R-tree, an index structure is not unique for a given set of data point; i.e. a different order of data insertions can result in a different index structure for the same dataset. Also, in an R-tree, the underlying partitions imposed by the index can overlap. The basic index structure of the R*-tree is the same as the R-tree, but it uses more sophisticated node insertion and splitting algorithms to reduce the size of the MBRs and the overlaps amongst MBRs.

SHORE has an R*-tree implementation, but it only works for two dimensions. So, following the description in [12], we implemented a general R*-tree in SHORE which supports multi-dimensional data. To make sure that our R*-tree implementation is not slower compared to the 2D R*-tree implementation in SHORE, we compared the performance of window queries on 2D datasets for the two implementations, and found that our implementation is actually consistently faster than the implementation in SHORE about 2-3 times.

5.3.1.2 Sort-Tile-Recursive R-tree (bulkloaded R-tree)

An alternative to constructing an index by inserting one tuple at at time, a bulkloading method can be used to build the index more efficiently. An R-tree bulkloading algorithm is not only faster for building an index, but the resulting index often has a better structure that typically has less MBR overlap and higher space utilization. This improved R-tree structure ultimately improves query operation times.

Among previously proposed R-tree bulkloading algorithms, we chose the Sort-Tile-Recursive(STR) algorithm [61] which is shown to outperform other existing bulkloading algorithms [49, 76]. The basic idea of the STR algorithm is to recursively split data space into partitions using the following partitioning rule. Given a d dimensional data containing n points and a disk page holding up to f points, data points are first sorted by the first dimension and divided into $\lceil p^{\frac{1}{d}} \rceil$ partitions where $p = \lceil n/f \rceil$ (p is the number of leaf pages in an index). As a result, each partition contains $f * \lceil p^{\frac{d-1}{d}} \rceil$ points. Next, each partition is recursively divided in the same way using the remaining $d - 1$ dimensions, and this procedure is repeated until all partitions contain no more than f points. Once all partitions are generated, the R-tree is constructed in a bottom-up fashion: Each partition is mapped into a leaf node, and then leaf nodes are aggregated to generate a non-leaf node. This process is repeated until a root node is created. With multi-dimensional point data used in this study, note that this algorithm generates leaf nodes with almost no overlaps although there can be some overlaps among non-leaf nodes depending on an aggregation rule used.

5.3.1.3 The Pyramid-Technique

The Pyramid-Technique is based on transforming high dimensional points to one dimensional points, and then using the B^+ tree to index the one dimensional points. For window query processing, the query is transformed into a set of range searches on the B^+ tree. We implemented both the naive and extended Pyramid-Techniques as described in [13]. The naive Pyramid-Technique is used for uniform data and the extended Pyramid-Technique is used for skewed data.

For the B^+ tree construction, the SHORE B^+ tree implementation is used: For static dataset the SHORE B^+ tree *bulkloading* algorithm is used, and for dynamic data B^+ tree is constructed by inserting one tuple at a time.

5.3.1.4 The Quadtree

We implemented the Quadtree in the following way: A Quadtree is initially created using records of a constant size for the non-leaf nodes, and using a page size for the leaf nodes (technically we are implementing a bucket Quadtree [80]). While building a Quadtree, we keep track of an overall index leaf node occupancy, which is defined as the number of entries inserted so far over the total number of entries that can be stored in current leaf nodes. Whenever the index leaf node occupancy is below a defined threshold (typically set to 0.5), the Quadtree is packed as follows: The entire tree is traversed using a breadth first search and each visited node is copied and written sequentially to disk, producing a new Quadtree index file. When copying each node, leaf nodes are packed to variable size records, leaving only a small amount of space to accommodate future updates. This resulting index has a compact disk layout, and the breadth-first traversal method for packing is a natural order for packing as siblings nodes are frequently co-accessed during the processing of a query. Consequently, the resulting Quadtree index is very efficient for index access.

We also implemented a bulkloading algorithm similar to the one described in [42]. The algorithm is designed for a linear Quadtree and requires control over selecting and flushing Quadtree pages in the buffer pool. However, the SHORE buffer pool management is transparent to our Quadtree implementation, and we cannot directly control the behavior of the SHORE buffer pool manager. For this reason, the algorithm is not directly applicable to our Quadtree implementation in SHORE, but the algorithm can be adapted to our implementation. The key idea in the algorithm is to sort data by their spatial location in the Quadtree and insert the data in that order. This approach reduces random and frequent disk I/Os, leading to a faster index construction time.

Adapting the idea, our Quadtree bulkloading is implemented as follows: Given a d dimensional dataset containing n points and a leaf node capacity f, assuming a uniform distribution of data, an expected leaf level l is calculated. Based on these parameters, a grid with 2^{d*l} cells is constructed (These cells correspond to the Quadtree MBRs at level l). Finally, each grid cell is processed one at a time and points in the grid cell are inserted, one at a time, into the Quadtree. After inserting all data points, in order to create the Quadtree with 100% space utilization, the Quadtree is rewritten using the packing algorithm described above with no extra space in each leaf node. Note that this Quadtree bulkloading algorithm mainly reduces the index construction time and index size, and it is different from the R-tree bulkloading algorithms that change the actual index structure. The performance comparison between dynamic and bulkloaded Quadtree construction is presented in Table 5.3 and 5.4.

5.3.2 Range Query Processing

Given an index I and a query window R, the range query returns all points $\in I$ that are contained in R. The range query processing in the R*-tree and Quadtree is based on a recursive traversal of an index. Starting from the root node of the index I, the search algorithm checks whether the MBR of a given node overlaps the query window R. If so, it recursively traverses its child nodes. When it reaches a leaf node, it returns data points in the leaf node that are contained in the window R. The range query processing in the Pyramid-Technique is based on the transformation of the window R into a set of range searches on the B^+ tree and implemented as described in [13].

5.4 Experimental Setup

In this section, we present experimental results comparing R*-tree, Pyramid-Technique, Quadtree indices and a sequential file scan implemented in SHORE. The

Dataset	R*tree	Pyramid-T.	Quadtree	File
Uniform-2D	3,041	6,950	4,124	5,918
Uniform-4D	5,220	8,934	6,227	7,906
Uniform-8D	11,790	12,994	10,889	11,835
MAPS-2D	3,041	7,058	3,441	6,009
MAPS-4D	5,220	9,072	5,531	8,027
MAPS-8D	11,790	13,194	12,555	12,017

Table 5.1: Index size with bulkloading (# pages)

Dataset	R*tree	Pyramid-T.	Quadtree	File
Uniform-2D	4,267	9,557	4,124	5,918
Uniform-4D	6,822	12,921	8,267	7,906
Uniform-8D	12,422	18,599	14,529	11,835
MAPS-2D	5,387	10,377	4,720	6,009
MAPS-4D	7,468	13,578	7,425	8,027
MAPS-8D	13,221	19,179	15,781	12,017

Table 5.2: Index size with dynamic insertion (# pages)

sequential file scan serves as a baseline. SHORE was configured to use 8KB page size, and in all experiments, the SHORE buffer pool size was set to 8MB (This small buffer pool size allows us to clearly see the effect of IOs). All experiments were performed on a machine with 2GHz Intel Xeon processor running Red Hat Linux version 2.4.20.

5.4.1 Dataset

For our experiments, we used both synthetic and real datasets. For the synthetic dataset, we used a dataset containing 2 million uniformly distributed points in 2, 4, and 8 dimensional data space (labeled as Uniform-2D, Uniform-4D, and Uniform-8D respectively).

In addition, we used the following two real datasets: the MAPS Catalog data containing photometric and astrometric information extracted from the Palomar Observatory Sky Survey and the Forest Cover data containing the forest cover type for 30 x 30 meter cells from US Forest Service Region 2 Resource Information System.

The MAPS dataset contains about 90 million objects with 39 attributes. To keep our experiments manageable, we used the first 2 million objects. To produce data with varying dimensionality, we used the first 2, 4, and 8 attributes from the original dataset to get 2, 4, and 8 dimensional datasets (MAPS-2D, MAPS-4D, MAPS-8D).

The Forest Cover dataset contains 581,014 entries with 54 attributes. Of these 54 attributes, there are 10 quantitative attributes. To measure the performance with varying dimensionality, we ran Principle Component Analysis (PCA) on the dataset over the 10 quantitative attributes to generate 2, 4, and 8 dimensional datasets (F.Cover-2D, F.Cover-4D, F.Cover-8D). The use of PCA reduces the dimensionality of data without a large loss of information, and this method is used to mimic the use of dimensionality reduction methods used in many high-dimensional applications.

For the range query workloads, we generated hypercube shaped range queries with varying query volumes. Query volume is defined as the percentage of the query hypercube volume over the data space volume. For the synthetic datasets, queries are uniformly randomly positioned in the underlying data space. For real data, we used skewed queries that follow the distribution of the underlying data.

In all experiments, for each point shown on the graph we have a corresponding query workload of 1024 queries and we report the average per query execution times and the average per query disk page accesses for range queries.

Our experiments consist of two main setups: experiments with static data and

Dataset	R*tree	Pyramid-T.	Quadtree
Uniform-2D	3	2.0	5.1
Uniform-4D	5.2	2.0	6.8
Uniform-8D	11.2	11.8	7.4
MAPS-2D	3.1	19.1	3.5
MAPS-4D	5.3	11.7	3.7
MAPS-8D	11.2	12.9	9.0

Table 5.3: Index building times with bulkloading (min)

Dataset	R*tree	Pyramid-T.	Quadtree
Uniform-2D	90.8	19.1	14.2
Uniform-4D	61.6	24.7	22.9
Uniform-8D	69.8	38.2	50.2
MAPS-2D	92.8	2.5	3.8
MAPS-4D	70.4	3.3	4.0
MAPS-8D	75.7	21.5	10.4

Table 5.4: Index building times with dynamic insertions (min)

dynamic data update. For each setup, index performance is evaluated using the range query. For the experiments with static and dynamic data, indices are created using bulkloading and dynamic index construction algorithms respectively.

In the interest of space, we only present experimental results with the synthetic dataset and the MAPS dataset in this paper. However, we note that the result with the Forest Cover dataset is very similar to the result with the MAPS dataset and these results do not change analysis we make in this paper.

5.4.2 Index size

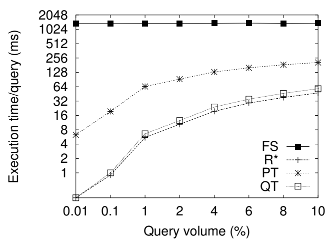
Table 5.1 and 5.2 show the sizes of indices using bulkloading and dynamic index construction algorithms respectively.

In Table 5.1, when indices are constructed using bulkloading algorithms, Quadtrees are about 6-36% larger than R*-trees. This is because Quadtrees have more non-leaf and leaf nodes than R*-trees and there is an extra storage overhead for saving meta-information for each node.

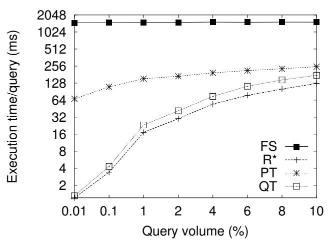
Table 5.2 shows index sizes when indices are constructed using dynamic algorithms. In this case, Quadtrees are larger than R*-trees by up to 21%. In addition to the larger number of non-leaf and leaf nodes, when indices are constructed using dynamic loading algorithms, the leaf node occupancy of Quadtrees is typically lower than that of R*-trees. For instance, the average leaf node occupancy in R*-trees is 73% while it is 72% in Quadtrees. In our current implementation, the minimum leaf node occupancy in Quadtrees is set to 70%. Leaf node occupancies in Quadtrees can be increased by using a higher minimum leaf node occupancy for packing, which reduces the index size and the query processing time, but incurs a larger index building times.

5.4.3 Index creation times

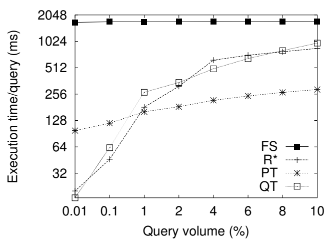
Table 5.3 and 5.4 show index creation times. As shown in Table 5.3, when indices are constructed using bulkloading, the performance differences amongst the three index structures is small. However, when the indices are constructed using dynamic loading algorithms, as shown in Table 5.4, the R*-tree is the slowest while the Pyramid-Technique and Quadtree construction times are comparable.



(a) Uniform-2D

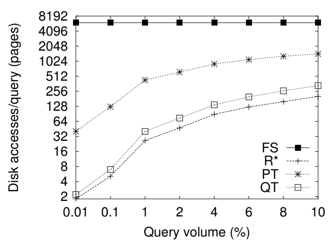


(b) Uniform-4D

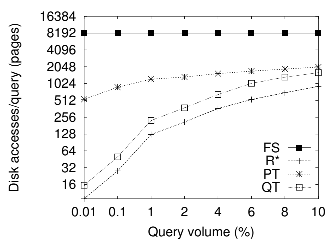


(c) Uniform-8D

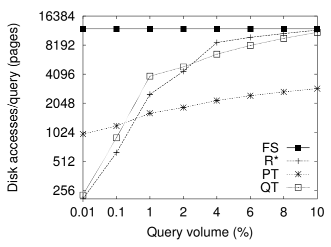
Figure 5.2: Range query with static data: Total execution time with uniform datasets



(a) Uniform-2D



(b) Uniform-4D



(c) Uniform-8D

Figure 5.3: Range query with static data: Number of disk page accesses with uniform datasets

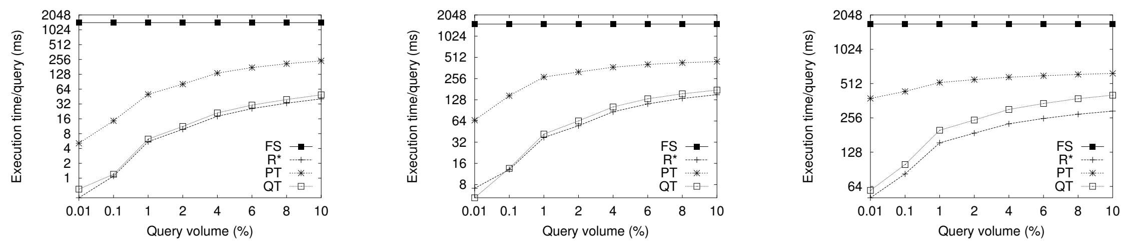
5.5 Experiments

5.5.1 Index evaluation with static data

In this section, we compare the performance of the indices with static data. Here, we assume that an entire dataset is available prior to an index construction and indices for the dataset are constructed using bulkloading algorithms.

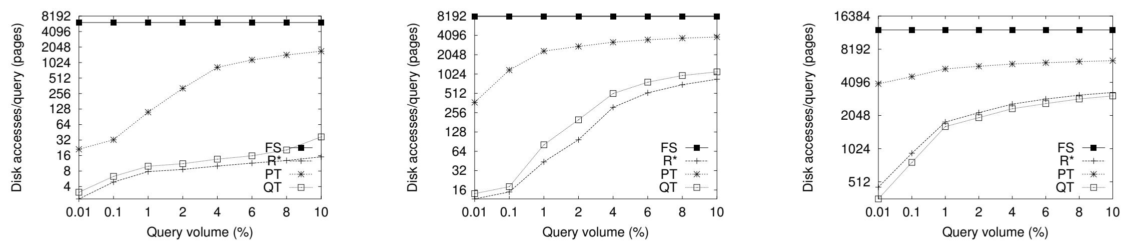
We first begin, by looking at the performance of the indices with uniform data with dimensionality varying from 2 to 8, and query volumes ranging from 0.01% to 10%.

Figure 5.2 shows the overall query execution times in this case, and Figure 5.3 shows the corresponding disk page accesses. From these figures we observe that the



(a) MAPS-2D (b) MAPS-4D (c) MAPS-8D

Figure 5.4: Range query with static data: Total execution time with MAPS datasets



(a) MAPS-2D (b) MAPS-4D (c) MAPS-8D

Figure 5.5: Range query with static data: Number of disk page accesses with MAPS datasets

query execution times are proportional to the number of disk accesses for all three indices. These results also indicate that the R*-tree shows slightly better performance than the Quadtree in most cases. The primary reason for the better performance of the R*-tree is due to the use of a STR bulkloading algorithm [61]. Although there can be a small degree of overlap amongst (the MBR of) the non-leaf nodes, this bulkloading algorithm produces a good index structure with almost no overlap amongst the leaf nodes. By sorting and partitioning the data by their spatial locations, the bulkloading algorithm significantly minimizes overlap among nodes, which results in reducing node accesses. In addition, the algorithm produces an index structure which is smaller in size than the Quadtree and the Pyramid-Technique. This leads to a smaller number of disk pages accesses than other index structures.

Regarding the performance of the Pyramid-Technique, it performs worse than the R*-tree and Quadtree with Uniform-2D and Uniform-4D. However, for the Uniform-8D dataset, although other indices still outperforms the Pyramid-Technique for small window queries, the Pyramid-Technique begins to outperform other indices for large window queries. In other words, with uniform data, the Pyramid-Technique does better than the R*-tree and Quadtree on high dimensional data with large window queries.

Figure 5.3 shows that with small window queries, the number of disk accesses with the Pyramid-Technique is significantly higher than that with the R*-tree and Quadtree. This figure also shows that the number of disk accesses incurred by the Pyramid-Technique increases more gradually compared to other indices. This is due to the *unbalanced* space split strategy used in the Pyramid-Technique. Especially, compared to the balanced space split used in the Quadtree, the unbalanced space split incurs more page accesses for small window queries, especially when window queries are further away from the center of data space. However, the unbalanced space split incurs less page accesses than the balanced space split for large window queries.

Although the Pyramid-Technique outperforms other indices in some cases with uniform datasets, with skewed data, the R*-tree and Quadtree outperform the Pyramid-Technique in all dimensions and also the speedup over the Pyramid-Technique is more significant with skewed data. These results are shown in Figures 5.4 and 5.5, which uses the skewed MAPS dataset. Note in this case, as before with the uniform dataset, the R*-tree outperforms the Quadtree by a small margin.

For the Pyramid-Technique, its unbalanced space split strategy can be adversarial with skewed data. Figure 5.6 illustrates this point with an example. This figure shows the Quadtree and the corresponding Pyramid-Technique index built on a skewed 2 dimensional dataset which is clustered in the bottom-left region. Each square in the Quadtree and each region in the Pyramid-Technique corresponds to a leaf page. A striped rectangle represents a query window, and gray regions are the ones affected by the window query. As shown in this figure, with a skewed query, there are more data regions that intersect with regions in the Pyramid-Technique compared to quads in the Quadtree, which in turns leads to a larger number of page accesses for the Pyramid-Technique.

To experimentally confirm the adversarial effect of the unbalanced space split for skewed data, we measured the *filtration ratio*, which is defined as the number of points that are *not* contained within the window query over the number of points retrieved in accessed pages. As more false hits are retrieved, the filtration ratio increases. With a 10% query volume and the MAPS-8D dataset, the filtration ratio of the Quadtree using a balanced space split strategy is 26.1%. For this same setting, the filtration ratio of the Pyramid Technique using the unbalanced space split strategy is 71.5%.

In summary, based on this experiment with uniform and skewed datasets, we make the following observations:

1. The R*-tree constructed using the STR bulkloading technique slightly outperforms the Quadtree in most cases.

2. The Pyramid-Technique outperforms the R*-tree and Quadtree with the high

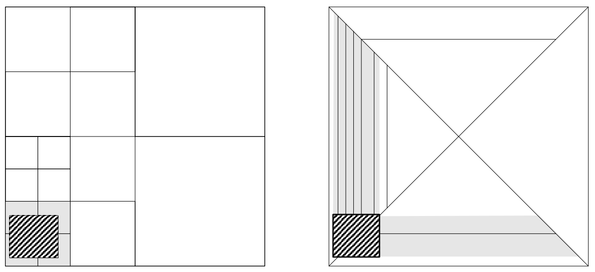

(a) Quadtree (b) Pyramid-Technique

Figure 5.6: Interaction between query and partitioning boundaries

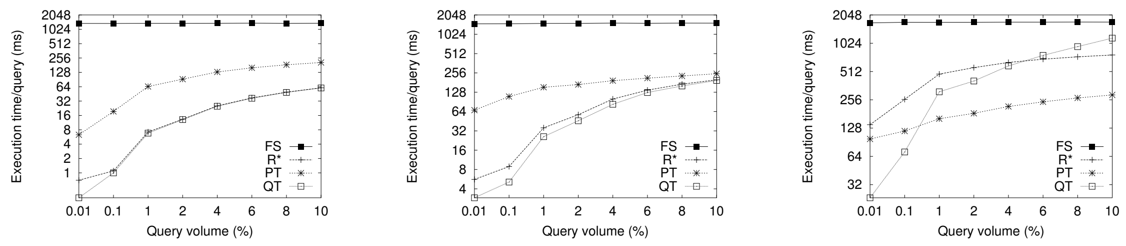
dimensional uniform data and large range queries.

3. Although the unbalanced space split in the Pyramid-Technique can be effective for high dimensional uniform data, it can be adversarial for skewed data.

5.5.2 Index evaluation with dynamic data

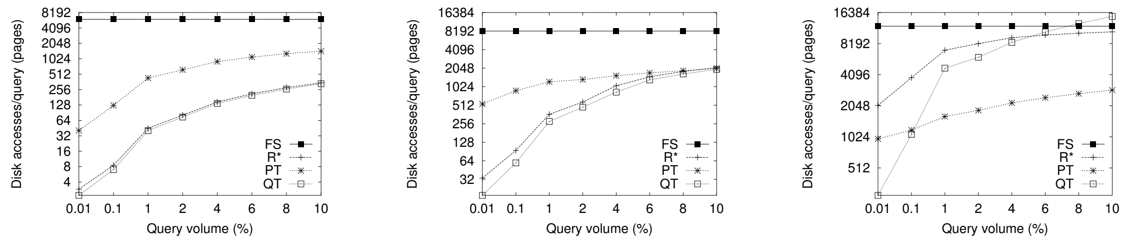
In this section, we compare the performance of indices with dynamic data. In contrast to the previous experiments with static data, here we assume that an entire dataset is not available at the index construction time and an index needs to be updated whenever data is available. In this case, dynamic algorithms, which insert one data entry at a time, have to be used. In other words, all index structures used in this section are built with dynamic index construction algorithms.

We first begin, by looking at the performance of the indices with uniform data. Figures 5.7 and 5.8 present average query execution times and disk accesses for query volumes ranging from 0.01% to 10% for 2D, 4D, and 8D uniform datasets. Overall, with 2 dimensional uniform datasets, the Quadtree and R*-tree have comparable performance although the number of disk accesses for the Quadtree is slightly less than



(a) Uniform-2D (b) Uniform-4D (c) Uniform-8D

Figure 5.7: Range query with dynamic data: Total execution time with uniform datasets



(a) Uniform-2D (b) Uniform-4D (c) Uniform-8D

Figure 5.8: Range query with dynamic data: Number of disk page accesses with uniform datasets

that of the R*-tree. In addition, both indices outperform the Pyramid-Technique. With 4 and 8 dimensional uniform datasets, the Quadtree slightly outperforms the R*-tree in most cases (except with large query volume queries in 8 dimensional data).

We note that, although the *bulkloaded* R*-tree outperforms the *bulkloaded* Quadtree as shown in Section 5.5.1, the *dynamic* R*-tree shows worse performance than the *dynamic* Quadtree. This is because the structure of the R*-tree is different depending on whether the R*-tree is constructed using a bulkloading algorithm or a dynamic algorithm. The structural difference in indices has an impact on the query processing performance. However, except for the index space utilization, the structure of the Quadtree and Pyramid-Technique remains the same regardless of the index construc-

Table 5.5: Relative increase on range query execution times

	R*-tree	**Quadtree**
Uniform-4D	2.07	1.12
Uniform-8D	2.65	1.51
MAPS-4D	1.25	1.04
MAPS-8D	1.31	1.04

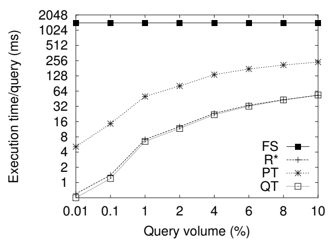
The numbers in this table represent range query execution time ratios. A range query execution time ratio is calculated as a range query execution time with dynamically loaded indices over bulkloaded indices. Range query execution times with 1% query volumes are used.

tion method. Consequently, their query processing performance is less affected by index construction methods. For example, as shown in Table 5.5.2, when we measure the relative increase in the query execution times with dynamic indices over bulkloaded indices, the performance of R*-trees degrades by up to 2.7 times while the performance change of the Quadtree is relatively smaller than the R*-trees.

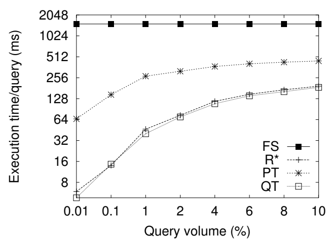
The main reason for the degraded performance of the dynamic R*-tree is due to the increased overlap amongst MBRs. While there is almost no MBR overlap in the bulkloaded R*-tree, there is an increased overlap amongst MBRs in the dynamic R*-tree, and the overlap amongst MBRs become even more significant with increasing dimensionality. To examine the structural difference between the bulkloaded and dynamic R*-tree indices, we measured the degree of MBR overlap in each index using the following formula:

$$overlap(l) = \frac{\sum_{i=1}^{n} Area(MBR_{i,l})}{Area(\bigcup_{i=1}^{n} MBR_{i,l})} \tag{V.1}$$

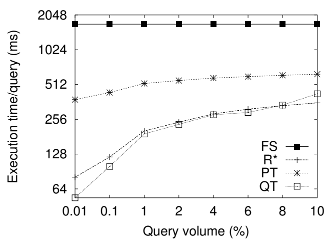
where *overlap*(l) is the average number of MBRs at a level l that overlap with a



(a) MAPS-2D

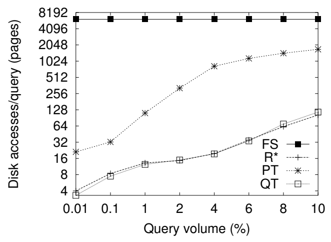


(b) MAPS-4D

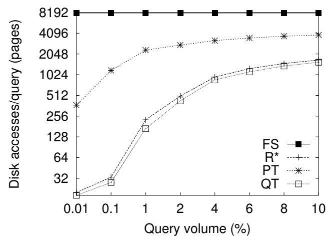


(c) MAPS-8D

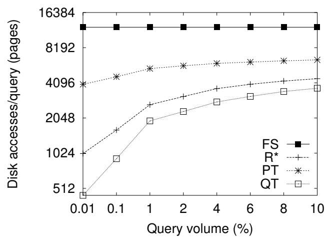
Figure 5.9: Range query with dynamic data: Total execution time with MAPS datasets



(a) MAPS-2D



(b) MAPS-4D



(c) MAPS-8D

Figure 5.10: Range query with dynamic data: Number of disk page accesses with MAPS datasets

random point in the data space of the index, $MBR_{i,l}$ is the i-th MBR at a level l, n is the total number of MBRs at level l, and $\bigcup_{i=1}^{n} MBR_{i,l}$ is the minimum bounding polygon that encompasses all $MBR_{i,l}$ at a level l.

The index structure differences measured with respect to MBR overlap is shown in Table 6.4. The table shows the overlap of the leaf-level MBRs in the bulkloaded and dynamically constructed indices. As shown in Table 6.4, the MBR overlap in the dynamic R*-tree is larger than that of the bulkloaded R*-tree especially with an increasing data dimensionality. The increased overlaps amongst MBRs in the dynamic R*-tree increases the degree of multiple path traversals during query processing, thus leads to increasing query processing times.

Table 5.6: Overlap in bulkloaded and dynamic R*-trees

Dataset	Bulkloaded	Dynamically loaded
Uniform-2D	1.0	1.7
Uniform-4D	1.1	6.4
Uniform-8D	1.2	77.9

The numbers in this table represent the MBR overlap of the leaf-level in the bulk-loaded and dynamic R*-tree.

So far, we have discussed the performance of dynamically constructed indices with uniform datasets. Figures 5.9 and 5.10 present average query execution times and disk accesses for skewed MAPS datasets. Similar to the result with uniform data, with skewed data, the Quadtree outperforms other indices.

In summary, based on this experiment, we make the following observations:

1. When indices are constructed using dynamic algorithms, the Quadtree outperforms the R*-tree, although the performance of these two index structures is comparable in 2 dimension.

2. Index construction methods have an impact on the structure and performance of the R*-tree, while index construction methods does not change the structure and performance of the Quadtree and Pyramid-Technique significantly.

5.6 Conclusions

In this chapter, we have compared the performance of the R*-tree, the Pyramid-Technique, and the Quadtree for indexing low to medium dimensional datasets under a variety of settings. Although the R*-tree is widely used for indexing such datasets, our results show that the Quadtree outperforms the R*-tree and the Pyramid-Technique in many cases. Our experimental results also reveal several important points: 1) The query processing performance of the R*-tree is affected by the index construction algorithms, while the Quadtree and Pyramid-Technique are relatively less affected by the index construction method. 2) Although the Quadtree is often ignored for indexing multi-dimensional data due to its low space utilization, the packing technique in our Quadtree can lead to a more efficient space utilization, improving the overall Quadtree performance.

With the increasing importance of multi-dimensional point indexing methods in a variety of data-centric applications, the result in this experimental study provides a useful guideline in selecting a right multi-dimensional point indexing method.

CHAPTER VI

Evaluation of Multi-dimensional Indexing Structures for kNN and Distance Join Queries

6.1 Introduction

In previous chapter, we compared multi-dimensional index structures using range query operations. In this chaper, we present an experimental study comparing the R*-tree and Quadtree for kNN and distance join query operations. Although a variety of query operations can be performed using these index structures, previous work has largely focused only on the range search operation. We go beyond this previous work and compare the performance of these index structures using k-nearest neighbor (kNN) and distance join queries. Similar to the previous chapter, we also compare multi-dimensional index structure with different index construction methods.

There have been several experimental studies comparing the R-trees and Quadtrees. Kothuri *et. al* [57] compared the R-tree and Quadtree using a variety of range queries on 2 dimensional Geographical Information Systems (GIS) spatial data, and showed

	Dynamic Insertion	**Bulkloading**
kNN	Quadtree	R*-tree
k Distance Join	Quadtree	R*-tree

Table 6.1: Index Evaluation Summary

that in general the R-tree outperforms the Quadtree. Hoel and Samet [45] have compared various R-tree variants (R-tree, R*-tree, and R+-tree) and the Quadtree for the traditional spatial overlap join operation. They showed that the R+-tree and Quadtree outperform the R-tree and R*-tree using 2 dimensional GIS spatial data. Although a variety of operations can be performed on a tree-based multi-dimensional indices, to the best of our knowledge, besides the aforementioned studies, there are no similar studies comparing the performance of the R*-tree and Quadtree using other types of query operations. In addition, point data with higher dimensionality, which has applications beyond traditional spatial applications, was not considered in either of the above works. Therefore, using the kNN and distance join query operations, we evaluate the performance of index structures in the context of multi-dimensional point data.

Our experiments reveal several interesting points. Table 6.1 shows the best performing index structure for a given query operation and index construction method. As shown in Table 6.1, when data is static (when a bulkloading algorithm is used for an index construction) and kNN/distance join operation is performed, the R*-tree shows the best performance. However, when data is dynamic (i.e., there are frequent updates), the Quadtree begins to outperform the R*-tree. The primary reason for the better performance of the R*-tree with static data is due to the use of a STR bulkloading algorithm [61]. However, once the dynamic R*-tree algorithm [12] is used, overlap amongst MBRs increase with increasing data dimensionality, and the R*-tree performance degrades.

The rest of this chapter is organized as follows. Section 6.2 describes the implementation details of query processing algorithms. Section 6.3 describes our experimental setup. Section 6.4 presents our experimental results. Section 6.6 summarizes our results.

6.2 Implementation

In this section, we describe our implementation of kNN and k distance join algorithms that use these R*-trees and Quadtrees.

6.2.1 kNN Query Processing

Given an index I, a query point p, and a value k, the kNN query returns k points $\in I$ which are closest to the query point p based on a distance function. In our implementation, the Euclidean distance function is used. We implemented the kNN query processing for the R*-tree and Quadtree based on the algorithm described in [41]. In [41], the algorithm is implemented using the R-tree. However, the algorithm is general enough to be adapted to any hierarchical spatial index structures such as the R*-tree and Quadtree.

To find k-nearest neighbors for a query point p, the algorithm traverses a given index I using a *best-first* traversal technique. With this technique, the next node to

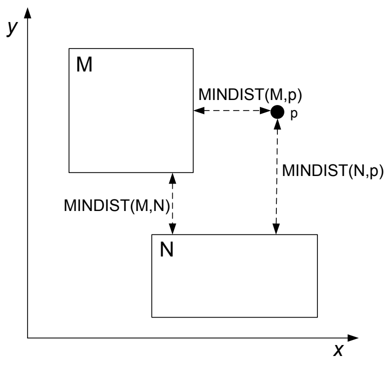


Figure 6.1: MINDIST example

traverse is the node with the least distance from the query point p. Note that a node in this algorithm is a non-leaf, leaf node, or actual object (point in this case). Also, the distance is a *minimum* distance, MINDIST, which is a lower bound estimation between a node and a query point p. Figure 6.1 shows an example of MINDIST calculation in 2 dimensional space.

To find a node with the least distance, the algorithm maintains a priority queue consisting of nodes, sorted based on their distances from the query point p. When the next node is retrieved from the priority queue, if the node is a non-leaf or a leaf node, it is expanded by pushing its child nodes into the priority queue. If the retrieved node is a point, the point is reported as the next nearest point to p. This process is repeated until k points are reported.

6.2.2 k Distance Join Query Processing

Given two indices, I_A and I_B, and a value k, k distance join returns a set of k pairs, (p_A, p_B), such that $p_A \in I_A$, $p_B \in I_B$, and the distances of the k pairs are the smallest among all possible pairs between I_A and I_B. In our experiment, we used the distance join algorithm described in [40]. This is a general algorithm for finding both an unlimited number or a specified k nearest pairs in order. In this work, we focus on k distance join which finds exactly k nearest pairs.

The distance join algorithm takes two indices, I_A and I_B, as inputs and maintains a priority queue containing item pairs, (i_a, i_b), with one item from each of I_A and I_B. Each item in the pair is either a node or an object, and therefore there are four types of item pairs containing node/node, node/object, object/node, or object/object pairs. The priority queue is sorted based on *minimum* distance between i_a and i_b. Each item pair in the priority queue is evaluated based on the best first traversal (The least distance item pair is evaluated first). When processing an item pair, (i_a, i_b), if both i_a and i_b are objects, it is reported as the next nearest pair. Otherwise, one of the items which is a node, for example, i_a, is expanded in the following way: item pairs, (i'_a, i_b), are produced where i'_a is a child node of i_a and only item pairs whose distance is smaller than an estimated maximum distance of k nearest pairs, d_{max}, are inserted into the priority queue. Given that the algorithm only needs to return k nearest pairs, d_{max} is estimated based on item pairs in the priority queue. Essentially, d_{max} is used to prune unnecessary item pairs that cannot be part of final k nearest pairs.

In computing k distance join, the original algorithm suffers from a large priority queue especially with item pairs inserted during the early stage of distance join processing. Basically, the algorithm estimates an upper bound distance for k nearest pairs, d_{max}, and inserts only item pairs whose distances $\leq d_{max}$ into the priority queue. Initially, d_{max} is set to an infinity, and d_{max} decreases as the algorithm proceeds. However, at the beginning, d_{max} is usually large, so many item pairs are inserted into the priority queue. However, note that many of those item pairs inserted early, especially, object/object pairs, are unlikely to be part of k nearest pairs at the end, and such pairs just remain in the priority queue, increasing the size of the queue and incurring overhead for queue operations.

Improvements over the incremental distance join algorithm: Based on the above observation, we improved the existing algorithm in the following way: Instead of keep four types of item pais, our implementation keeps three types of item pairs, node/node, node/object, and object/node pairs, in the main priority queue, and object/object pairs in a separate queue called the result queue. Since we are only interested in k nearest object/object pairs, the size of the result queue is fixed at k, allowing only k nearest object pairs found so far. This modification has the following effects: Compared to the existing algorithm, all unnecessary object/object pairs are removed from the priority queue, keeping only necessary k object/object pairs in the result queue. Consequently, the size of the priority queue and overhead for queue operations is reduced.

The existing algorithm also maintains an auxiliary priority-queue like data structure to estimate d_{max}. The size of this data structure is bounded by k, but any item pairs inserted into the main priority queue is also inserted into this data structure. Details on estimating d_{max} using this data structure is referred to [40]. Although the d_{max} estimation based on the information on this data structure can be useful, it requires managing an extra data structure and queue operations. In our implementation, we simply prune item pairs based on the maximum k nearest pair distance in the result queue and remove the overhead of maintaining this extra data structure.

Another improvement over the existing algorithm is dynamic priority queue management. With a large dataset or a large k, the priority queue can grow quickly and can become larger than the available main memory. To solve this problem, the existing algorithm proposed a hybrid memory/disk scheme which partitions a priority queue based on distances of item pairs [40]. Pairs with a distance $\leq D$ is stored in an in-memory heap structure, otherwise in an unsorted in-memory list or a file on disk. However, the main problem with this scheme is that it requires users specifying the distance cutoff, D, which is very sensitive to input data distributions. In other words, a wrong choice of D leads to very poor performance(No methods for picking the right D value is provided in [40]). We adopt a similar strategy, but propose a scheme that dynamically decides D simply based on an available memory size. Given a limit on the memory available for storing the heap, the maximum number of nodes that can be inserted into the heap, is calculated. Once the heap is full, half of the nodes in the heap, with distances larger than the other half is offloaded to a disk file. Then a distance cutoff, D, is dynamically set to the maximum distance of the item pairs left in the heap. Whenever an item pair needs to be inserted into the queue, if the distance of that item pair is $\leq D$, then it is inserted into the heap. Otherwise, it is written to the disk file. When the priority queue is empty, item pairs in the disk file are sorted by their distances and the top $0.5t$ item pairs with shorter distances than others are moved into the in-memory heap. A new D value is chosen based on the maximum distance of item pairs in the heap. The benefit of this scheme is that it doesn't require a manual configuration and a fine tuning of D. The value of D is dynamically decided during the distance join.

6.3 Experimental Setup

In this section, we present experimental results comparing index-based kNN and k distance join queries for multi-dimensional data. Three types of indices, R*-tree, STR R-tree, and Quadtree, are implemented in SHORE. In the following sections, the bulkloaded R*-tree represents the STR R-tree unless specified.

In the following experiments, for each point shown on the graph, we report the average per query disk page accesses since disk I/O is a dominant factor for determining overall execution times and overall execution times are proportional to disk page accesses. Other performance measures such as the average per query execution times and the average per query distance calculation are also presented whenever needed.

6.3.1 Datasets

In this section, we describe the datasets used in the experiments.

kNN query: We used both synthetic and real data. For synthetic data, we used a uniform dataset containing uniformly distributed points in 2 to 8 dimensional space (Uniform-2D, Uniform-4D, and Uniform-8D). For real data, we used the MAPS Catalog data containing photometric and astrometric information extracted from the Palomar Observatory Sky Survey (`http://iparrizar.mnstate.edu/~juan/MAPS_Database/`). The MAPS data contains about 90 million objects with 39 attributes. To keep our experiments manageable, we used the first 2 million objects. To produce data with varying dimensionality, we used the first 2, 4, and 8 attributes from the original dataset to get 2, 4, and 8 dimensional datasets (MAPS-2D, MAPS-4D, MAPS-8D).

The kNN query workload was generated by using point queries that follow the distribution of the underlying data.

k distance join query: From the above datasets, we generated datasets for distance join queries. The above datasets containing 2 million points are divided into smaller datasets containing 0.5 million points each. Two of the 0.5 million point datasets (Uniform-2D-A, Uniform-2D-B, Uniform-4D-A, Uniform-4D-B, Uniform-8D-A, and Uniform-8D-B) are used as inputs for distance joins. For experiments with the k distance join queries, we used the smaller datasets to keep our experiments manageable. A distance join query is much more expensive and takes longer than a nearest neighbor query. A distance join query processes two datasets and the number of distance computations and disk I/Os incurred are proportional to $N * M$, where N and M are the number of entries in two input datasets. On the other hand, a kNN query processes a single dataset, and the number of distance computations and disk I/Os are proportional to N, where N is the number of entries in the dataset.

SHORE was configured to use 8KB page size. For experiments with kNN queries, the SHORE buffer pool size was set to 8MB, 16MB, and 32MB, for 2D, 4D, and 8D datasets, each containing 2 million points. For experiments with k distance join queries, the SHORE buffer pool size was set to 4MB, 8MB, and 16MB, for two 2D, 4D, and 8D datsets, each containing 0.5 million points. The buffer sizes are approximately proportional to 1/4 of the corresponding index sizes. This small buffer pool size allows us to clearly see the effect of IOs. For each index structure, data objects (points in

Table 6.2: Input index sizes for kNN queries(# pages)

	Bulkload		Dynamic Insertion	
Dataset	R*tree	Quadtree	R*tree	Quadtree
Uniform-2D	3,041	4,124	4,200	4,124
Uniform-4D	5,220	6,227	7,269	8,932
Uniform-8D	11,790	10,889	12,738	15,646
MAPS-2D	3,041	3,446	4,259	5,258
MAPS-4D	5,220	5,531	7,217	8,084
MAPS-8D	11,790	12,556	13,210	19,467

Table 6.3: Input index sizes for k distance join queries(# pages)

	Bulkload		Dynamic Insertion	
Dataset	R*tree	Quadtree	R*tree	Quadtree
Uniform-2D-A	762	1,031	1,051	1,031
Uniform-2D-B	762	1,031	1,045	1,031
Uniform-4D-A	1,307	1,382	1,869	1,710
Uniform-4D-B	1,307	1,381	1,828	1,721
Uniform-8D-A	2,949	3,877	3,182	4,199
Uniform-8D-B	2,949	3,878	3,242	4,546
MAPS-2D-A	762	841	1,048	1,351
MAPS-2D-B	762	842	1,038	1,336
MAPS-4D-A	1,307	1,358	1,790	2,083
MAPS-4D-B	1,307	1,357	1,788	2,086
MAPS-8D-A	2,949	3,159	3,146	4,857
MAPS-8D-B	2,949	3,155	3,164	5,101

this case) are directly stored in leaf pages. All experiments were performed on a machine with a 2.2 GHz AMD Opteron processor running Red Hat Linux version 2.6.12.

6.3.2 Index size

This section describes details about the indices that are used as inputs for kNN queries and distance joins. Table 6.2 and 6.3 show the sizes of both indices constructed using bulkloading and dynamic index construction algorithms for the kNN and distance join experiments.

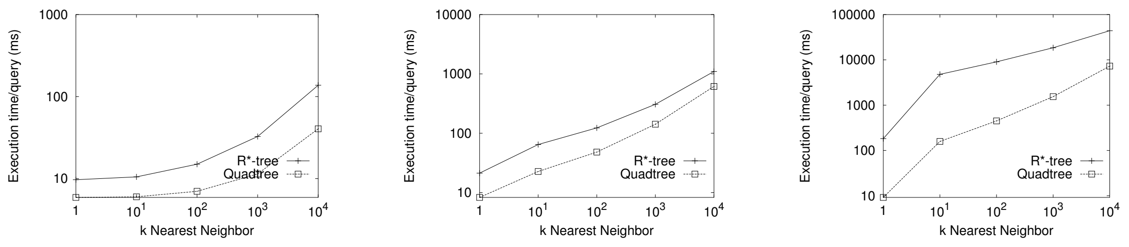
In Table 6.2, when indices are constructed using bulkloading algorithms, Quadtrees are up to 36% larger than R*-trees. This is because Quadtrees have more non-leaf and leaf nodes than R*-trees and there is an extra storage overhead for saving meta-information for each node. We note that leaf node occupancies in all indices are 100% when bulkloading algorithms are used.

When indices are constructed using dynamic algorithms, Quadtrees are larger than R*-trees by up to 47%. In addition to the larger number of non-leaf and leaf nodes, when indices are constructed using dynamic loading algorithms, the leaf node occupancy of Quadtrees is typically lower than that of R*-trees. For instance, the average leaf node occupancy in R*-trees is 74% while it is 64% in Quadtrees. In our current implementation, the minimum leaf node occupancy in Quadtrees for initiating an index packing is set to 50%. Leaf node occupancies in Quadtrees can be increased by using a higher minimum leaf node occupancy for packing. This reduces index sizes and query processing times, but incurs larger index building times.

6.4 Experiments

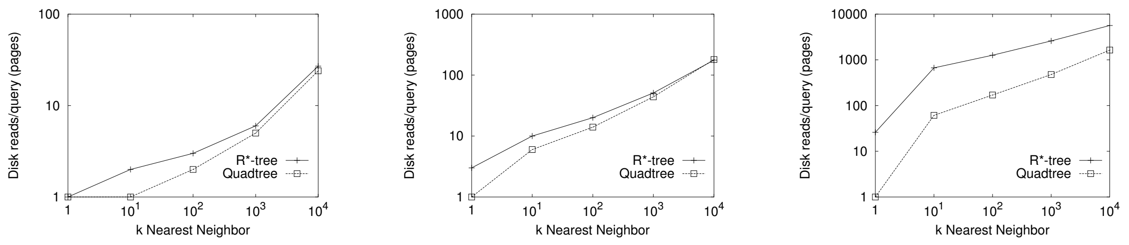
Our experimental evaluation consists of two major parts. First, in Section 6.4.1, we compare the performance of indexing structures using kNN queries. Second, in Section 6.4.2, we compare the performance of indexing structures using k distance join queries. In each experiment, we also investigate the index performance associated with index construction methods.

6.4.1 kNN Query



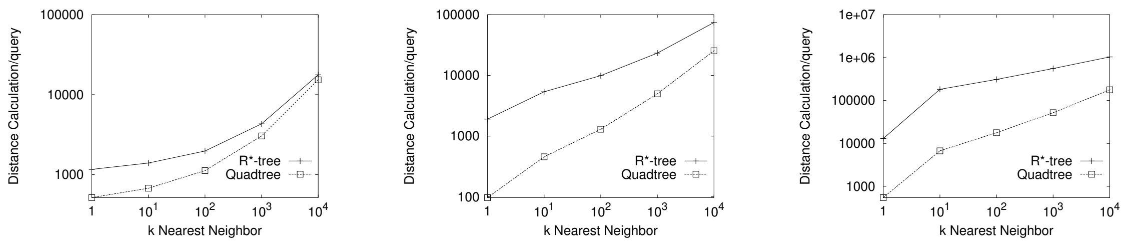
(a) Uniform-2D (b) Uniform-4D (c) Uniform-8D

Figure 6.2: kNN query with dynamically constructed indices: Total execution time with uniform datasets



(a) Uniform-2D (b) Uniform-4D (c) Uniform-8D

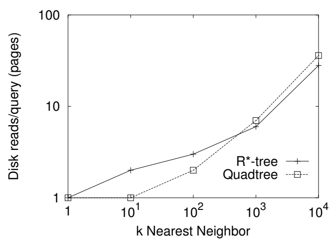
Figure 6.3: kNN query with dynamically constructed indices: Number of disk page accesses with uniform datasets



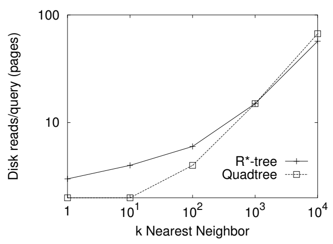
(a) Uniform-2D (b) Uniform-4D (c) Uniform-8D

Figure 6.4: kNN query with dynamically constructed indices: Number of distance calculation with uniform datasets

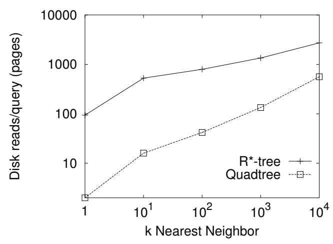
In this section, we compare the performance of the indices using the kNN query operation. This section consists of two main experiments: the first experiment uses



(a) MAPS-2D

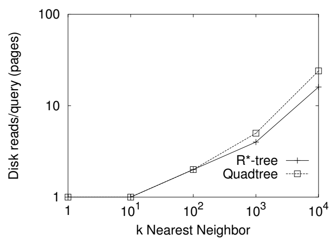


(b) MAPS-4D

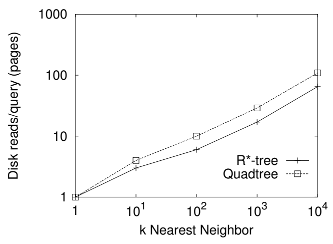


(c) MAPS-8D

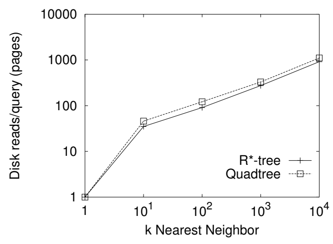
Figure 6.5: kNN with dynamically constructed indices: Number of disk page accesses with MAPS datasets



(a) Uniform-2D

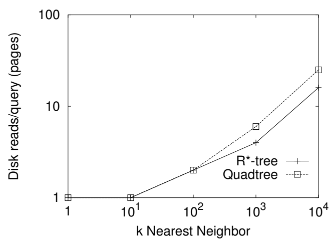


(b) Uniform-4D

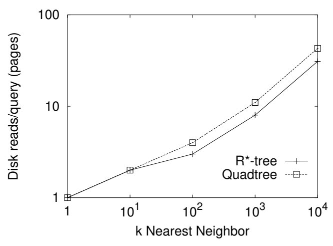


(c) Uniform-8D

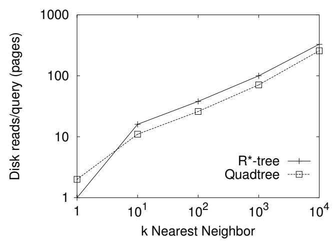
Figure 6.6: kNN with bulkloaded indices: Number of disk page accesses with uniform datasets



(a) MAPS-2D



(b) MAPS-4D



(c) MAPS-8D

Figure 6.7: kNN with bulkloaded indices: Number of disk page accesses with MAPS datasets

indices constructed using dynamic algorithms, and the second experiment uses indices constructed using bulkloading algorithms.

6.4.1.1 Comparison with dynamically constructed indices

We first begin by looking at the performance of the indices with uniform data. Figures 6.2, 6.3, and 6.4 present average query execution times, disk accesses, and number of distance calculations for uniform datasets. From these figures we observe that query execution times are proportional to the number of disk accesses and number of distance computations. However, we note that the number of disk accesses is a dominant factor that determines total execution times (CPU times typically contribute to only less than 5% of total execution times). These figures show that the Quadtree outperforms the R*-tree with uniform datasets by 1.7-3.4X, 1.8-2.8X, and 6.0-20.1X for 2D, 4D, and 8D uniform datasets. It also shows that the performance of the Quadtree over the R*-tree increases significantly with increasing data dimensionality.

Figure 6.5 shows the average number of disk accesses with skewed data (For brevity, we only show the number of disk accesses with skewed data since total execution times are roughly proportional to the number of disk accesses). Similar to the result with uniform data, with skewed data the Quadtree outperforms the R*-tree. The only difference is that the relative speedup of the Quadtree over the R*-tree is slightly higher with 8D skewed data than that with uniform data. For instance, the Quadtree outperforms the R*-tree with MAPS datasets by 0.8-1.4X, 0.9-2.5X, and 7.5-45.7X for 2D, 4D, and 8D MAPS datasets.

6.4.1.2 Comparison with bulkloaded indices

Figure 6.6 and 6.7 show the average number of disk accesses for uniform and MAPS datasets when bulkloaded indices are used for kNN query processing. Unlike previous results, when bulkloaded indices are used as underlying index structures, overall the R*-tree outperforms the Quadtree. For instance, with 2D, 4D, and 8D uniform datasets, in terms of total execution times, the R*-tree outperforms the Quadtree by 1.1-1.2X, 1.1-1.2X, 1.1-1.4X. With 2D, 4D, and 8D MAPS datasets, the R*-tree outperforms the Quadtree by 1.0-1.2X, 1.2-1.3X, and 0.6-1.1X. However, there is a case where the Quadtree outperforms the R*-tree. For example, the Quadtree outperforms the R*-tree with a 8D MAPS dataset.

6.4.1.3 Analysis

In Section 6.4.1.1, the result shows that the dynamic Quadtree outperforms the dynamic R*-tree, and the Quadtree speedup over the R*-tree increases with increasing dimensionality. The better performance of the Quadtree is primarily due to the overlap-free and regular space decomposition partitioning used in the Quadtree. In general, the performance of the kNN query algorithm depends on how closely distances among points are estimated using MBRs. MBRs of overlap-free and regular shapes are more advantageous than large, irregular, and overlapped MBRs for distance estimation, and accurate distance estimation eventually reduces the total number of distance calculations and disk page accesses.

More specifically, in finding k nearest neighbors, the number of distance calculations and disk page accesses are mainly determined by how MINDIST is close to actual distances between a query point and points in MBRs. (As discussed Section 6.2.1, MINDIST is a lower bound distance between a query point and points in a given MBR.) For the distance estimation using MINDIST, regular and small MBR shapes in the Quadtree are more suitable for estimating distances between points than irregular and large MBRs in the R*-tree. Furthermore, with skewed data, the shape of MBRs is likely to be highly unbalanced in the R*-tree so that MINDIST deviates significantly from actual distances. This also explains why the relative speedup of the Quadtree over other indices is improved with skewed data.

The main reason for poor performance of the dynamic R*-tree with increasing dimensionality is due to the increased overlap amongst MBRs (shown in Table 6.4). While there is almost no MBR overlap in the Quadtree, there is significant overlap amongst MBRs in the dynamic R*-tree, and the overlap amongst MBRs becomes even more significant with increasing dimensionality.

Although the *dynamic* R*-tree performs worse than the *dynamic* Quadtree, the result in Section 6.4.1.2 shows that the *bulkloaded* R*-tree shows better performance than the *bulkloaded* Quadtree. This is because the structure of the R*-tree is significantly different depending on whether the R*-tree is constructed using a bulkloading algorithm or a dynamic algorithm. The structural difference in indices has a dramatic impact on the query processing performance. However, except for the index space utilization, the structure of the Quadtree remains the same regardless of the index construction method. Consequently, query processing performance with the Quadtrees is less affected by index construction methods.

The above observation is more clearly shown in Table 6.5. Table 6.5 presents two major performance measures affecting total execution times with 4D uniform datasets: number of disk page accesses and distance computations. Regardless of the underlying index construction methods, with Quadtrees the number of distance computations remain the same. The number of disk page accesses is increased by 1.1-1.7X with the dynamic Quadtree due to the increased index size. This leads to increasing the total execution times by 1.0-1.1X with the dynamic Quadtree.

In contrast, index structures of the dynamic R*-tree and bulkloaded STR R-tree can be significantly different and these differences directly lead to substantially different query operation performance. As shown in Table 6.5, compared to the bulkloaded R*-tree, the number of distance computations and disk accesses is increased by 1.1-1.7X and 1.7-2.4X respectively. This results in increasing the total execution times by 1.7-2.9X with 4D uniform datasets. We note that the the performance difference between the dynamic R*-tree and bulkloaded STR R-tree will be more significant with increasing dimensionality.

To be clear about the structural difference in the bulkloaded STR R-tree and dynamic R*-tree, we measured the degree of MBR overlaps in R*-trees. To measure the degree of MBR overlap in each index, we used the following formula:

$$overlap(l) = \frac{\sum_{i=1}^{n} Area(MBR_{i,l})}{Area(\bigcup_{i=1}^{n} MBR_{i,l})} \tag{VI.1}$$

where *overlap*(l) is the average number of MBRs at a level l that overlap with a random point in the data space of the index, $MBR_{i,l}$ is the i-th MBR at a level l, n

Table 6.4: Overlap in bulkloaded and dynamic R*-trees with a 2 million point uniform dataset

Dataset	Dynamic Insertion	Bulkload
Uniform-2D	1.7	1.0
Uniform-4D	6.4	1.1
Uniform-8D	77.9	1.2

The numbers in this table represent the MBR overlap of the leaf-level in the bulkloaded and dynamic R*-tree.

is the total number of MBRs at level l, and $\bigcup_{i=1}^{n} MBR_{i,l}$ is the minimum bounding polygon that encompasses all $MBR_{i,l}$ at a level l.

The MBR overlap in the dynamic R*-tree measured using the above formula is shown in Table 6.4. It indicates that the MBR overlap in the dynamic R*-tree is significantly larger than that of the bulkloaded Quadtree, especially with an increasing data dimensionality. The increased overlaps amongst MBRs in the dynamic R*-tree increases the degree of multiple path traversals during query processing, thus leading to increasing query processing times.

In summary, based on the above experiment, we make the following observations:

1. The regular and overlap-free shape of MBRs in the Quadtree is more suitable for a kNN query processing than the irregular and overlapping MBRs in the R*-tree. This property of the Quadtree eventually results in a smaller number of distance calculations and disk accesses compared to the R*-tree.

2. Index construction methods have a significant impact on kNN query processing, especially with the R*-tree. When indices are constructed using dynamic algorithms, the Quadtree outperforms the R*-tree. When indices are constructed

Table 6.5: Performance comparison of the bulkloaded and dynamically constructed indices using Uniform-4D datasets

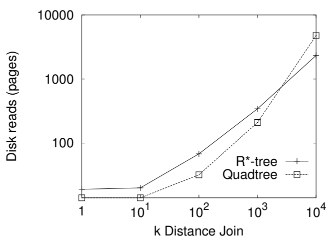
R*-tree			
Dynamic Insertion		Bulkload	
Disk reads	distance comp.	Disk reads	distance comp.
3,047	1,950,642	967	877,417
10,368	5,506,541	3,060	2,293,736
20,400	10,160,252	6,112	4,451,380
51,754	23,883,840	17,676	12,280,251
180,426	76,241,782	66,804	45,329,044

Quadtree			
Dynamic Insertion		Bulkload	
Disk reads	distance comp.	Disk reads	distance comp.
1,328	99,163	1,201	99,163
5,794	466,652	4,409	466,652
13,938	1,327,763	9,881	1,327,763
45,008	5,093,369	29,641	5,093,369
184,522	26,088,757	111,423	26,088,757

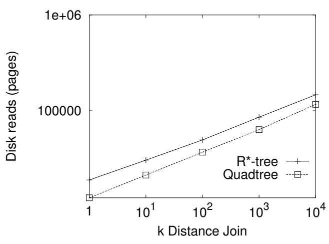
using bulkloading algorithms, the R*-tree slightly outperforms the Quadtree.

6.4.2 k Distance Join Query

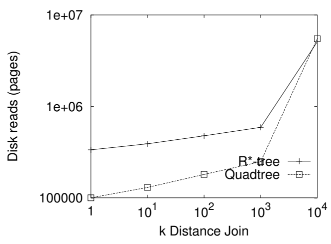
In this section, we compare the performance of the indices using k distance join queries. Similar to the kNN query experiments, this section also consists of two main experiments: the first experiment uses indices constructed using dynamic algorithms, and the second experiment uses indices constructed using bulkloading algorithms. In the following experiments, for brevity, we only report total number of disk page accesses. The total execution times are omitted as they are proportional to the the number of disk accesses.



(a) Uniform-2D

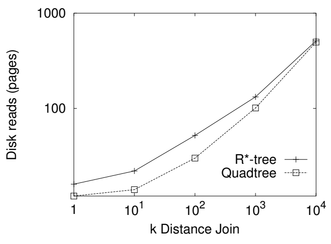


(b) Uniform-4D

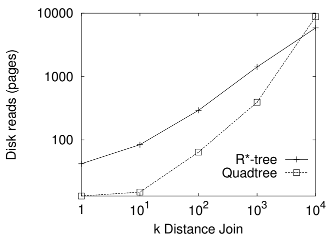


(c) Uniform-8D

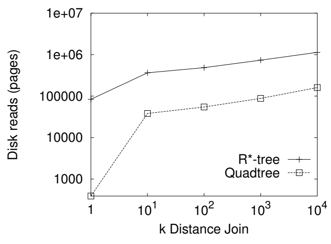
Figure 6.8: Distance join query with dynamically constructed indices: Number of disk page accesses with uniform datasets



(a) MAPS-2D



(b) MAPS-4D



(c) MAPS-8D

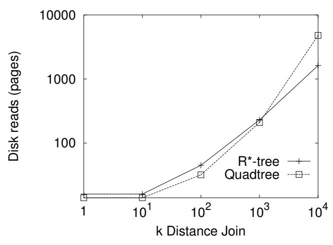
Figure 6.9: Distance join query with dynamically constructed indices: Number of disk page accesses with MAPS datasets

6.4.2.1 Comparison with dynamically constructed indices

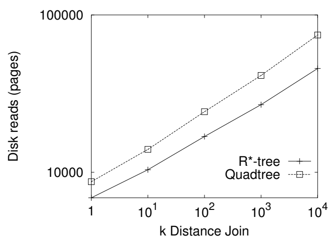
We first begin by looking at the performance of the indices with uniform data. Figure 6.8 presents the number of disk accesses for uniform datasets.

This figure shows that overall the Quadtree incurs fewer disk accesses than the R*-tree for uniform datasets. However, we note that there are cases where the Quadtree incurs more disk accesses than the R*-tree. For example, with a large k value, the Quadtree can incur more disk accesses than the R*-tree. For example, with $k = 10000$, the Quadtree incurs more disk page accesses than the R*-tree with 2D and 8D uniform datasets.

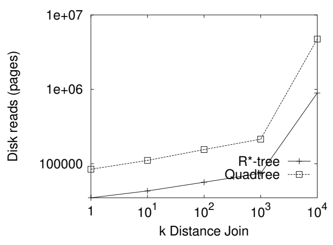
In general, fewer disk accesses lead to better execution times: the Quadtree outper-



(a) Uniform-2D

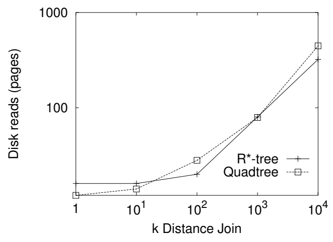


(b) Uniform-4D

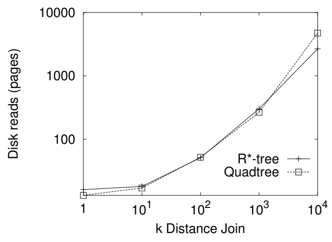


(c) Uniform-8D

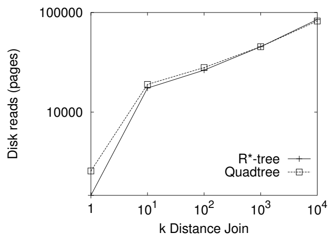
Figure 6.10: Distance join query with bulkloaded indices: Number of disk page accesses with uniform datasets



(a) MAPS-2D



(b) MAPS-4D



(c) MAPS-8D

Figure 6.11: Distance join query with bulkloaded indices: Number of disk page accesses with MAPS datasets

forms the R*-tree by 0.9-2.3X, 1.4-2.5X, and 1.0-3.8X with uniform datasets. In addition, from Figure 6.8, we observe the following: 1) the performance of the Quadtree over the R*-tree increases with increasing data dimensionality, and 2) the Quadtree speedup over the R*-tree decreases with increasing k.

The Figure 6.9 shows the number of disk accesses with skewed MAPS data. Similar to the result with uniform data, with skewed data, the Quadtree outperforms the R*-tree. The only difference is that the relative speedup of the Quadtree over the R*-tree is higher with skewed MAPS data than with uniform data. For instance, the Quadtree outperforms the R*-tree with MAPS datasets by 1.0-1.7X, 0.7-5.6X, and 7.1-9.6X for 2D, 4D, and 8D MAPS datasets.

6.4.2.2 Comparison with bulkloaded indices

Figures 6.10 and 6.11 show the number of disk accesses for uniform and MAPS datasets when bulkloaded indices are used as underlying index structures. Unlike previous results, when bulkloaded indices are used, the STR R-tree incurs fewer disk accesses than the Quadtree in most cases. The smaller number of disk accesses leads to better performance by the STR R-tree than the Quadtree. For instance, the STR R-tree outperforms the Quadtree by 1.1-2.0X, 1.2-1.5X, 1.9-4.2X in 2D, 4D, and 8D uniform datasets respectively. The STR R-tree also outperforms the Quadtree by 0.8-1.4X, 0.8-1.8X, 1.0-1.8X with 2D, 4D, and 8D MAPS datasets. In both results, the relative speedup of the STR R-tree over the Quadtree increases with increasing k.

6.4.2.3 Analysis

In this section, we discuss factors that contribute to performance differences in distance join query processing.

We first compare R*-trees and Quadtrees constructed using dynamic algorithms. As discussed in Section 6.4.2.1, performance of the R*-tree is worse than the Quadtree, and it becomes even worse with higher dimensional datasets.

In general, index-based distance join algorithms reduce their processing costs by pruning unnecessary item pairs. The effectiveness of pruning is based on how well distances among objects are estimated using the MBR information in the indices. The actual structure of the index can significantly affect this pruning. For example, in the case of the R-tree, if the MBRs have a lot of overlap, then the pruning is going to be less effective. In other words, the structure of the index can have a dramatic impact on the performance of the distance join algorithm.

As data dimensionality increases, the structure of the R*-tree deteriorates rapidly, generating large MBRs and increasing overlaps among MBRs. For example, the average number of overlapping leaf-level MBRs per random point is 1.5, 2.8, and 26.2 for the uniform 2D, 4D, and 8D datasets containing 0.5 million points respectively. Due to the large and overlapping MBRs in 4D and 8D datasets, estimated distances using MBRs are often highly deviated from actual distances among objects, and the difference can lead to evaluating unnecessary item pairs first rather than item pairs that actually containing top k closest object pairs. This effect can increase distance computations and queue operations. We note that inherently the Quadtree has no overlaps among MBRs and also the STR R-tree has almost no overlaps among MBRs with 4D and 8D datasets. The average number of overlapping leaf-level MBRs per random point in the STR R-tree is 0.99, 0.98, and 0.91 for the uniform 2D, 4D, and 8D datasets respectively.

Now let us compare the performance of distance joins on the bulkloaded STR R-tree and Quadtree. While the dynamically constructed R*-tree is significantly slower than the dynamically constructed Quadtree, the bulkloaded STR R-tree is faster than the bulkloaded Quadtree in distance join query processing. Let us consider the factors that explain the differences in the join performance with different index construction methods.

While the Quadtree uses a disjoint decomposition of spaces, the main drawback of the R-tree partitioning is that it does not result in a disjoint decomposition of space. As a result, nodes in the R-tree can overlap, which can affect join performance. However, in the STR R-tree used here, this drawback is mostly overcome by its decomposition rules, especially for point data. According to the STR algorithm, there can be some overlaps among non-leaf nodes, but there is almost no overlap among leaf nodes for point data. In addition, since only 0.5%-1.1% are nonleaf nodes in the STR R-tree with 2 to 8 dimensional data, overlaps in the STR R-tree are negligible. Leaf-level MBR overlaps are 0.99, 0.98, and 0.92 with uniform 2D, 4D, and 8D datasets each. (Non-leaf level MBR overlaps are 2.23, 3.82, and 7.61 with the same datasets.)

While the overlap amongst the MBRs is no longer a distinguishing performance factor, the two index structures are quite different in terms of the degree of MBR granularity and the shape of MBRs. For example, with our uniform datasets, while 756, 1296, and 2916 leaf nodes cover the entire 2D, 4D, and 8D data space in the STR R-tree, 1024, 4096, and 65501 leaf nodes cover the same data space in the Quadtree. This basically means that the size of MBRs in the Quadtree is smaller than that of the R-tree. (Note that both indices use the same page size.) Too many MBRs with very small sizes also can be a factor that degrades performance. Essentially, the number of node accesses is proportional to $|N_A| * |N_B|$, where N_A and N_B correspond to the number of nodes in the two indices, A and B, that are being joined. When indices have a large number of small nodes, the number of node accesses is increased (subsequently disk I/Os as well). This effect is clearly shown in the following performance measures. When k is set to 100 with 8D uniform data, the number of disk accesses is 56,404 and

154,970 in the R-tree and the Quadtree respectively. This results in a faster R-tree performance over the Quadtree by 2.6 times.

In summary, based on this experiment with distance join queries, we make the following observations:

1. When indices are constructed using dynamic algorithms, the Quadtree outperforms the R*-tree. When indices are constructed using bulkloading algorithms, the STR R-tree outperforms the Quadtree.

2. Distance join performance of the R*-tree is significantly affected by the index construction methods, while the Quadtree is relatively less affected by the index construction methods.

3. The key factor that influences the performance of index-based distance join algorithms is how to decompose data into a set of MBRs with appropriate granularity.

6.5 Discussion

6.5.1 Choice of an indexing structure for ProCC

In this section, we discuss the choice of indexing structures for ProCC described in Chapter IV, and see how the experimental results presented in previous and this chapter can be actually applied in building real applications. Although a difference choice of index structures can lead to a very different result in performance, in many scientific applications, a common choice of indexing method is the R*-tree.

For the choice of the indexing structure in proCC, we consider two popularly used indexing structures: R*-tree and Quadtree. Also, we consider major factors to choose a right indexing method for ProCC: data dimensionality, data update characteristic and query operation. First, for data dimensionality, ProCC transforms protein structure data into a set of 10 dimensional point vectors. So it handles relatively higher dimensional data. Second, for data update characteristic, protein structure databases such as PDB are dynamic as newly solved protein structures are added daily into databases. Finally, for query operation, the type of query operation needed in ProCC is a range query operation.

Considering the above factors and experimental result presented in previous and this chapters, the Quadtree is a better choice for ProCC. To confirm this, we conducted experiments to compare the performance of the R*-tree and the Quadtree for proCC data. The SCOP 1.67 database is used to build the R*-tree and the Quadtree indices and randomly selected 1000 domains from SCOP 1.69 database are used as queries. The experimental result shows that the average execution time per query with the Quadtree is 2.3 seconds while it is 10.2 seconds with the R*-tree. This shows that our experimental study provides some useful insights and guidelines for choosing a right indexing method.

6.6 Conclusions

In this chapter, we have compared the performance of the R*-tree and the Quadtree for indexing low to medium dimensional datasets under a variety of settings. Our experimental results reveal several important points: 1) The query processing performance of indices is affected by the type of query operations and the index construction algorithms. 2) The query processing performance of R*-tree is significantly affected by the index construction methods, while the Quadtree is relatively less affected by the index construction method. 3) The regular and disjoint partitioning method used by the Quadtree has an inherent structural advantage over the R*-tree in performing kNN and distance join queries. 4) An often dismissed index structure (the Quadtree) can be a better choice than the widely used R*-tree for index-based kNN query and distance join algorithms when indices are constructed dynamically.

With the increasing importance of multi-dimensional point indexing methods in a variety of data-centric applications, the results of this experimental study provide a useful guideline in selecting the appropriate multi-dimensional point indexing method. We hope that this study provides valuable insights for choosing the right indexing method for k nearest neighbor and distance join query operations and also motivates a careful consideration of index structures when designing methods for other important multi-dimensional query operations.

CHAPTER VII

Conclusion

This thesis develops efficient and scalable querying methods for biological data. Since many biological datasets are large and are rapidly growing in size [2, 3], it is natural to consider the use of index-based methods for efficient query processing. Therefore, the bulk of this thesis focuses on developing index-based methods for querying large biological datasets.

In Chapter II, we presented a new algorithm, called miBLAST [52], that evaluates batch query workloads efficiently. A common computational task in a number of bioinformatics applications is to search a large nucleotide sequence database using a *set* of nucleotide sequence queries. Existing tools [9, 15, 51, 72] such as BLAST [9] can be used, but they are computationally very expensive. To meet the need for a new tool that can significantly speed up the evaluation of large query workloads, we developed miBLAST. At the core, miBLAST employs a q-gram filtering and an index join for efficiently detecting similarity between the query sequences and database sequences. This set-oriented technique, which indexes both the query and the database sets, results in substantial performance improvement over existing methods.

In Chapter III, we described another sequence alignment algorithm, called ProbeMatch, for matching a large number of oligonucleotides against a genome database. Important biological applications such as genome resequencing and genome-wide polymorphism discovery require matching such a large set of oligonucleotides against genome databases allowing only a few errors [39]. To speedup this sequence alignment task, we developed a new method called ProbeMatch. ProbeMatch uses gapped q-grams and q-grams of various patterns. This technique results in fewer initial sequences to examine with no loss in sensitivity under a k mismatch model. Unlike existing tools which only performs ungapped alignments [6, 38, 62], ProbeMatch generates both ungapped and gapped alignments, and shows its efficiency and effectiveness by producing more alignments fast.

In Chapter IV, we go beyond handling sequence data, and described an algorithm for handling more complex biological data such as protein structure data [22, 73]. To query protein structure data, we develop an algorithm that models protein structure data as a multi-dimensional data set, and uses a multi-dimensional index structure for an efficient data retrieval. Based on this approach, we develop a tool, called ProCC [53], for automated and efficient protein classification and clustering. With no significant difference in overall accuracy, we showed that proCC provides a clear advantage over existing methods [20] with respect to computational cost.

We developed various index-based algorithms for searching biological databases, however, a fundamental question remains: What is a good choice for an index structure for an index-based application? In order to answer the above question, in Chapter V and VI, we investigate an essential database problem which reexamines the state-of-the-art multi-dimensional point index structures and provided useful guidelines in selecting the appropriate multi-dimensional indexing method for index-based applications [54].

In the long run, index-based methods can lead to new and more efficient algorithms for querying and mining biological datasets. The examples above, which include query processing on biological sequence and geometrical structure datasets, both employ index-based methods very effectively. While the database research community has long recognized the need for index-based query processing algorithms, the bioinformatics community has been slow to adopt such algorithms. However, since many biological datasets are growing very rapidly, database-style index-based algorithms are likely to play a crucial role in modern bioinformatics methods. The work proposed in this thesis lays the foundation for such methods.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Critical Assessment of Fully Automated Structure Prediction. http://cafasp4.cse.buffalo.edu/dp/update.html.

[2] Protein Data Bank. http://www.rcsb.org/pdb .

[3] Protein Information Resource. http://pir.georgetown.edu.

[4] PyMol. http://www.pymol.org.

[5] SVM^{light} Support Vector Machine. http://svmlight.joachims.org.

[6] A. Cox. Eland: Efficient local alignment of nucleotide data (unpublished).

[7] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search In Sequence Databases. In *FODO*, pages 69–84, 1993.

[8] S. F. Altschul and W. Gish. Local Alignment Statistics. *Methods Enzymol*, 266:460–480, 1996.

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *J.Mol.Biol.*, 215:403–410, 1990.

[10] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings*, 30:483–485, 1967.

[11] S. W. andUdi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35:83–91, 1992.

[12] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles$^+$. In *SIGMOD*, pages 322–331, 1990.

[13] S. Berchtold, C. Böhm, and H.-P. Kriegel. The Pyrmaid-Technique: Towards Breaking the Curse of Dimensionality. In *SIGMOD*, pages 142–153, 1998.

[14] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *VLDB*, pages 28–39, 1996.

[15] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based Database Searching Using a Suffix Array QUASAR. RECOMB, 1999.

[16] 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 *SIGMOD*, pages 383–444, 1994.

[17] O. Çamoglu, T. Can, A. K. Singh, and Y.-F. Wang. Decision tree based information integration for automated protein classification. *Journal of Bioinformatics and Computational Biology*, 13(3):717–742, 1996.

[18] O. Çamoglu, T. Kahveci, and A. K. Singh. Index-based similarity search for protein structure databases. *Journal of Bioinformatics and Computational Biology*, 2(1):99–126, 2004.

[19] J.-M. Chandonia, G. Hon, N. S. Walker, L. L. Conte, P. Koehl, M. Levitt, and S. E. Brenner. The ASTRAL Compendium in 2004. *Nucleic Acids Research*, 32:D189–92, 2004.

[20] S. Cheek, Y. Qi, S. S. Krishna, L. N. Kinch, and N. V. Grishin. SCOPmap: Automated assignment of protein structures to evolutionary superfamilies. *BMC Bioinformatics*, 5(1):197, 2004.

[21] D. Chivian, D. E. Kim, L. Malmström, P. Bradley, T. Robertson, P. Murphy, C. E. Strauss, R. Bonneau, C. A. Rohl, and D. Baker. Automated prediction of CASP-5 structures using the Robetta server. *Proteins*, 53(6):524–533, 2003.

[22] L. L. Conte, S. E. Brenner, T. J. P. Hubbard, C. Chothia, and A. G. Murzin. SCOP database in 2002: refinements accommodate structural genomics. *Nucleic Acids Research*, 30(1):264–267, 2002.

[23] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[24] A. E. Darling, L. Carey, and W. chun Feng. The Design, Implementation, and Evaluation of mpiBLAST. 4th International Conference on Linux Clusters, 2003.

[25] R. Day, D. A. Beck, R. S. Armen, and V. Daggett. A consensus view of fold

space: Combining scop, cath, and the dali domain dictionary. *Protein Science*, 12:2150–2160, 2003.

[26] G. J. E and Z. Ralf. SSEP-Domain: protein domain prediction by alignment of secondary structure elements and profiles. *Bioinformatics*, 22(2):181–187, 2006.

[27] A. J. Enright and C. A. Ouzounis. An automatic graph layout algorithm for similarity and network visualization. *Bioinformatics*, 17(9):853–854, 1995.

[28] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and Effective Querying by Image Content. *J. Intell. Inf. Syst.*, 3:231–262, 1994.

[29] L. Florea, G. Hartzell, Z. Zhang, G. M. Rubin, and W. Miller. A Computer Program for Aligning a cDNA Sequence with a Genomic DNA Sequence. *Genome Research*, 8:967–974, 1998.

[30] D. Frishman and P. Argos. Knowledge-based protein secondary structure assignment. *Proteins*, 23(4):566–579, 1995.

[31] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1997.

[32] I. Gargantini. An effective way to represent quadtrees. *CACM*, 25(12):905–910, 1982.

[33] G. Getz, M. VendruScolo, D. Sachs, and E. Domany. Automated assignment of

SCOP and CATH protein structure classifications from FSSP scores. *Proteins*, 46:405–415, 2002.

[34] W. Gish. WU-BLAST. http://blast.wustl.edu. http://blast.wustl.edu, 1996-2004.

[35] J. Gough, K. Karplus, R. Hughey, and C. Chothia. Assignment of Homology to Genome Sequences using a Library of Hidden Markov Models that Represent all Proteins of Known Structure. *Bioinformatics*, 313(4):903–919, 2001.

[36] R. L. Grossman, C. Kamath, P. Kegelmeyer, V. Kumar, and R. R. Namburu, editors. *Data Mining for Scientific and Engineering Applications.* Kluwer Academic Publishers, 2001.

[37] A. Guttman. R-trees: a Dyanmic Index Structure for Spatial Indexing. In *SIGMOD*, pages 44–57, 1984.

[38] H. Li. Mapping and assembly with quality. http://maq.sourceforge.net.

[39] L. W. Hillier1, G. T. Marth, A. R. Quinlan, D. Dooling, G. Fewell, D. Barnett, P. Fox, J. I. Glasscock, M. Hickenbotham, W. Huang, V. J. Magrini, R. J. Richt, S. N. Sander, D. A. Stewart, M. Stromberg, E. F. Tsung, T. Wylie, T. Schedl, R. K. Wilson, and E. R. Mardis. Whole-genome sequencing and variant discovery in c. elegans. *Nature Methods*, 5:183–188, 2008.

[40] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, 1998.

[41] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[42] G. R. Hjaltason and H. Samet. Improved Bulk-Loading Algorithms for Quadtrees. In *ACM-GIS*, pages 110–115, 1999.

[43] G. R. Hjaltason and H. Samet. Speeding up construction of pmr quadtree-based spatial indexes. In *VLDB Journal*, pages 109–137, 2002.

[44] E. G. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Line Segment Databases. In *SIGMOD*, pages 205–214, 1992.

[45] E. G. Hoel and H. Samet. Benchmarking Spatial Join Operations with Spatial Output. In *VLDB Journal*, pages 606–618, 1995.

[46] L. Holm and C. Sander. Protein structure comparison by alignment of distance matrices. *Journal of Molecular Biology*, 233:123–138, 1993.

[47] L. Holm and C. Sander. Touring protein fold space with Dali/FSSP. *Nucleic Acids Research*, 26:316–319, 1998.

[48] J. Hou, G. E. Sims, C. Zhang, and S.-H. Kim. A global representation of the protein fold space. In *The Proceedings of the National Academy of Sciences*, volume 100, pages 2386–2390, 2003.

[49] I. Kamel and C. Faloutsos. On Packing R-trees. In *CIKM*, 1993.

[50] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *VLDB*, pages 500–509, 1994.

[51] W. J. Kent. BLAT-The BLAST-like Alignment Tool. *Genome Research*, 12:656–664, 2002.

[52] Y. J. Kim, A. Boyd, B. D. Athey, and J. M. Patel. miBLAST: Scalable Evaluation of a Batch of Nucleotide Sequence Queries with BLAST. *Nucleic Acid Research*, 33(13):4335–4344, 2005.

[53] Y. J. Kim and J. M. Patel. A framework for protein structure classification and identification of novel protein structures. *BMC Bioinformatics*, 7:456, 2006.

[54] Y. J. Kim and J. M. Patel. Rethinking Choices for Multidimensional Point Indexing: Making the Case for the Often Ignored Quadtree. In *CIDR*, pages 281–291, 2007.

[55] J. Klivington. Personal Communication, 2005.

[56] I. Korf and W. Gish. MPBLAST: Improved BLAST Performance with Multiplexed Queries. *Bioinformatics*, 16:1052–1053, 2000.

[57] R. Kothuri, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparision using GIS Data. In *SIGMOD*, pages 546–556, 2002.

[58] P. E. D. Lachance and A. Chaudhuri. Microaarry Analysis of Developmental Plasticity in Monkey Primary Visual Cortex. *Journal of Neurochemistry*, 88:1455–1469, 2004.

[59] I. Lee. Personal Communication, 2005.

[60] I. Lee, A. A. Dombkowski, and B. D. Athey. Guildlines for incorporating non-perfectly matched oligonucleotides into target-specific hybridization probes for a DNA microarray. *Nucleic Acids Research*, 32:681–690, 2004.

[61] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, pages 497–506, 1997.

[62] R. Li, Y. Li, K. Kristiansen, and J. Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[63] K.-I. Lin, H. Jagadish, and C. Faloutsos. The TV-tree: An Index Structure for High-Dimensional Data. *VLDB Journal*, 3:517–542, 1994.

[64] T. Madej, J.-F. Gibrat, and S. H. Bryant. Threading a database of protein cores. *Proteins*, 23(3):356–369, 1995.

[65] M. Madera, C. Vogel, S. K. Kummerfeld, C. Chothia, and J. Gough. The SUPERFAMILY database in 2004: additions and improvements. *Nucleic Acids Research*, 32:235–239, 2004.

[66] A. C. Martin. The ups and downs of protein topology: rapid comparison of protein structure. *Protein Engineering*, 13:829–837, 2000.

[67] Y. S. Moon, K. Y. Whang, and W. S. Han. General Match: A Subsequence Matching Method in Time-Series Databases Based on Generalized Windows. In *SIGMOD*, pages 382–393, 2002.

[68] Y. S. Moon, K. Y. Whang, and W. K. Loh. Duality-Based Subsequence Matching in Timeseries Databases. In *ICDE*, pages 263–272, 2001.

[69] A. G. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. Scop: A structural classification of proteins database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247:536–540, 1995.

[70] G. Navarro and R. Baeza-Yates. A Practical q-gram Index for Text Retrieval Allowing Errors. *CLEI Electronic Journal*, 1(2):1725–1729, 1998.

[71] W. Niblack, R. Barber, W. Equitz, M. D. Flickner, E. H. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC Project: Query Images by Content Using Color Texture, and Shape. In *SPIE*, pages 173–187, 1993.

[72] Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: A Fast Search Method for Large DNA Databases. *Genome Research*, 11:1725–1729, 2001.

[73] C. A. Orengo, A. D. Michie, S. Jones, D. T. Jones, M. B. Swindells, and J. M. Thornton. CATH - a hierarchic classification of protein domain structures. *Structure*, 5:1093–1108, 1997.

[74] P. Røgen and B. Fain. Automatic classification of protein structure by using gauss integrals. In *The Proceedings of the National Academy of Sciences*, volume 100(1), pages 119–114, 2003.

[75] J.-M. Rouillard, M. Zuker1, and E. Gulari. OligoArray 2.0: design of oligonucleoide probes for DNA microarray using a thermodynamic approach. *Nucleic Acids Research*, 31:3057–3062, 2003.

[76] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *SIGMOD*, pages 17–31, 1985.

[77] S. Burkhardt and J. Kärkkäinen. One-gapped q-gram filters for levenshtein distance. *Combinatorial Pattern Matching*, pages 73–85, 2001.

[78] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informaticae XXIII*, pages 1001–1018, 2003.

[79] H. K. Saini and D. Fischer. Meta-DP: domain prediction meta-server. *Bioinformatics*, 21(12):2917–2920, 2005.

[80] H. Samet. The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, 16(2):187–260, 1984.

[81] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human-Mouse Alignments with BLASTZ. *Genome Research*, 1:103–107, 2003.

[82] I. N. Shindyalov and P. E. Bourne. Protein structure alignment by incremental combinatorial extension(ce) of the optimal path. *Protein Engineering*, 9:739–747, 2005.

[83] A. P. Singh and D. L. Brutlag. Hierarchical protein structure superposition using both secondary structure and atomic representation. In *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology*, pages 284–293, 1997.

[84] M. Uddin, D. E. Wildman, G. Liu, W. Xu, R. M. Johnson, P. R. Hof, G. Kapatos, L. I. Grossman, and M. Goodman. Sister Grouping of Chimpanzees and Humans as Revealed by Genome-wide Phylogentic Analysis of Brain Gene Expression Profiles. In *The Proceedings of the National Academy of Sciences*, volume 101, pages 2957–62, 2004.

[85] S. Van Dongen. *Graph clustering by flow simulation.* PhD thesis, University of Utrecht, 2000.

[86] H. Wang, B. C. Ooi, K.-L. Tan, T.-H. Ong, and L. Zhou. BLAST++: BLASTing Queries in Batches. *Bioinformatics*, 19:2323–2324, 2003.

[87] N. Weskamp, D. Kuhn, E. Hüllermeier, and G. Klebe. Efficient similarity search in protein structure database by k-clique hashing. *Bioinformatics*, 20(10):1522–1526, 2004.

[88] M. A. Wright and G. M. Church. An open-source oligomicroarray standard for human and mouse. *Nature biotechnology*, 20:1082–1083, 2002.

[89] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A Greedy Algorithm for Aligning DNA Sequences. *Journal of Computational Biology*, 7:203–214, 2000.