INDEXING TECHNIQUES FOR METRIC DATABASES WITH COSTLY SEARCHES

By

JAYENDRA G. VENKATESWARAN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2007
©2007 Jayendra G. Venkateswaran
To my wonderful parents
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Dr. Tamer Kahveci, for his exceptional guidance during the course of this research. He has been a constant source of motivation and was always available for technical discussions and professional advice.

I am also indebted to my co-advisor, Dr. Christopher Jermaine for his excellent guidance and support in major part of my research.

I am grateful to my dissertation committee members Alin Dobra, Sartaj Sahni and Lei Zhou for their help and guidance.

Additionally, I thank all my friends at University of Florida-Gainesville, for their friendship. Last, but not the least, I owe a special debt of gratitude to my family. I would not have been able to get this far without their constant support and encouragement.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>12</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1  INTRODUCTION</td>
<td>13</td>
</tr>
<tr>
<td>1.1 Similarity Search</td>
<td>13</td>
</tr>
<tr>
<td>1.1.1 Similarity Search Query</td>
<td>14</td>
</tr>
<tr>
<td>1.1.2 Similarity Join Query</td>
<td>16</td>
</tr>
<tr>
<td>1.2 Performance Issues</td>
<td>18</td>
</tr>
<tr>
<td>1.3 Contributions and Organization</td>
<td>19</td>
</tr>
<tr>
<td>1.3.1 Similarity Search Query</td>
<td>19</td>
</tr>
<tr>
<td>1.3.2 Similarity Join Query</td>
<td>21</td>
</tr>
<tr>
<td>1.3.3 Dissertation Organization</td>
<td>22</td>
</tr>
<tr>
<td>2  METRIC SPACE MODEL</td>
<td>23</td>
</tr>
<tr>
<td>2.1 Euclidean Distance</td>
<td>23</td>
</tr>
<tr>
<td>2.2 Edit Distance</td>
<td>24</td>
</tr>
<tr>
<td>2.3 Earth Mover’s Distance</td>
<td>25</td>
</tr>
<tr>
<td>3  RELATED WORK</td>
<td>27</td>
</tr>
<tr>
<td>3.1 Partition-based Indexing Techniques</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 R-Tree and its Variants</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2 The STR-Ordering</td>
<td>30</td>
</tr>
<tr>
<td>3.2 Nearest Neighbor Queries</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1 k-Nearest Neighbor Query</td>
<td>31</td>
</tr>
<tr>
<td>3.2.2 Reverse-Nearest Neighbor Query</td>
<td>32</td>
</tr>
<tr>
<td>3.2.3 All Nearest Neighbor Query</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Reference-based Indexing Techniques</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1 Distance-matrix based Indexing Methods</td>
<td>36</td>
</tr>
<tr>
<td>3.3.2 Reference-based Hierarchical Indexing Methods</td>
<td>38</td>
</tr>
<tr>
<td>3.4 String Search Methods</td>
<td>40</td>
</tr>
<tr>
<td>4  REFERENCE-BASED INDEXING FOR STATIC DATABASES</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Maximum Variance</td>
<td>42</td>
</tr>
<tr>
<td>4.1.1 Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>4.1.2 Computational Complexity</td>
<td>45</td>
</tr>
</tbody>
</table>
4.2 Maximum Pruning

4.2.1 Algorithm

4.2.2 Computational Complexity

4.2.3 Sampling-based Optimization

4.2.3.1 Estimation of gain

4.2.3.2 Estimation of largest gain

4.2.4 Impact of the Sample Query Set

4.3 Assignment of References

4.3.1 Motivation and Problem Definition

4.3.2 Algorithm

4.3.3 Computational Complexity

4.4 Search Algorithm

4.4.1 Algorithm

4.4.2 Computational Complexity

4.5 Experiments

4.5.1 Effect of the Parameters

4.5.1.1 Impact of \( m \)

4.5.1.2 Impact of query set size, \(|Q|\)

4.5.2 Comparison of Proposed Methods

4.5.2.1 Impact of query range (\( \epsilon \))

4.5.2.2 Impact of number of references (\( k \))

4.5.3 Comparison with Existing Methods

4.5.3.1 Impact of query range (\( \epsilon \))

4.5.3.2 Impact of number of references (\( k \))

4.5.3.3 Impact of input queries

4.5.3.4 Scalability in database size

4.5.3.5 Scalability in string length

5 REFERENCE-BASED INDEXING FOR DYNAMIC DATABASES

5.1 Overview of SP and TP

5.1.1 Basic Approach

5.1.2 Maintaining the Query Distribution

5.2 Single-pass Algorithm in Depth

5.2.1 Computational Complexity

5.3 Three-pass Algorithm in Depth

5.3.1 Computational Complexity

5.4 Experiments

5.4.1 Query Performance

5.4.1.1 Experimental setup

5.4.1.2 Experimental results

5.4.2 Distance Computations

5.4.3 Impact of Index Construction Time

5.4.4 Analyzing the Results
# GENERALIZED NEAREST NEIGHBORS FOR SIMILARITY JOINS

6.1 Problem Definition .................................................. 92
6.2 Overview of the Algorithm ........................................... 93
6.3 Predicting the Solution: Priority Table Construction ........ 94
6.4 Static Search Strategies ............................................. 98
   6.4.1 Fetch All ......................................................... 98
      6.4.1.1 Creating clusters ......................................... 98
      6.4.1.2 Ordering clusters ......................................... 98
      6.4.1.3 Processing clusters ....................................... 99
   6.4.2 Fetch One ......................................................... 99
6.5 Dynamic Strategy ..................................................... 101
6.6 Further Improvements for GNN Queries .......................... 102
   6.6.1 Adaptive Filter ................................................ 102
   6.6.2 Partitioning ..................................................... 103
   6.6.3 Packing .......................................................... 105
6.7 Experiments ........................................................... 105
   6.7.1 Evaluation of Optimizations .................................. 106
   6.7.2 Comparison of Proposed Methods ............................... 110
      6.7.2.1 Evaluation of buffer size ................................. 110
      6.7.2.2 Evaluation of the number of NN ......................... 111
   6.7.3 Comparison to Existing Methods ............................... 111
      6.7.3.1 Evaluation of buffer size ................................. 112
      6.7.3.2 Evaluation of the number NN ............................. 114
      6.7.3.3 Evaluation of database size ............................. 115
      6.7.3.4 Evaluation of the number of dimensions .............. 116

7 CONCLUSION ............................................................ 117

REFERENCES ............................................................... 120

BIOGRAPHICAL SKETCH .................................................. 127
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1</td>
<td>Summary of symbols and definitions.</td>
<td>42</td>
</tr>
<tr>
<td>4-2</td>
<td>A list of proposed methods.</td>
<td>56</td>
</tr>
<tr>
<td>4-3</td>
<td>Comparison with Tree-based index structures.</td>
<td>66</td>
</tr>
<tr>
<td>6-1</td>
<td>Comparison with an Optimal solution.</td>
<td>107</td>
</tr>
<tr>
<td>6-2</td>
<td>Comparison with GORDER.</td>
<td>110</td>
</tr>
<tr>
<td>6-3</td>
<td>Comparison with RkNn.</td>
<td>110</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Similarity Search Example.</td>
<td>14</td>
</tr>
<tr>
<td>1-2</td>
<td>Similarity Join Queries Example.</td>
<td>16</td>
</tr>
<tr>
<td>2-1</td>
<td>Edit Distance Example.</td>
<td>24</td>
</tr>
<tr>
<td>2-2</td>
<td>EMD Example: Image1</td>
<td>25</td>
</tr>
<tr>
<td>2-3</td>
<td>EMD Example: Image2</td>
<td>25</td>
</tr>
<tr>
<td>2-4</td>
<td>EMD Example: Transformed Image1</td>
<td>25</td>
</tr>
<tr>
<td>3-1</td>
<td>Evolution of Index Structures in Metric Spaces.</td>
<td>28</td>
</tr>
<tr>
<td>3-2</td>
<td>Reference-based Indexing Example.</td>
<td>34</td>
</tr>
<tr>
<td>3-3</td>
<td>Example for Omni.</td>
<td>36</td>
</tr>
<tr>
<td>4-1</td>
<td>Maximum Variance example.</td>
<td>43</td>
</tr>
<tr>
<td>4-2</td>
<td>Number of comparisons for Omni with varying number of references.</td>
<td>52</td>
</tr>
<tr>
<td>4-3</td>
<td>Number of comparisons for MP-D for DNA object database for different values of $m$.</td>
<td>59</td>
</tr>
<tr>
<td>4-4</td>
<td>Number of comparisons for MP-D for protein database for different values of $m$.</td>
<td>60</td>
</tr>
<tr>
<td>4-5</td>
<td>Number of comparisons for MP-D for image database for different values of $m$.</td>
<td>60</td>
</tr>
<tr>
<td>4-6</td>
<td>Impact of $</td>
<td>Q</td>
</tr>
<tr>
<td>4-7</td>
<td>Impact of $</td>
<td>Q</td>
</tr>
<tr>
<td>4-8</td>
<td>Comparison of the proposed methods for DNA database for queries with varying ranges.</td>
<td>63</td>
</tr>
<tr>
<td>4-9</td>
<td>Comparison of the proposed methods for Protein database for queries with varying ranges.</td>
<td>64</td>
</tr>
<tr>
<td>4-10</td>
<td>Comparison of the proposed methods for DNA database with a varying number of references.</td>
<td>65</td>
</tr>
<tr>
<td>4-11</td>
<td>Comparison of the proposed methods for Protein database with a varying number of references.</td>
<td>65</td>
</tr>
<tr>
<td>4-12</td>
<td>Comparison with other methods on DNA database for queries with varying ranges.</td>
<td>68</td>
</tr>
<tr>
<td>4-13</td>
<td>Comparison with other methods on protein database for queries with varying ranges.</td>
<td>68</td>
</tr>
<tr>
<td>4-14</td>
<td>Comparison with other methods on image database for queries with varying ranges.</td>
<td>69</td>
</tr>
<tr>
<td>4-15</td>
<td>Comparison with other methods on DNA database for a varying number of references.</td>
<td>70</td>
</tr>
</tbody>
</table>
6-9 Comparison of the proposed methods on two-dimensional image database for different values of $k$. .......................................................... 109

6-10 Comparison with other methods on two-dimensional image database for different buffer sizes. .......................................................... 111

6-11 Comparison with other methods on protein database for different buffer sizes. .... 112

6-12 Comparison of SS, RT, and Mux FD on two-dimensional image database for different values of $k$. .......................................................... 113

6-13 Comparison of SS, RT, and Mux FD on protein databases for different values of $k$. .... 114

6-14 Comparison with other methods on two-dimensional image database for varying database sizes. .......................................................... 115

6-15 Comparison with other methods on two-dimensional image database for varying number of dimensions. .......................................................... 116
INDEXING TECHNIQUES FOR METRIC DATABASES WITH COSTLY SEARCHES

By
Jayendra G. Venkateswaran

December 2007

Chair: Tamer Kahveci
Major: Computer Engineering

Similarity search in database systems is becoming an increasingly important task in modern application domains such as artificial intelligence, computational biology, pattern recognition and data mining. With the evolution of information, applications with new data types such as text, images, videos, audio, DNA and protein sequences have began to appear. Despite extensive research and the development of a plethora of index structures, similarity search is still too costly in many application domains, especially when measuring the similarity between a pair or objects is expensive. In this dissertation, the similarity search queries we consider are classified under similarity search and similarity join queries. Several new indexing techniques to improve the performance of similarity search are proposed. For the similarity search queries, reference-based indexing methods applicable to both static and growing databases are proposed. For similarity join queries, a generalized nearest neighbor framework and several search and optimization algorithms are proposed. The extensive experiments evaluates the different parameters used by the proposed methods and performance improvements over the state-of-art algorithms.
CHAPTER 1
INTRODUCTION

Similarity search refers to the retrieval of database objects that are similar or close to a given query object. It has applications in diverse fields such as databases, artificial intelligence, computational biology, pattern recognition and data mining. The use of data structures to speed the search - known as indexing - is an important technique when answering questions over large databases. Many indexing techniques have been proposed for similarity search. Despite the extensive research and the development of a plethora of index structures, similarity search is still too costly in many application domains especially when measuring the similarity between a pair or objects is expensive. New indexing techniques are needed to improve the performance of searches in these databases.

1.1 Similarity Search

Classical databases have stored numerical or alphabetical information. The database objects are stored using different database models such as the relational [30], object-oriented [3] and object-relational [83] models. Exact match, range and join queries over these models are typically based on whether or not two keys are equal or not equal or whether a key falls in a range of values.

With the evolution of information, applications with new data types such as text, images, videos, audio, DNA and protein sequence have began to appear. The problem with these new data types is that they can neither be ordered for range queries, nor meaningful to perform equality comparisons on them. For example, consider the application of image databases. A user might be interested in retrieving the images from the database that have similar color, texture and shape to the given query image [71]. Existing database models identify images from the filenames, keywords and so on. These approaches cannot be applied for the similarity search query. The query requires a more complicated distance function that can capture the similarity between two images accurately.

*Similarity search* [28, 37, 45] retrieves database objects that are similar or close to a given query object, given some arbitrary, user-defined similarity function. In this dissertation, similarity
search queries are classified under the following two categories: a) similarity search queries and b) similarity join queries. The next two subsections describes these two categories in detail.

### 1.1.1 Similarity Search Query

There are two types of similarity search queries: a) range queries and b) nearest neighbor queries.

Let $D()$ be a distance function. In a **Range Query** (RQ), the goal is to find the database objects that are similar to a given query. Given a database $R$, a query $q$, and a similarity threshold $\epsilon$, the search returns the set of all objects in $S$ whose distance to $q$ is less than $\epsilon$. Formally, the answer to a RQ is defined as:

$$RQ(q, \epsilon) = \{r \in R : D(q, r) \leq \epsilon\}$$

Figure 1-1 presents an example for a RQ. Here the shaded circles represent the objects in $R$. The black circle represents a given query $q$ and the distance threshold is given by $\epsilon$. The query returns the five objects enclosed by the circle with $q$ as the center and $\epsilon$ as the radius.

For an example of a RQ, consider the application spell checker in text retrieval. The **Edit Distance** (ED) \[58\] between two strings gives the minimum number of transformations needed
to convert one string to another. Given a text database, a query string $q$ and a similarity threshold $\epsilon$, the spell checker will look for the database strings that are within the $\epsilon$ distance from the query string. As another example, consider a content-based image retrieval system. A commonly used measure in this application is the Earth Movers Distance (EMD) [71]. EMD measures the transformation needed to convert the histogram distribution of one image to that of another image. This is used as a measure of dissimilarity between two given images. Given an image database, a query image $q$ and an EMD threshold $\epsilon$, the image retrieval system returns the images from the database that have a EMD distance within $\epsilon$ from $q$.

In a Nearest Neighbor (NN) query, given a database $R$ and a query object $q$, the goal is to retrieve the element in $R$ that is closest to $q$ based on some distance measure. The answer to a NN query is defined as:

$$NN(R, q) = \{ r \in R : \forall v \in R, D(q, r) \leq D(q, v) \}$$

In a variation of the NN query, the $k$-Nearest Neighbor ($k$NN) query, given a database $R$, a query $q$ and a positive integer $k$, the goal is to return a set of $k$ objects from $R$ similar to $q$ based on some distance measure. The answer to a $k$NN query is defined as:

$$KNN(R, q, k) = \{ U \subseteq R : |U| = k, \forall u \in U, v \in \{ R - U \}, D(q, u) \leq D(q, v) \}$$

Figure 1-1 presents an example of the two NN query types. The result of a NN query on the object $q$ is given by, $NN(R, q) = \{ a \}$. The result of 2NN query on the object $q$ is given by $NN(R, q, 2) = \{ a, b \}$.

For an example application, consider geographic information system (GIS) given in [70]. The user of a tourist information system may require information about the locations of the attractions which are close to the current location, or the tourist might be interested in knowing the three nearest hospitals from the location of a car accident. These queries require computing
the Euclidean distance between a given location to the database of locations of attractions or hospitals.

1.1.2 Similarity Join Query

In a similarity join query [28], given two databases $R$ and $S$, the goal is to retrieve the pairs of objects from the two databases that are similar to each other based on a given criterion. All similarity join queries can be further divided into the following three types: a) the $k$-Nearest Neighbor ($k$NN) join query [19], b) the Reverse Nearest Neighbor (RNN) join query [55] and c) the All Nearest Neighbor (ANN) join query [97]. Figure 1-2 presents an example of the different types of join queries. Here the shaded circles represent the objects from $R$ (\{a, b, c\} $\in$ $R$) and the unshaded circles represent the objects from $S$ (\{1, 2, 3, 4\} $\in$ $S$).

The $k$NN join query is the most commonly used type of join query. Given a positive integer $k$, the join query returns $k$ objects from $S$ similar to the objects in $R$ based on some distance measure. The answer to a $k$NN join query is defined as:

$$KNN(R, S, k) = \{< r, U >: r \in R, U \subseteq S, |U| = k, \forall u \in U, v \in \{S - U\}, D(r, u) \leq D(r, v)\}$$

In the example given in Figure 1-2, a 2NN join query between $R$ and $S$ returns the set of points \{\{a, \{1, 2\}\}, \{b, \{2, 3\}\}, \{c, \{2, 4\}\}\}. For an example application, consider $k$NN
classification in data mining [18]. Given a set of database object that are not classified under any category $R$ and set of objects $S$ that are already classified, the $k$NN query over $S$ is computed for every object in $R$. Then the unclassified objects are assigned to the category that forms the majority of its $k$ nearest neighbors.

In a RNN join query, given $R$ and $S$, the goal is to retrieve the set of all objects from $S$ such that every object from the result set is a nearest neighbor for some object in $R$. For each object in this subset from $S$ ($\forall u \in U, U \subseteq S$), there exists some object in $R$ ($\exists v \in R$) such that $u$ is the nearest neighbor of $v$. The answer to a RNN join query is defined as:

$$RNN(R, S) = \{ U : U \subseteq S, \forall u \in U, \exists v \in R, D(v, u) \leq D(v, t), \forall t \in S \}$$

A variant of this query, the $Rk$NN query, retrieves the set of all objects in $S$ which are one of the $k$ nearest neighbors of some object in $R$.

An example of a reverse 2NN query is given in Figure 1-2. It returns $\{2\}$, which is one of the two nearest neighbor of the objects $\{a, b, c\} \in R$. For an example application of $Rk$NN query, consider a decision support system. Suppose that people usually dine at one of the $k$ nearest restaurants. An entrepreneur who wants to invest in Mexican restaurant business would want to know the potential customers who have a Mexican restaurant as one of the $k$ nearest restaurants. In this application, given the locations of all customers and locations of all restaurants, the $RkNN$ join query would return the locations of the customers who have at least one Mexican restaurant as one of their $k$ nearest ones. These customers are likely to use the restaurant due to its geographical proximity.

In an ANN query, given $R$ and $S$, the goal is to retrieve for every object in $R$ its nearest neighbor in $S$. The answer to an ANN join query is defined as:

$$ANN(R, S) = \{ < r, s > : r \in R, \exists s \in S, D(r, s) \leq D(r, v), \forall v \in S \}$$

The ANN query is not commutative, i.e. in general $ANN(R, S) \neq ANN(S, R)$. In a variant of this query type, called the closest-pair query [19], given a positive integer $k$, the goal is
to retrieve $k$ objects pairs from the two databases that are closest among all objects pairs from the ANN query.

An example ANN query is given in Figure 1-2. It returns the set of object pairs $\{\{a, 1\}, \{b, 2\}, \{c, 4\}\}$. For an example application, consider a geographic information system. The user of a tourist information system may require information about the nearest gas station for each attraction. In this application, given the locations of all gas stations and locations of all attractions, the ANN join query would return the location of the nearest gas station to each attraction.

### 1.2 Performance Issues

There are two issues that make similarity queries hard: 1) Even for small databases sizes, distance computations can be very expensive and 2) For larger databases, even for inexpensive distance computations, limitations in available main memory (when the database size is much larger than the main memory) can result in costly similarity join queries. The remainder of this section describes these two performance issues in detail.

Even though the databases over which similarity search queries are performed, are not always huge (and are sometimes small enough to be stored in several gigabytes of main memory), tremendous expense is associated with computing the distance from a query to each of the data objects. This renders the access method most applicable to main-memory resident data sets – sequential scan – infeasible for similarity search queries. For example, consider searching an image database using Earth Movers’ Distance (EMD) [71] as the similarity measure. Computing EMD is an expensive linear programming problem and takes about 40 seconds to compare two given images. A small database of about 4000 images can be easily loaded into the memory. However, due to the high complexity of EMD, even a single search on this database can take up to 45 hours. As another example, consider a less complex measure, the edit distance (ED), used over DNA databases. Computing the edit distance between two strings requires $O(n^2)$ time [67], which can translate to seconds of CPU time for long strings. The human genome contains 30 million strings of length 100 [10]. Searching for the substrings similar to a given substring in this database can take up to an hour.
The second issue that can affect the performance of a query is large database size, particularly for the various types of join queries. The obvious way to join two databases \( R \) and \( S \) is to perform one distance computation for each \((r \circ s) \in R \times S\). But if \(|R| = |S| = \text{one million}\), this is \(1 \times 10^{12}\) distance computations! For an example application, consider the decision support system. An entrepreneur who wants to invest in Mexican restaurants would want to know the potential customers who have a Mexican restaurant as one of the \( k \) nearest restaurants. If \( R \) is the set of all houses in the country and \( S \) is the set of all restaurants, the similarity join query requires comparison of all pairs of locations from \( R \) and \( S \). The databases are too big to be loaded into the memory. This results in pages being fetched again and again during the search and hence increases the I/O costs. The sort-based and hash-based methods [33, 38, 54] for handling large databases do not work here – these work only for exact-match queries (not similarity join queries). Thus, even with inexpensive distance computations, new indexing techniques are needed.

1.3 Contributions and Organization

In this dissertation, the above two performance issues in metric databases are addressed. The following sections present the contributions of this dissertation.

1.3.1 Similarity Search Query

The problem of reducing the number of distance computations needed to obtain the results of a similarity search query is considered. The work presented in this dissertation tries to solve this problem using several \textit{reference-based indexing} methods [35, 48, 99]. In the reference-based indexing approach, a small fraction of database objects, referred to as the set of \textit{reference objects}, are selected. The distances between references and database objects are pre-computed and stored as an index. Given a query, the search algorithm computes the distance from each of the reference to the query object. Without any further comparisons, objects that are close to or far away from the reference can be removed from the candidate set based upon those distances.
For a similarity search query in a database with a complex similarity measure, three primary problems are addressed while using reference-based indexing: 1) selection of references, 2) assignment of references and 3) reference selection for dynamic databases.

References are selected such that they represent all parts of the database. Two novel strategies for the selection of references are proposed. They are:

- **Maximum Variance:** This approach is based on the following two rules: i) If a query $q$ is close to a reference $v$, the objects far away from $v$ can be pruned and objects close to $q$ are added to the result set, and ii) If $q$ is far from $v$, the objects close to $v$ are pruned. These observations imply that the spread of the database objects around a reference is a good measure for reference quality. In the Maximum Variance method, for each reference candidate, the distance to each database object is computed. Then the variance of these distances is determined. Then references with high variances that can prune objects not pruned by the others are selected.

- **Maximum Pruning:** The second approach is based upon a combinatorial calculation that specifically selects the references to maximize the pruning of objects. It is based on the following two rules: i) Select the references based on the number of objects it can prune, and ii) Objects not pruned by one reference should be pruned by at least one of the other references. Since the complexity of the method increases with the database size, sampling techniques to speed the process are used.

The second problem is the mapping of references to database objects. In the proposed solution, the number of references is larger than in other methods. In order to keep the number of references assigned to each object manageable, only the best references for each given database object are used to index it. Thus, each database object may have a different set of references assigned to it.

The third problem is reference-based indexing for dynamic databases. For such databases, two incremental variations of the Maximum Pruning strategy are proposed. After insertion of a new object, these incremental methods update the reference set using the distribution of
actual queries. The actual queries are sampled using the reservoir algorithm [93]. Realistic environments are simulated, where the distribution of the database objects and queries changes as the application and user base evolve over time.

Papers describing the proposed indexing techniques have been already published. The approaches for static databases from Chapter 4 are from the paper co-authored by Deepak Lachwani, Tamer Kahveci and Christopher Jermaine and were originally published in the 2006 Very Large Data Base (VLDB) conference [91]. An extension of this work on reference indexing with approaches for dynamic databases has been accepted by the Very Large Data Base Journal (VLDBJ) [90].

1.3.2 Similarity Join Query

For similarity join queries, we first consider the problem of high I/O costs due to the redundant disk fetches in large databases. Three strategies are proposed to index and process the disk pages. These strategies reduce this redundancy as well as the overall I/O costs for the queries. For the problem of high CPU costs due to a large number of complex distance computations, several optimizations are proposed that reduce the number of actual objects that are compared.

In this dissertation, the all of the following ideas are introduced first time and have not been proposed elsewhere:

- **The Generalized Nearest Neighbor (GNN) Query**: A new database primitive called the GNN is defined. A GNN query finds the set of all objects \( S_t \subseteq S \) that appear in the \( k \)-NN set of at least \( t \) objects of \( R \), where \( t \) is a cutoff threshold. It provides a generalized framework for the different kinds of nearest neighbor join queries, \( kNN \), \( ANN \) and \( RkNN \).

- **Priority Table**: A global-coarse-grain view of the objects in the data space using a Priority Table (PT) is proposed. The PT defines the object pairs that (potentially) need to be compared to answer a given GNN query.
• **Search Strategies:** The problem of redundant disk seeks is addressed by proposing three search strategies, Fetch All (FA), Fetch One (FO) and Fetch Dynamic (FD). The first algorithm, FA, is a pessimistic approach providing as many candidate pages as possible from $S$ for each page of $R$. The second algorithm, FO, is an optimistic approach that returns one candidate page at a time from $S$ for each page of $R$. The third algorithm, FD, dynamically decides the number of pages that needs to be fetched for each page of $R$ by analyzing query history.

• **Search Optimizations:** The problem of expensive distance computations is addressed by proposing three optimizations. The first one, called the row filter, prunes the pages of $S$ that are not in the $k$NN set of sufficiently many objects in $R$. The second one, called the column filter, prunes the pages of $R$ whose nearest neighbors do not contribute to the result. The third optimization, called the adaptive filter, prunes the objects of $S$ that are too far from the objects of $R$. The costs are reduced further by pre-processing the input databases using a packing technique called Sort-Tile-Recursive (STR) [76].

The paper describing the proposed work on Generalized nearest neighbor for similarity join queries has been published already. The methods and experiments presented in Chapter 6 are from the paper co-authored by Tamer Kahveci and Orhan Camoglu, which was originally published in the 2006 Extended Data Base Technology (EDBT) conference [89].

### 1.3.3 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents the describes a metric space model with few examples. Related work is presented in Chapter 3. Chapter 4 presents strategies for selecting the references and how these references can be used to answer a similarity search query in a static database. Chapter 5 presents the reference selection strategies for databases with frequent updates and varying query distributions. Chapter 6 presents a generalized nearest neighbor framework for large databases using the three search strategies. The conclusions are given in Chapter 7.
CHAPTER 2
METRIC SPACE MODEL

Similarity queries are needed in many diverse applications such as the geographic information system (GIS), the multimedia database system (MDS), the text retrieval and the computational molecular biology. Thus, it is one of the most studied topics in the context of databases. These applications often use a metric similarity measure such as the Euclidean Distance, the Edit Distance and the Earth Mover’s Distance to compute the dissimilarity between two objects. This chapter presents the concept of metric measures and elaborates on several measures that are commonly used in recent databases.

Given any two objects \(a\) and \(b\) from the database, a distance measure \(D()\) is said to be a metric if it satisfies the following four properties:

- \(D(a, b) = D(b, a)\) (Symmetry)
- \(D(a, b) \geq 0\) (Non-Negativity)
- \(D(a, a) = 0\) (Reflexivity)
- \(D(a, c) \leq D(a, b) + D(b, c)\) for any object \(c\) (Triangle Inequality)

If a measure does not satisfy the strict non-negativeness property, the measure is called pseudo-metric. The triangle inequality property is a key property of the metric databases that has been used effectively by the indexing techniques. Following are some of the commonly used measures in spatial, text and image databases respectively.

2.1 Euclidean Distance

A Euclidean Distance (EuD) is used to measure the proximity of the database objects in vector space. Vector space can be seen as a metric space where objects are represented arrays of real numbers. Given two vectors, \(R = \{r_1, r_2, \ldots, r_n\}\) and \(S = \{s_1, s_2, \ldots, s_n\}\) the EuD between them is defined as:

\[
EuD(R, S) = \sqrt{\sum_{i=1}^{n} (r_i - s_i)^2}
\]
The complexity of \( Eud \) is linear in terms of the number of terms or the dimensionality of the vector.

There exists a lot of techniques for indexing in vector spaces [? ]. These methods perform well up to twenty dimensions. After that their performance degrades with the increase in the number of dimensions, referred to as the curse of dimensionality, and thus inefficient in practice. Several applications with objects represented in vectors space, like GIS and decision support systems, use \( Eud \) as the distance measure.

### 2.2 Edit Distance

An Edit Distance (\( ED \)) is used to measure the similarity between two strings. Given two strings \( P \) and \( Q \), the \( ED \) between \( P \) and \( Q \) is the minimum number of edit operations required to transform \( P \) into \( Q \). For each letter \( P_i, P_i \in P \), an edit operation refers to one of the following three: a) insert a new character after the letter \( P_i \), b) delete the letter \( P_i \), and c) replace the letter \( P_i \) with another.

Figure 2-1 presents two strings \( P \) and \( Q \) and the \( ED \) between them. The two strings, each having 12 characters, can be optimally aligned at a cost of 3 (two insertions and one replacement).

The dynamic programming solution to find the \( ED \) and the alignment between two strings runs in \( O(n^2) \) time and space [67], where \( n \) is the average length of a string. If \( \epsilon \) is the allowable number of transformations, the space and time complexity for the bounded version of this problem is \( O(n\epsilon) \) \([4, 6, 65, 87]\). In many applications, \( \epsilon = O(n) \), thus making the complexity of the bounded version \( O(n^2) \) \([42, 50]\). Several applications, such as spell check and computational molecular biology, use \( ED \) to compare two given strings.
2.3 Earth Mover’s Distance

An Earth Mover’s Distance (EMD) [71] is a measure of work needed to transform the histogram distribution of one image to another. Given two distributions, one can be seen as a mass of earth properly spread in space, while the other distribution can be seen as a collection of holes in that same space. It can always be assumed that there is at least as much earth as needed to fill all the holes to capacity by switching the earth and the holes if necessary. Then, the EMD measures the least amount of work needed to fill the holes with the earth. Here, a unit of work corresponds to transporting a unit of the earth by a unit of the (ground) distance.

Figures 2-2 to 2-4 present an example of for the EMD similarity measure. Here the black and grey bars represent the given two simple histogram distributions (Figures 2-2 and 2-3). The EMD is given by the least amount of work needed to convert distribution Figure 2-2 to Figure 2-4, i.e. transforming the region represented by white block in the first bin to the grey region represented by second bin.

The EMD involves solving a linear programming based transportation problem [31], which may take a long time. For example, for a 256-dimensional features extracted from images that are partitioned into 8 X 12 tiles, each EMD computation takes 40 s, so, a similarity search on
a database of 4,000 images can take almost 45 hours to complete. The EMD has been used extensively in medical image applications. Here images are represented by multi-dimensional distributions of the histogram of its features such as color, shape and texture.
CHAPTER 3
RELATED WORK

Different indexing schemes have been proposed to speed up the similarity search queries. The basic idea in these methods is to use the triangle inequality property of the metric measure to filter out the database objects that can be proved to be far enough from the query objects, without doing the actual distance computations. These techniques in general follow the following two step filter-refine process:

- **Filter Phase**: This phase determines the collection of objects that can be the candidates for the result set, referred to as a *candidate set*.
- **Refinement Phase**: The actual similarity of each object in the candidate set with the query is computed to eliminate the false positives and to find the answer to the query.

The evolution of the index structures discussed in this section is given in Figure 3-1, based on the taxonomy of several multi-dimensional index structures given by Gaede and Gunther [37]. The indexing techniques can be classified under two categories: Partition-based indexing techniques and Reference-based indexing [28, 45] techniques.

Numerous methods employ index structures to filter unpromising database objects quickly. One of the important classes of index structures partitions the data space into *hierarchical sets*. Section 3.1.1 discusses the index structures that belong this class with a focus on the R-Tree. Section 3.1.2 discusses how the R-Trees are packed to improve the query performance. Section 3.2 presents the existing methods for NN queries based on partition-based approach. Section 3.3 presents another important class of index structures, namely *reference-based indexing*. Section 3.4 discusses methods that are specific for the text retrieval application.

### 3.1 Partition-based Indexing Techniques

The first class of index structures, partition-based techniques, divide the data space into sets. Objects in each set are close to each other in the data space. Each set is represented by its bounding box or region, so that comparing a query object with the representative box or region has good chance of eliminating the objects in them without any further comparisons. Generally, these methods are hierarchical in nature, i.e. the sets are recursively divided into subsets.
Several index structures have been proposed in literature for storage and retrieval using partition-based indexing techniques. Some of the popular ones include the kd-tree [11, 12], the R-tree [43], quadtrees [36, 74], the R+-Tree [79], the R*-Tree [68], the X-Tree [14] and the SR-Tree [53]. These techniques make extensive use of coordinate information to group and classify points in the space. For example, kd-trees divide the space along different coordinates and R-trees group points in hyper-rectangles.

### 3.1.1 R-Tree and its Variants

This section focuses on the R-Tree structure. R-Tree is one of the most important partition-based index structure. It is one of the most common hierarchical structure for indexing spatial objects in high-dimensional spaces. A number of R-Tree variants have evolved over all these years.

R-Tree [43] is a multi-dimensional generalization of the B-Tree [8]. Similar to B-tree, R-tree is a height-balanced tree with index records in its leaf nodes containing pointers to data objects.
It uses the minimum bounding rectangle (MBR) as a representation of the underlying objects. MBR of a group of objects is the smallest rectangle enclosing them. Entries in the leaf nodes are of the form (MBR, $ObjPtr$), where MBR represents the bounding box of the object pointed by $ObjPtr$. The entries in non-leaf nodes are of the form (MBR, $ChildPtr$), where MBR represents the bounding boxes of the child node pointed by $ChildPtr$. Let $M$ be the maximum number of entries possible in a node, let $m \leq M/2$ represent the minimum number of entries in a node and $d$ the number of dimensions. This lower bound $m$ prevents the degeneration of the tree and ensures efficient storage utilization. If the number of entries in a node falls below $m$, the node is deleted and the rest of its entries are distributed among the sibling nodes.

By storing the bounding boxes of geometric objects such as points, polygons and more complex objects, an R-Trees can be used to determine which objects intersect a given query region. As each node has at least $m$ entries, the height of an R-Tree of $N$ objects can at most be $\log N - 1$. Generally, nodes will tend to have more than $m$ entries, which will decrease the height of the tree and improve space utilization.

In an R-Tree, all data objects that overlap with the query region are searched to retrieve objects in the query region. When a node is searched, more than one subtree may need to be traversed. Thus, it is not possible to guarantee good worst-case performance. With efficient update algorithms, the tree will be maintained in such a form so as to eliminate the irrelevant regions (regions that are away from the query region) and examine only data near the search area. The search algorithm descends down the tree, at each level selecting those entries whose MBRs overlap with that of the query. When a leaf-node is reached, entries whose MBRs overlap with the MBR of the query region, are selected.

A variant of R-Tree, the R+-Tree [79], ensures that there exists only one path while searching for a data object. This is done by splitting the overlapping rectangles and increasing the storage required by having duplicate entries. Another variant of R-Tree, the X-Tree [14] was designed for high-dimensional objects with overlap-free split according to a split history and supernodes. A supernode is an oversized node, which prevents overlap when an efficient
split axis cannot be determined. R*-Tree \[68\] is an improved variant of the R-Tree. In addition to using criteria like margin, area and overlap, it uses the concept of forced-reinsertion to re-organize the structure for better storage utilization. The SS-Tree \[96\] is an index structure designed for similarity indexing for multi-dimensional data. It is an improvement of R*-Tree, but uses bounding spheres instead of bounding rectangles and uses a modified forced re-insertion algorithm. Unlike R*-Tree, the SS-Tree re-inserts entries when the entries in a node are not re-inserted. The SR-Tree \[53\] can be viewed as a combination of SS-Tree and R*-Tree. It uses the intersection between the bounding sphere and the bounding rectangle. This outperforms both SS-Tree and R*-Tree. The size of the directory entry is increased significantly by this approach.

An extensive survey of these methods have been given by Samet \[74\], Gaede and Gunther \[?\] and Bohm et.al. \[18\]. Unfortunately the existing techniques are very sensitive to the data space dimension. The Similarity queries have an exponential dependency on the dimension of the space, referred to as the *curse of dimensionality*. Due to this reason, despite its complexity, researchers prefer metric spaces using complex data types.

### 3.1.2 The STR-Ordering

For applications such as multimedia database systems using static databases, pre-processing the database objects and **packing** the partition-based index structure, say R-Tree, yields better space utilization and improved query performance. **Packing** refers to grouping of similar objects (objects within a close neighborhood) together. The problem of grouping the objects necessitated the usage of several heuristics. Space-filling curves such as Z-Ordering \[73\] and Hilbert-Ordering \[51\] transform the data objects into an one-dimensional space and order them. There exists several extensive works on the space-filling curves \[17, 73, 75\]. This section describes another heuristic, the STR-Ordering proposed by Leutenegger et.al \[76\].

The Sort-Tile Recursive (STR) is a method to order the MBRs of R-Tree based static databases. Let \(N\) be the number of \(d\)-dimensional objects and \(b\) the number of entries per node of the R-Tree. The data space is divided into \(\sqrt[3]{r/b}\) vertical slices so that each slice contains enough
rectangles to pack roughly $\sqrt{r/b}$ nodes. Then, the entries are recursively partitioned. Following steps are involved in packing,

- Determine the number of leaf pages $P = r/b$ and let $S = P^{1/n}$ be the number of slices.
- Sort the MBRs by the first dimension of the center point and partition them into $S$ vertical slices. A slice consists of a run of $(P^{n-1/n} \times b)$ consecutive MBRs from the sorted list. The last slice may contain fewer than $(P^{n-1/n} \times b)$ rectangles.
- Each slice is now processed with the remaining $n - 1$ coordinates and this continues recursively.

The STR method orders the database objects such that similar objects are grouped together and packed in either the same or neighboring nodes. By effectively pruning the objects using their MBRs, the number of distance computations of objects and hence the CPU time of the search can be improved significantly. Every node in the index structure, say R-Tree, is packed up to its capacity. This reduces the number of nodes and hence the height of the underlying R-Tree. Each node is stored in a disk page. Reduction in the number of nodes decreases the number of pages fetched from the disk and thus the I/O cost of a search improves dramatically.

### 3.2 Nearest Neighbor Queries

The commonly used nearest neighbor queries are: a) $k$-Nearest Neighbor (KNN) b) Reverse Nearest Neighbor (RNN) query and c) All Nearest Neighbor (ANN) query.

#### 3.2.1 $k$-Nearest Neighbor Query

A number of index-based methods have been developed for $k$NN queries. Hjaltson and Samet [44] used PMR quadtree to index the search space. They search this tree in a depth-first manner until the $k$ nearest neighbors are found. It is an edge-based variant of PM quadtree that uses probabilistic splitting rule.

Roussopoulos et. al., [70] employed a branch-and-bound R-Tree traversal algorithm. It uses the MINDIST and MINMAXDIST to order the MBRs of the R-Tree. Given two MBRs, the MINDIST gives the minimum distance between them while MINMAXDIST guarantees the presence of at least one object in the second MBR. The $k$ nearest objects are stored in a buffer in
sorted order. The MBRs having a distance higher than the current $k$th largest distance are pruned during $k$NN search.

The two-phase method [56] gets the resultant objects in two phases. In the first phase, a $KNN$ search is performed using the distance function on feature vectors. The actual distance to these $k$ objects are computed and the maximum value is determined. In the second phase, a range query on the index returns all objects within this maximum distance from the same feature distance function. For all the objects from the result set of the range query, the actual object distance is computed and $kNN$s are returned.

Seidl and Kriegel [78] proposed a method that runs in multiple phases, iteratively updating the $kNN$ distance. In the first phase, it sorts the objects in increasing order of their feature distance. Then for every object from the sorted list with distance less than the current $kNN$ distance, the actual query-to-object distance is computed and the $kNN$s are updated.

Berchtold et. al., [13] divide the search space using voronoi cells. It first pre-computes the result of any nearest-neighbor search which corresponds to a computation of the voronoi cell of each database object. Then, the voronoi cells are stored in an index structure efficient for high-dimensional data spaces. As a result, nearest neighbor search corresponds to a simple point query on the index structure. Beyer et. al., [15] show that for a broad set of data distributions most of the known $kNN$ algorithms run slower than sequential scan. Thus, despite its simplicity, sequential scan still remains a formidable competitor to index-based $kNN$ methods.

### 3.2.2 Reverse-Nearest Neighbor Query

Given a query $q$, Reverse Nearest Neighbor (RNN) retrieves all objects that have $q$ as their nearest neighbor. Korn et. al., [55] introduced the *Reverse Nearest Neighbor (RNN)* problem. They pre-compute the nearest neighbor of all the objects in the database and generate the nearest neighbor circles for the objects. Then during search, all objects which have the query in their nearest neighbor circle are retrieved. Because the RNN-tree is optimized for RNN, but not NN search, Korn et. al., [55] use an additional (conventional) R-tree on the data points for nearest neighbors and other spatial queries.
In order to avoid the maintenance of two separate structures, Yang and Lin introduced the Rdnn-tree for RNN queries [98]. Similar to the RNN-tree, a leaf node of the RdNN-tree contains vicinity circles of data points. On the other hand, an intermediate node contains the MBR of the underlying points (not their vicinity circles), together with the maximum distance from every point in the sub-tree to its nearest neighbor. As given in their experiments, the RdNN-tree is efficient for both RNN and NN queries because, intuitively, it contains the same information as the RNN-tree and has the same structure (for node MBRs) as a conventional R-tree.

Stanoi et. al. [82], does not use pre-computation and is to extend an existing solution to bi-chromatic RNN query. The basic idea is to find the RNNs of an object, is to dynamically construct the influence region or a Voronoi cell by examining the R-tree of the database objects. Here, an influence region is defined as a polygon in space which encloses the locations that are closer to the query object than to any other object. Once the influence region is computed, a range query in the R-tree of objects is performed to locate RNNs of the object.

Tao et. al., [84] generalize the RNN problem to arbitrary number of NNs using a filter-and-refine approach. The method uses an hierarchical tree-based index structure such as R-Tree to compute a nearest neighbor ranking of the query object. The key idea is to iteratively construct a Voronoi cell around the query object from the ranking. Objects that are beyond k Voronoi planes w.r.t. the query can be pruned and need not to be considered for Voronoi construction. The remaining objects must be refined, i.e. for each of these candidates, a k-nearest neighbor query must be launched.

### 3.2.3 All Nearest Neighbor Query

Despite its wide use in many areas, All Nearest Neighbor (ANN) is the least studied NN query type. MuX uses a two-level index structure called MuX index [19]. At the top level, MuX index contains large pages (or MBRs). At the next level, these pages contain much smaller buckets. For each bucket from S, it computes a pruning distance as it scans the candidate points from R. It prunes the pages, buckets, and points of S beyond this distance for each bucket of R. GORDER [97] is a block nested loop join method. It first reduces the dimensionality of R and S.
Figure 3-2. Reference-based Indexing Example.

by using Principal Component Analysis. It then places a grid on the space defined by the reduced dimensions and hashes data objects into grid cells. Later, it reads blocks of data objects from grid cells by traversing the cells in grid order and compares all the objects in pairs of grid cells whose MINDIST is less than the pruning distance defined by the $k$th NN.

3.3 Reference-based Indexing Techniques

In a reference-based indexing method, a small fraction of database objects, referred to as the set of reference objects, are selected. The distances between references and database objects are pre-computed and stored as an index. Given a query, the search algorithm finds the distance from each of the reference to the query object. Without any further comparisons, objects that are too close to or too far away from a reference are removed from the candidate set based upon those distances with the help of the triangle inequality.

Figure 3-2 illustrates the reference-based indexing in a hypothetical two-dimensional space. Here, the database objects are represented by points. The distance between two points in this space corresponds to the underlying distance between the two objects (e.g., if the points denote strings, $rdist_1$ between the points $ref_1$ and $p$ corresponds to the edit distance between the strings represented by them). In a reference-based indexing, the distances between the object $p$ and references $ref_1$ and $ref_2$ are pre-computed. Let $rdist_1$ and $rdist_2$ be the two pre-computed distances, respectively. Given a query $q$ with range $r$, the first step is to compute query-to-reference distances $qdist_1$ and $qdist_2$. A lower bound for the distance between $q$ and an object
$p$ with reference $ref_1$ is computed as $\text{bound}_1 = |qdist_1 - rdist_1|$ using the triangle inequality. Similarly, $\text{bound}_2$ gives a lower bound for the distance between $q$ and $p$ with reference $ref_2$.

Since $\text{bound}_1 > r$ with $ref_1$ as the reference, object $p$ can be pruned from the candidate set of $q$.

Let $V = \{v_1, \cdots, v_m\}$ denote the set of reference objects, where $v_i \in S$ and $|V| = m$, the number of references. The distances from the references to the database objects are pre-computed and is given by the set $\{D(s_i, v_j) | (\forall s_i \in S) \land (\forall v_j \in V)\}$. This is a one-time cost for the database.

During search, the algorithm first computes the distances from the query $q$ to the references. For each object $s$, a lower bound ($LB$) and an upper bound ($UB$) for $D(q, s)$ are defined as:

$$LB = \max_{v \in V}\{|ED(q, v) - ED(v, s)|\}$$

$$UB = \min_{v \in V}\{|ED(q, v) + ED(v, s)|\}$$

$LB$ and $UB$ are used in reference-based indexing as follows. For each $(q, s_i)$ pair:

- If $r < LB$, add $s$ to the pruned set.
- If $r > UB$, add $s$ to the result set.
- If $LB \leq r \leq UB$, add $s$ to the candidate set.

After this pruning, only the objects in the candidate set are compared with $q$ using the costly comparison operation to complete the query.

The factors that determine the cost of the strategies used for selection and assignment of references to database objects are memory and computation time. Let the available main memory be $B$ bytes and $|S| = N$ be the number of objects in the database. We assume four bytes are used to store a distance value and an object uses on average $z$ bytes of storage. Thus, $zm$ bytes are needed to store a reference. For each object $s \in S$ and its reference $v_i \in V$, the object-reference mapping is of the form $[i, D(s, v_i)]$. Thus, $8mN$ bytes are used to store the pre-computed distances for $N$ objects. The value for $m$ can be obtained by comparing the available memory with the memory needed for storage: $B = 8mN + zm$.

Let $Q$ be the given query set, $t$ be the time taken for one object comparison and $c_{avg}$ be the average number of objects in the candidate set for each query. First, each query object is
Figure 3-3. Example for Omni.

compared with each of the $m$ references. This takes $tm|Q|$ time. Then, for each query, the $c_{avg}$ objects have to be compared to filter out the false positives. This takes $tc_{avg}|Q|$ time. The total time taken is given by $t(m + c_{avg})|Q|$. 

Reference-based similarity searches in metric spaces can be divided into two categories: distance-matrix based and tree-based reference indexing methods. An extensive survey of these categories is given by Chavez et al. [28] and Hjaltason and Samet [45].

3.3.1 Distance-matrix based Indexing Methods

Several methods have been proposed for selecting references in distance-matrix based methods. Filho et al. [35] proposed a method called the Omni. Omni selects references from the convex hull of the dataset. This is done by selecting objects that are far away from each other and may achieve poor pruning rates. Multiple references prune the same set of objects that are near the hull. There are redundant references, since an object can be pruned by one of them. Furthermore, none of the references prune the objects that are far from the hull.

Figures 3-2 and 3-3 illustrate this problem. In Figure 3-2, $ref_1$ and $ref_2$ are the references selected using the Omni method. Object $p$ is close to $ref_1$ and far away from $ref_2$. For query $q$, $p$ can be pruned by both $ref_1$ and $ref_2$, representing a “wasted” reference. On the other hand, objects inside the hull will not be pruned at all, as illustrated in Figure 3-3. Here, the bounds obtained for the query $q$ and object $d$ using the two Omni references do not remove $d$ from the candidate set. Had the reference $rand$ in this figure been selected instead, then it would have been possible to prune $d$. It is essential to select the references so that each reference is
able to prune a different subset of the database. A similar approach for selecting references in content-based image retrieval has been proposed by Vleugels et al. [94].

Reiner et al. [57] proposed a spacing-based selection of references. The basic idea is to add database objects to the index based on two criteria. The first criterion is called *spacing* where references with small variance of spacing are supposed to have have discriminative power over the database. The second criterion, *correlation*, reduces the redundancy of the references by using the linear correlation coefficients among the reference objects. It tries to minimize the number of false positives. As soon as either the variance of spacing of one object or the correlation of a pair of objects exceed a certain threshold, a reference object is replaced by a randomly chosen new reference object. However, there were no guidelines on selecting the spacing and correlation thresholds and the number of reference objects.

Brisaboa et.al. [21] proposed Sparse Spatial Selection (SSS) method that adapts to the updates in dynamic databases. An object becomes a reference, if it is located at more than a fraction of the maximum distance with respect to all of the existing references. The number of references selected depends on the intrinsic dimensionality of the underlying database. This approach is dynamic in nature and selects references that are not outliers. But it can result in redundancy, i.e. reference that prune same set of database objects like Omni.

Bustos et.al. [23] present a criterion to compare the efficiency of two sets of references of the same size. They select a reference set that maximizes the mean of query-to-object distances. Different reference selection strategies were proposed: a) *Selection* that selects from a set of $N$ random sets of references one that maximizes the mean b) *Incremental* selects a new reference that maximizes the mean of the current set of references and c) *Local Optimum* is an iterative strategy that starts with a random reference set and in each iteration replaces one that makes less contribution to the mean. They select references that are far from the other database objects. This approach had the same drawbacks as that of Omni and does not always result in good set of references.
The Approximating and Eliminating Search Algorithm (AESA) [72] is simply a matrix with the pre-computed distance computations between every pair of objects in the database. When answering a query, AESA computes the distance between the query and an arbitrarily set of objects, and used these distances to generate the candidate set. This method has high storage and pre-processing costs. The Linear AESA (LAESA) [64] solves this problem by selecting a small subset of the objects called as Base Prototypes (BP) as references and pre-computing the distances from the other objects. Its efficiency depends on the number of selected objects as BP and their location with respect to other database objects. The objects are simply linearly traversed and those that cannot be eliminated after considering the references are compared directly against the query object. This method has high CPU costs like AESA.

### 3.3.2 Reference-based Hierarchical Indexing Methods

Traditional tree-based indexing methods impose a hierarchy on the reference objects that guides the order of distance computations. These methods use $O(\log n)$ references, since they partition the space hierarchically using one or more references at each level.

The M-Tree [29] is a height-balanced tree with the data objects in its leaf nodes. It aims at providing dynamic capabilities and good I/O performance with fewer distance computations. A set of reference objects are selected at each node and objects closer to the reference objects are organized into a subtree rooted by that reference. Each reference stores its covering radius. The search algorithm recursively searches the nodes that cannot be pruned using the covering radius criterion. The main problem with M-Tree is that it has poor selectivity at higher dimensions.

A variation of the M-Tree, the Slim-Tree [86], reduces the amount of overlap between the tree nodes. It uses the slim down algorithm which leads to better tree. It also makes use of faster splitting algorithm based on the minimal spanning tree. It makes use of \textit{chooseSubtree} algorithm for the slim-tree which leads to tighter trees, thus have fewer disk pages, and faster retrievals. These tree-based structures are height balanced and attempt to reduce the height of the tree at the expense of flexibility in reducing the overlap between the nodes. This constraint was relaxed in the Density Based Metric Tree (DBM-Tree) [92] by reducing the overlap between nodes in
high-density regions, resulting in an unbalanced tree. An optimization algorithm that reorganizes
the objects in the nodes improving the performance is presented. The DF-Tree [85] selects a
global set of representatives in order to prune candidate objects when answering queries. Its
pruning with respect to number of distance computation is very high. But it is less efficient than
the slim-tree in number of disk access.

The Vantage-Point tree (VP-tree) [99] partitions the data space into spherical cuts by
selecting random reference points from the data. The goal is to reduce the number of distance
calculations to answer similarity queries in the VP-tree. A variation of the VP-Tree called the
MVP-Tree [20] uses more than one reference at each level. Using many representatives the
MVP-tree requires lesser distance calculations to answer similarity queries than the VP-tree.

Burkhard-Keller Tree (BKT) [22] provides interesting techniques for partitioning a metric
database where the recursive process is materialized as a tree. The first technique partitions a
database by choosing a reference from the database and grouping the objects with respect to
their distance from it. The second technique partitions the original database into a fixed number
of subsets and chooses a reference from each of the subsets. The reference and the maximum
distance from the reference to a point of the corresponding subset are also maintained to support
the nearest neighbor queries.

The Fixed-Queries Tree (FQT) [5], improves BKT, where all the references stored in
nodes of the same level are the same. The actual objects are stored in the leaves. The advantage
of this approach is that some comparisons between the query and nodes are saved along the
backtracking that occurs in the tree. The Fixed-Height FQT (FHQT) [5] has all leaves at the same
depth thus making some leaved deeper than necessary. This makes sense because we may have
already performed the comparison between the query and reference of an intermediate level.
But has space limitations similar to FQT. The Fixed Queries Array (FQA) [27] is a compact
representation of the FHQT. It is similar to traversing the leaves of a FHQT with fixed height, left
to right. Using the same memory, FQA is able to use more references than FHQT improving its
efficiency.
The Spaghettis [26] present a dynamic reference based method. During the pre-processing, the distance between the reference to database objects are computed. It reduces the CPU cost while retaining the array structure of FHQT by sorting each array and saving the permutations with respect to the preceding array. All these methods when applied to continuous distance functions lose their linear CPU time. Except M-Tree based methods, all other structures are for static databases and are not suitable for large databases.

Pivoting M-Tree (PM-Tree) [80], a hybrid between the two categories, is an extension of the M-Tree. It combines the hierarchy used by tree-based methods with a distance-matrix based method. The result is a flexible metric access method providing more efficient similarity search performance than the M-Tree. But pivot selection requires a part of the database to be known in advance. A combination of PM-Tree and a slim down algorithm makes an efficient metric access method.

3.4 String Search Methods

In this section, we discuss some of the existing methods that are specific for string database searches. In string databases, there are two types of string alignment: Global Alignment and Local Alignment. Global alignment attempt to align all residues in two strings and are useful when the strings are similar and have approximately same size. A dynamic programming based algorithm was proposed by Needleman and Wunsch [67]. Local Alignment are more useful for dissimilar strings that can contain regions of similarity. The Smith-Waterman Algorithm [81] is a dynamic programing based solution local alignment of two strings. A number of index structures have been developed to reduce the cost of searches in string databases. They can be classified under three categories: \( k \)-gram indexing, suffix trees, and vector space indexing.

A \( k \)-gram is a string of length \( k \), where \( k \) is a positive integer [40]. \( k \)-gram based methods look for the shortest substrings that match exactly; these strings are then extended to find longer alignments with mismatches and inserts/deletes. \( k \)-grams are usually indexed using hash tables. Two of the most well known genome search tools that use hash tables are FASTA [69] and
BLAST [2]. The performance of these tools deteriorates quickly as the size of the database increases.

Suffix trees were first proposed by Weiner [95] under the name *position tree*. Later, efficient suffix tree construction methods [61, 88] and variations [34, 47, 52, 60] were developed. The suffix tree for the string $S$ is defined as a tree where each path from the root to a leaf node represents a suffix of $S$. Its time complexity for building a suffix tree is $O(length(S))$. However, there are two significant problems with the suffix-tree approach: (1) suffix trees manage mismatches inefficiently, and (2) suffix trees are notorious for their excessive memory usage [66]. The size of the suffix tree varies between 10 to 37 bytes per letter [32, 47, 52, 62].

Suffix Array [60] is basically a lexicographically sorted list of all suffixes of the string $S$. The suffix array was developed to reduce the space consumption in a suffix tree. A binary search on the list gives the matching suffixes.

A number of index structures have been developed to function in vector space, such as SST [39] and the frequency vectors [49, 50]. The frequency vector of a string stores the number of letters of each type in that string. This method computes a lower bound to the distance between two strings using the frequency vectors corresponding to the two strings. It uses this lower bound to eliminate unpromising strings. However, as the query range increases frequency vectors perform poorly.
CHAPTER 4
REFERENCE-BASED INDEXING FOR STATIC DATABASES

An important issue overlooked by the existing methods is that the performance of reference-based indexing can be improved by selecting references that have a significant number of objects close to and far from them. The key symbols used throughout this chapter are summarized in Table 4-1.

Table 4-1. Summary of symbols and definitions.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>Database of objects</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Similarity threshold</td>
</tr>
<tr>
<td>$q$</td>
<td>Query object</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of objects in the database</td>
</tr>
<tr>
<td>$V$</td>
<td>Reference set</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of references in $V$</td>
</tr>
<tr>
<td>$t$</td>
<td>Time taken to compare a pair of database objects</td>
</tr>
<tr>
<td>$Q$</td>
<td>Sample query set</td>
</tr>
<tr>
<td>$G_i$</td>
<td>Gain in pruning from reference $v_i \in V$</td>
</tr>
<tr>
<td>$f$</td>
<td>Size of a sample database</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Accuracy of the estimated maximum gain</td>
</tr>
<tr>
<td>$\psi$</td>
<td>Probability of the estimated maximum gain</td>
</tr>
<tr>
<td>$\mu$ and $\sigma$</td>
<td>Mean and variance of the distribution of gains</td>
</tr>
<tr>
<td>$h$</td>
<td>Size of a sample candidate set</td>
</tr>
<tr>
<td>$k$</td>
<td>Subset of references</td>
</tr>
<tr>
<td>$I_C$</td>
<td>Index construction time</td>
</tr>
</tbody>
</table>

4.1 Maximum Variance

The variance of the distribution of distances of a reference to other objects is a good indicator of the spread of objects in the database around that reference, and the first heuristic uses this observation. The Maximum Variance heuristic assumes that queries follow the same distribution as the database objects. It selects a reference set that represents the object distribution of the database. Each new reference prunes some part of the database not pruned by the current objects in the reference set.

This suggests an algorithm for choosing reference points. For example, in Figure 4-1, points in two-dimensional space represent the database objects. The database is given by the objects
{a, b, c, d, e, f, g}. Objects are represented by shaded points. e and c are the references selected by Maximum Variance. Objects f, g are close to e and b is far from e. Object d is close to c and object a is far from c. The edit distance between two objects corresponds to the distance between the points representing them. Object e has the highest variance of distances. So e is chosen as the first reference. Objects close to e (objects f and g) and far from it (object b) can be pruned using e as the reference. The objects not pruned by e (objects a, c and d) remain in the candidate set. A object from the candidate set having the next highest variance, c, is selected as the next reference. Reference c can remove the object d close to it and a far from it from the candidate set. This process is repeated until all of the references have been assigned.

Let $L$ denote the length of the longest object in $S$. For a object $s_i \in S$, $\mu_i$ and $\sigma_i$ are the mean and variance of its distances to other objects in $S$. A cut-off distance, $w$, is computed to measure the closeness of two objects. $s_j (s_j \in S)$ is close to $s_i$ if $ED(s_i, s_j) < (\mu_i - w)$ and $s_j$ is far away from $s_i$ if $ED(s_i, s_j) > (\mu_i + w)$. $w$ is computed as a fraction of $L$, given by $w = L \cdot perc$, where $0 < perc < 1$. $s_j$ is not considered as a potential reference if $ED(s_i, s_j) < (\mu_i - w)$ or $(\mu_i + w) < ED(s_i, s_j), \exists s_i \in V$. A large value for $perc$ will include objects that are close to or far away from the existing references. This results in objects being
pruned by multiple references. A small value for \(perc\) can remove promising references, resulting in a small number of references. Experimental results show that \(perc = 0.15\) is a good choice.

### 4.1.1 Algorithm

| **input** | Object database \(S\), with \(|S| = N\). |
|-----------|------------------------------------------|
|           | Number of references \(m\).             |
|           | Cutoff percentage \(perc\).             |
|           | Length of a object \(L\).               |
| **output**| Set of references \(V = \{v_1, v_2, \ldots, v_m\}\) |

1. \(V = \{\}\). /* Initialize */
2. **for** \(s_i \in S\) **do**
   3. Select sample set of objects, \(St \subset S\).
   4. Compute \(D_i = \{ED(s_i, s_j) \mid \forall s_j \in St\}\).
   5. Compute mean \(\mu_i\) and variance \(\sigma_i\) of the distances in \(D_i\).
3. **end**
4. \(w = L \cdot perc\).
5. Sort the \(N\) objects in descending order of their variances.
6. **while** \(|V| < m\) **do**
   7. \(V = V \cup s_1\).
   8. \(S = S - \{s_j\}, \forall s_j \in S\) with \(ED(s_1, s_j) < (\mu_1 - w)\) or \(ED(s_1, s_j) > (\mu_1 + w)\). /* Remove objects close to or far away from the new reference */
   9. **end**
10. Return of set of reference objects, \(V\).

**Algorithm 1**: Maximum Variance Algorithm.

Algorithm 1 presents the algorithm in detail. For each object \(s_i \in S\), a sample database \(St, St \subset S\) is selected in Step 2.a and the set of distances \(D_i = \{ED(s_i, s_j) \mid \forall s_j \in St\}\) are computed (Step 2.b). The mean \(\mu_i\) and variance \(\sigma_i\) of the distances in \(D_i\) are computed in Step 2.c. The distances are then sorted in descending order of their \(\sigma_i\) values (Step 4). Then, the following are done repeatedly until the required number of references are obtained. The object \(s_1\) with maximum variance is selected as the next reference and added to \(V\) (Step 5.a). Then the objects from \(S\) that are close to or far away from the new reference \(s_1\) (Step 5.b) are removed. Steps 5.a and 5.b are repeated until there are enough number of references, i.e. \(|V| = m\). Each iteration of the algorithm selects a new reference that is neither close to nor far away from the existing references.
4.1.2 Computational Complexity

Each candidate reference is compared with all objects in the sample, \( S' \) (Step 2). This requires \( O(N|S'|) \) distance computations. Each distance comparison takes \( O(L^2) \). Sorting the variances of \( N \) objects in Step 3 takes \( O(N \log N) \). Steps 5.a-5.c take \( O(mN) \) time in the worst case. Thus the overall time of the algorithm is \( O(NL^2|S'| + N \log N + mN) \).

4.2 Maximum Pruning

This section describes the second approach for choosing the reference set. The method, called *Maximum Pruning*, combinatorially attempts to compute the best reference set for a given query distribution.

A purely combinatorial approach which tests all possible combinations of references in order to maximize performance over a given query distribution would be prohibitively expensive. Exhaustively testing all possible combinations of \( m \) references from \( S \) takes \( O(C^N_m \times N \times |Q|) \) time, where \( C^N_m \) is \( \binom{N}{m} \). This is due to the fact that a purely combinatorial approach would need to consider all possible reference sets, one-at-a-time, and for each it would need to compute the pruning power with respect to \( Q \). The method could perhaps be improved a bit by making use of the fact that many of the reference sets to be tested would be overlapping, but it seems impossible to reduce the complexity of an exact computation below \( O(C^N_m) \). To speed up this computation, a greedy solution to this problem is proposed. In order to speed the greedy solution even further, sampling-based optimizations are considered in Section 4.2.3.

4.2.1 Algorithm

Every object in the database is considered as a candidate reference. An initial set of references is selected. Then the references in the set are refined iteratively. At each iteration, an existing reference replaced with a new reference if this modification improves pruning with respect to the sample query set \( Q \). The algorithm terminates when the reference set cannot be improved.

Algorithm 2 presents the Maximum Pruning algorithm. \( S, Q, \) and \( m \) are given as input. Here, \( \text{PRUNE}(Vt, q, s) \) returns true if one of the references in \( Vt \) can prune \( s \). \( \text{MAX}(P) \) and

45
**Algorithm 2**: Maximum Pruning Algorithm.

MAX($G$) returns the maximum of the values in $P$ and $G$ respectively. The reference set $V$ is first initialized as the top $m$ references obtained using the Maximum Variance (MV) method (though this is not a requirement, since one can start with a random initial reference set). The MV method selects as references those database objects having a high variance in pairwise distances with other database objects.

The Do-While loop replaces one existing reference with a better reference during each iteration. An iteration of this loop starts by initializing the array $G$ to zero. Each entry $G[i]$ of $G$ shows the amount of pruning gained by including the $i$th candidate reference in the reference set. The term gain is used to denote the amount of improvement in pruning. Steps 2.a - 2.d iterate
over all candidate references \( v_i \) and the gain obtained by replacing an existing reference with \( v_i \) is computed. This is done as follows: the total number of objects pruned using an existing reference set for all queries in \( Q \) is computed in Step 2.a. The array \( P \) is initialized in Step 2.b. Each entry \( P[a] \) of \( P \) denotes the number of objects pruned after replacing the \( a \)th existing reference with \( v_i \). Step 2.c iterates over all existing references. At each iteration, an existing reference is replaced with \( v_i \) and the total number of objects pruned for all queries in \( Q \) is computed. The largest possible gain obtained by subtracting the number of objects pruned using the original reference set from that of the best possible new reference set is computed in Step 2.d. If there is no gain, then the algorithm terminates, returning the current reference set (Step 3). Otherwise, the reference set is updated using the new reference that gives the best possible gain (Steps 4 to 6).

### 4.2.2 Computational Complexity

The algorithm needs access to the distances between all pairs of objects in \( S \) (Step 2). This requires \( O(N^2) \) comparisons. Note that this a one time cost, and is not incurred at each iteration of the Do-While loop. The PRUNE function requires the distances between all (query, object) pairs. The pre-computation of these distances requires \( N|Q| \) comparisons. So, the total number of comparisons is \( O(N^2 + N|Q|) \). Step 2 has to consider \( O(N^2) \) pairs of objects. For each pair it computes the gain for all queries in \( Q \) after replacing objects in the reference set one by one. This takes \( O(m|Q|) \) time since a new reference can replace any of the \( m \) references (Step 2.c). Thus, the overall time taken by this algorithm is \( O(N^2m|Q|) \) for each iteration of the Do-While loop. There can be at most \( N \) iterations, leading to the worst case complexity \( O(N^3m|Q|) \).

### 4.2.3 Sampling-based Optimization

Although the Maximum Pruning algorithm in Section 4.2.1 is faster than a purely combinatorial approach, it is still impractical for large databases. To address this problem, two sampling-based optimizations to improve the complexity of the algorithm by reducing the number of (object, reference) pairs processed are proposed. The first optimization reduces the number of objects and the second reduces the number of references.
input: Database $S$, Sample queries $Q$, $\epsilon = q$.

output: $G[i] \forall v_i \in S$.

1 for each $v_i \in S$ do
  2 $\hat{g}_1 = \hat{g}_2 = 0$. /* Initialize total and estimate for 2nd moment */
  3 $\alpha = 0$. /* Initialize sampling fraction */
end

repeat
  6 Select a random $s_j \in S$, where $s_j \neq v_i$.
  7 newGain = GAIN($v_i, s_j, Q$).
  8 $\hat{g}_1 += \text{newGain}$; $\hat{g}_2 += newGain^2$.
  9 $\alpha = \alpha + 1/|S|$.
 10 $\hat{\sigma}^2 = \hat{g}_2/\alpha^2 - \hat{g}_1^2/(|S|\alpha^3)$.
until $2\alpha\hat{\sigma}/\hat{g}_1 > \epsilon$.

$G[i] = \hat{g}_1/\alpha$.

Algorithm 3: Estimation of Gain.

4.2.3.1 Estimation of gain

The number of objects that must be considered when computing the gain associated with a new reference point is reduced in the first optimization. This algorithm replaces Step 2 of the Maximum Pruning algorithm in Algorithm 2. The gain is estimated based on a small sample of the database rather than the entire database. One of the most important technical considerations in the design of this algorithm is how to decide whether the gain estimate is accurate enough based upon the sample.

Algorithm 3 presents the sampling algorithm in detail. $S$, $Q$ and $\epsilon$ are given as input. It returns the total gain obtained by replacing an existing reference with a new reference for all possible new references. For each candidate reference $v_i$, a random object $s_j \in S$ is selected at every iteration (Step 3.a). The gain by using $v_i$ as a reference for $s_j$ for $Q$ is computed in Step 3.b. The gain is computed as follows. Steps 2.a to 2.c of the Maximum Pruning algorithm in Algorithm 2 are executed to compute the total pruning achieved with respect to $s_j$ by replacing each existing reference with $v_i$. Then the gain is given by the best pruning over all possible replacements. The total gain seen as well as the total squared gain seen (which can be used to estimate the second moment of the gains that have been sampled) is updated in Step 3.c. The
sampling fraction is updated in Step 3.d, and an estimate for the variance of the gain is computed
in Step 3.e. The algorithm terminates when the desired accuracy is reached. The accuracy of
the gain estimate is guessed by making use of the Central Limit Theorem, which implies that
errors over estimated sums and averages of this type are normally distributed. Since 95% of the
standard normal distribution’s mass is within two standard deviations of zero, if the current gain
estimate $\hat{g}_1/\alpha$ is treated as the true gain and terminate when twice the relative standard deviation
is less than $\epsilon$, it is assured that the relative error is less than $\epsilon$ with 95% probability.

For an average sample size of $f$ with $f \ll N$, this approach reduces the complexity to
$O(N^2 fm|Q|)$. This is because it iterates over $f$ objects rather than all $N$ objects while computing
gain.

4.2.3.2 Estimation of largest gain

The Maximum Pruning algorithm uses all database objects as candidate references when
selecting the reference set. The number candidate references processed in each iteration is
reduced by the second optimization with the help of sampling. The goal is to use a small subset
of the database as the candidate references, and yet achieve pruning rates close to Maximum
Pruning.

Formally, the problem is defined as follows. Let $G[i]$ be the gain that can be achieved
by including the $i$th reference in the reference set. Let $G[e]$ be the largest gain (i.e., $e =
\arg\max_i\{G[i]\}$). Given two parameters $\tau$ and $\psi$ where $0 \leq \tau, \psi \leq 1$, the candidate refer-
ence set has to be sampled to ensuring that the largest gain from this sample is at least $\tau G[e]$ with
probability $\psi$.

Since $G[e]$ is not known in advance, the Type-I Extreme Value Distribution (also known
as the Gumbel distribution [41]) can be used to estimate its value. This is done as follows. Lets
assume that each $G[i]$ is produced by sampling repeatedly from a normally distributed random
variable. The first step is to determine the mean and standard deviation of this variable. To do
this, a sample set of candidate references is selected and the mean $\mu$ and the standard deviation $\sigma$
of the gains are computed for the sample. These are taken as the mean and standard deviation of the underlying distribution.

Since the values in $G$ are assumed to be the samples from a normal distribution, the largest gain $G[e]$ is known to have a Gumbel distribution whose parameters can be computed using $\mu$ and $\sigma$. Let $N$ and $t$ be the number of candidate references and the sample size, respectively. The two parameters of the Gumbel distribution, referred to as location parameter $a$ and a scale parameter $b$, are computed as follows:

$$a = \sqrt{2 \log N}$$

$$b = \sqrt{2 \log N} - \frac{\log \log N + \log 4\pi}{2 \sqrt{2 \log N}}$$

The mean of the corresponding Gumbel distribution is then calculated as:

$$\hat{\mu} = \sigma \left[ -\frac{-\log_e (-\log_e (0.5))}{a} + b \right] + \mu$$

This tells exactly what the expected value of the gain associated with the best reference object is. Thus, the sampling stops when the best gain (with high probability) is almost good enough; that is, when it is at least $\tau \hat{\mu}$. To compute how many samples are needed, let $P(x < \tau \hat{\mu})$ be the probability that the gain of a random reference $x$ is less than $\tau \hat{\mu}$. This probability can be calculated from the distribution of $G[i]$. The probability that the gain of all the $w$ randomly selected references are less than $\tau \hat{\mu}$ is $P(x < \tau \hat{\mu})^w$. Solving the inequality:

$$1 - P(x < \tau \hat{\mu})^w \geq \psi$$

gives the number of samples needed (denoted by $w$) as:

$$w \leq \frac{\log_e (1 - \psi)}{\log_e (P(x < \tau \hat{\mu}))}.$$  

The experiments have used a value $\tau = \psi = 0.99$. Each iteration of the Do-While loop of the Maximum Pruning algorithm in Algorithm 2 computes the gain from random candidate references until the required accuracy is reached. In each iteration, a different sample
of candidate references is used. If the average sample size of the \( m \) iterations is \( h \), \( h \ll N \), then this optimization along with the first optimization reduces the overall complexity to \( O(Nfhm|Q|) \).

### 4.2.4 Impact of the Sample Query Set

All reference selection strategies must at least implicitly assume that queries follow a certain distribution, since the quality of a given reference is always dependent upon the query workload that is to be answered. For example, as described in the Introduction, a single reference is enough to prune all of the data in a portion of the data space that is far from most queries; additional references in such a portion of the data space are wasted. The implicit assumption behind most existing reference selection strategies seems to be that queries are distributed similar to database objects or that the queries are spread evenly throughout the space to be indexed [21, 23, 35, 64, 94].

MP differs from most selection strategies in that it makes any assumptions about the query distribution explicit. Of course, MP can easily be run by simply using the data to be indexed as the query distribution; in this case, like other methods, MP will be optimized to run on queries that are similar to database objects. However, if the queries follow different distribution or if the query distribution keeps changing, MP has the benefit that it can explicitly take into consideration these factors while creating the index.

As seen in the experiments of Sections 4.5 and 5.4, the efficiency and efficacy of MP do depend upon the size and accuracy of the training query set. However, MP is surprisingly robust in this regard, and even for small and/or a somewhat inaccurate training set, MP outperforms its competitors. Thus, it can be argued that the fact MP uses a training query set is actually a beneficial feature of the MP methodology.

### 4.3 Assignment of References

#### 4.3.1 Motivation and Problem Definition

A simple way to improve the performance of a reference-based index is to increase the number of references. For example, the number of object comparisons required by Omni [35] to
Figure 4-2. Number of comparisons for Omni with varying number of references.

answer a query over a DNA object database as a function of the number of references is given in Figure 4-2 (see Section 4.5 for details). This uses the DNA database with a query range of 8. It can be seen that initially the number of comparisons required to answer a query reduces with increase in the number of references. After 400 references, it begins to increase linearly with the number of references. Thus, the “optimal” number of references for this particular example is in the hundreds.

Unfortunately, due to memory constraints, selecting and assigning 400 references to each and every object in a database is not always a practical solution. For example, the human genome (with 3 billion base pairs) contains 30 million objects of length 100. From the cost model, the main memory storage of an index that contains 400 references for each of these objects would require about 90GB of main memory. While 90GB of RAM may be feasible, it is certainly at the upper end of what would be acceptable. For an even larger database (or even in the case of the human genome if one were to index the substring starting at each and every base pair) the memory requirements quickly become unmanageable.
This section proposes a new strategy that selects only those references that can help in pruning a given database object. This allows to construct an index whose performance is equivalent to an index that makes use of a much larger set of references, but whose memory costs are much smaller because only a small number of reference-to-object distances are stored for each database object.

Formally, given a set of $m$ references, the goal is to assign a set of $k$ references ($k < m$) to each database object such that these remove $s$ from the candidate set of as many queries as possible. When $m$ is much smaller than the database size, this strategy improves the performance with little increase in the storage cost.

### 4.3.2 Algorithm

It is important to note that the task of assigning the references to the database objects is orthogonal to actually choosing the references. The set of $m$ initial references ($V$) can be chosen in many different ways; it can be chosen using the Maximum Pruning algorithm, or using one of the other selection strategies. The only assumption here is that the size of the set $V$ is substantially larger than the number of references.

Algorithm 4 shows the algorithm for assigning references to each database object. $Q$, $S$, the potential reference set $V$, and the number of references per object, $k$, are given as input. The algorithm returns a mapping from each database object to $k$ references in $V$. For each database object $s$, the algorithm iteratively maps one reference until $k$ references are mapped. This is done as follows. An array $V\text{count}$ is maintained, where $V\text{count}[i]$ gives the number of queries for which an object is pruned using the reference $v_i \in V$. The algorithm selects the reference $e \in V$ for which $s$ is pruned for maximum number of queries (Steps 3.b-3.c). $e$ is removed from $V$ and mapped to $s$ in Steps 3.e and 3.f. The queries for which $s$ is pruned using $e$ are removed from the query set in Step 3.g. This is done to ensure that the new references are selected only to improve the queries for which $s$ can not be pruned using the existing references. Each reference costs $|Q|$ extra object comparisons. This is because each query needs to be compared with all the references. Thus, a reference is useful only if it prunes a total of more than $|Q|$ objects for all
Algorithm 4: Algorithm to Assign References to Objects.
the queries in $Q$. If the total gain from a reference is not greater than $|Q|$, it is removed from the reference set (Step 4). The reference sets of objects which have less than $k$ references are then updated with new references from $V$ (Step 5).

**4.3.3 Computational Complexity**

During index construction, the algorithm needs access to the distances between all (query, reference) pairs. This takes $O(tm|Q|)$ time, where $t$ is the time taken for one object comparison. If the selection strategy is Maximum Pruning, then the pre-computed query-reference distances are used. In each of the $k$ iterations, Step 3.b of the algorithm can process up to $O(m|Q|)$ query-reference pairs. Thus the overall time taken by the algorithm is $O(Nmk|Q|)$.

**4.4 Search Algorithm**

Sections 4.1 and 4.2 discussed how to find references and how to map them to database objects (Section 4.3). This section describes how to use the mapped references to answer range queries.

**4.4.1 Algorithm**

Let $S$ be the database of $N$ objects and $V$ be the reference set. The set of $k$ references mapped to $s_i$ is given by $E_i, \forall s_i \in S$. Here, $E_i \subseteq V$. The edit distances $ED(v, s_i), \forall s_i \in S$ and $\forall v \in E_i$ are pre-computed. This is a one time cost for the database. For a query $q \in Q$ and range $\epsilon$, the edit distances from $q$ to the reference objects are computed, i.e. $ED(q, v_i) \forall v_i \in V$. For each $(q, s_i)$, the lower and upper bounds $LB$ and $UB$ are computed as given in Chapter 1. Depending on the $LB$, the $UB$, and the $\epsilon$, $s_i$ is inserted into one of the two sets $Result$ set and $Candidate$ set as follows. If $UB \leq \epsilon$, $s_i$ is inserted into the result set. If $LB \leq \epsilon \leq UB$, $s_i$ is inserted into the candidate set. Otherwise, $s_i$ is pruned. Once the candidate set is determined, the actual object comparison between $q$ and all the objects in the candidate set to filter out false positives is performed.

Given an object database $S$ with $N$ objects, the selection strategy selects $m$ references. The assignment strategy maps each object with $k, k \leq m$, references. For each object $s$ and its reference $v_i \in V$, its mapping is of the form $[i, ED(s, v_i)]$. The $Nk$ edit distances between the
objects and references mapped to the objects are stored in a file. The selection strategy, mapping
database objects \( s \in S \) to references and computing the \([s, v_i]\) edit distances are all one-time
costs for a object database.

During database search, references and reference to object edit distances are loaded into
the memory. With the average object size as \( z \) bytes and four bytes for an integer, this requires
\((8Nk + mz)\) bytes of memory. Here 8 bytes are used to store each of the \( Nk \) mappings. With
the increase in \( m \), the objects can be assigned better references. However this will increase the
number of query to reference computations. Hence \( m \) is restricted to a fraction of the database
size in the experiments.

### 4.4.2 Computational Complexity

Given a query set \( Q \), the edit distances of all \([\text{query, reference}]\) pairs are computed. This
involves \( m|Q| \) object comparisons. For every \([\text{query, object}]\) pair, all \( k \) references of the object
are compared to compute the lower and upper bounds. This takes \( O(Nk|Q|) \) time. If \( C_m \) is the
average candidate set size for \( Q \) using the \( m \) references, it takes \( C_m|Q| \) object comparisons to get
the final results. For an average object length of \( L \), the object comparison takes \( O(L^2) \). Thus, the
overall time taken by the search algorithm is given by, \( O((m + C_m)|Q|L^2 + Nk|Q|) \).

### 4.5 Experiments

This section compares the performance of different methods for indexing based on the
number of object comparisons performed.

<table>
<thead>
<tr>
<th>Assignment Strategy</th>
<th>Selection Strategies</th>
<th>Maximum Variance</th>
<th>Maximum Pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same References</td>
<td>MV-S</td>
<td></td>
<td>MP-S</td>
</tr>
<tr>
<td>Diff. References</td>
<td>MV-D</td>
<td></td>
<td>MP-D</td>
</tr>
</tbody>
</table>

The proposed methods that have been implemented are listed in Table 4-2. For the proposed
algorithms, two different reference-to-object assignment strategies are considered. The first one
is the traditional way of assigning all references to the database objects (MV-S, MP-S). The
second strategy is the proposed approach of increasing the reference set and assigning different
subset of references to each database object (MV-D, MP-D). For MV-D and MP-D, 200 candidate references are selected (i.e. \( m = 200 \)) unless otherwise stated and the top \( k \) \((k < m)\) references are used for each database object to index that object. The methods are compared with many existing methods: 1) Omni [35], 2) the Sparse spatial selection strategy (Sparse) proposed by Brisaboa et.al. [21], 3) the random (BRAND) and 4) incremental (BINC) reference selection strategies proposed by Bustos et.al. [23]. The following four types of databases have been used in the experiments:

**DNA Database:** The *Escherichia Coli* (*E.Coli*) (K-12 MG1655) genome from GenBank [10] ([ftp://ftp.ncbi.nih.gov/genbank/](ftp://ftp.ncbi.nih.gov/genbank/)) has been used. The alphabet size of a DNA object is 4 (A, C, G, and T). Four databases of non-overlapping objects of lengths 25, 50, 100 and 200 have been created. Each database is obtained by chopping the E.Coli database into 20,000 non-overlapping objects. Databases of different sizes containing 5,000, 10,000, 15,000 and 30,000 non-overlapping objects, each of length 100, are also created similarly.

**Protein database:** Protein objects in the Eukaryota kingdom of organisms are used for this database. They are obtained from SwissProt [7] ([ftp://ftp.ebi.ac.uk/pub/databases/swissprot/](ftp://ftp.ebi.ac.uk/pub/databases/swissprot/)). The alphabet size is 20. A set of 4,000 objects having up to 500 amino acids is selected randomly.

**Text database:** A text database from 33 randomly selected books from the Gutenberg project ([http://www.gutenberg.org/](http://www.gutenberg.org/)) is created. The alphabet consists of 36 alphanumeric characters. The database contains 8,000 objects of length 100.

**Retinal Images:** Color histograms of images of retinal tissues from the UCSB BioImage database are used ([http://bioimage.ucsb.edu/](http://bioimage.ucsb.edu/)). The database consists consists of histograms of 3932 images. The images of retina are obtained under different conditions corresponding to different stages of a biological process [16, 59]. All comparisons are made using EMD.
In order to select the queries and the query workload used to train the algorithms, the following protocol is used. For each object database, 200 objects from different parts of the same species/database are selected. For the image database, 200 images are selected randomly from the database. 100 are used as sample queries for MP and another 100 are used as query objects. Note that the query and database objects for the DNA databases are taken from different parts of the same species and they do not overlap. Similarly for the protein data, the query objects are taken from proteins from a different part of the database without any overlap. For the DNA database, the mean distance between the query and the database objects is 56 and for the protein data, the mean distance is 192. Three additional query sets tested in Section 8.2.3 have also been created: one from each of the organisms *Danio Rerio* (Zebrafish), *Mus Musculus* (Mouse) and *Heliconius Melpomene* (Butterfly). Each query set contains 100 objects of length 100 and each is selected randomly from its organism.

The experiments ran on an Intel Pentium 4 processor running Linux with 2.8 GHz clock speed and 1GB of main memory.

4.5.1 Effect of the Parameters

This section presents experimental results of the proposed methods under different parameter settings.

4.5.1.1 Impact of $m$

The goal of this experiment is to understand the behavior of MP-D for different cardinalities of the reference set, $|V| = m$. Three databases have been used, the DNA object, protein and image databases. The number of references is 32. The query range for the DNA database is 8. It is 240 for the protein database and 16% for the image database.

The number of object comparisons for different reference set cardinalities for the three databases are given in Figures 4-3 to 4-5. For the DNA database, up to $m = 200$, the number of object comparisons reduces at a fast rate. From $m = 200$ to 300, there is very little improvement in performance. For $m > 300$, the number of object comparisons increases. This is due to increase in number of query-reference distance computations. For the image database, the
Figure 4-3. Number of comparisons for MP-D for DNA object database for different values of $m$.

Improvements obtained are only up to $m = 200$. The number of comparisons increases slowly from $m = 200$ to 300 and for $m > 300$, the rate of increase is fast. From these results, it can be concluded that using an $m$ value in the low hundreds is a good choice, since this gives good performance and allows for reasonable index construction times.

One question is How can one determine the best value of $m$ if the index construction time is completely ignored? Figures 4-3 to 4-5 show that the number of object comparisons follow a U-shaped curve. This is indeed intuitive: For small $m$, the number of possibilities for selecting different references is small. Thus, the pruning rate drops and the number of comparisons with unpruned candidates increases. For large $m$, the candidate size decreases. However, the number of comparisons with the references (i.e., $m$) increases. In other words, as $m$ increases, the benefit gained by pruning more candidates becomes less than the cost of comparing the query to the reference set. Since this curve follows a U-shape, the best value of $m$ can be determined by either using binary search over $m$ or by adopting the Newton-Rhapson method.
Figure 4-4. Number of comparisons for MP-D for protein database for different values of $m$.

Figure 4-5. Number of comparisons for MP-D for image database for different values of $m$. 
4.5.1.2 Impact of query set size, $|Q|$ 

The goal of this experiment is to understand the behavior of MP-D for different cardinalities of the sample query set, $|Q|$. The DNA object database of 30,000 objects has been used. The number of references is 32 and query range $s$ 16.

The time taken to construct the index using MP-D for different cardinalities of the sample query set is given in Figure 4-6. Time taken to create the index using MP-D for different values of $|Q|$. It can seen that with the increase in the size of sample query set, the index construction time increases almost linearly.

The number of comparisons needed is given in Figure 4-7. Number of comparisons for MP-D for different values of $|Q|$. With increasing cardinality of the training query set, the number of comparisons reduces. From $|Q| = 200$ to 1000, there is slight improvement in the performance. Given that the cost of building the index increases rapidly for increasing training query set (Figure 4-6), in all of the experiments $|Q| = 100$ is chosen.
4.5.2 Comparison of Proposed Methods

This section presents the experimental results comparing MP-S and MP-D under different parameter settings.

4.5.2.1 Impact of query range ($\epsilon$)

The goal of this experiment set is to understand the behavior of the proposed methods for different query ranges. It compares the performance of MV-S, MP-S, MV-D and MP-D with $\epsilon = 2$ to 32 for DNA objects and $\epsilon = 60$ to 420 for protein database. The number of references used for DNA and protein databases are 4 and 32 respectively. The plots are given in log scale.

The number of object comparisons for the DNA database is given in Figure 4-8. This increases with the query range for all four methods. For different ranges, MP-D and MV-D have lesser number of object comparisons compared to those of MV-S and MP-S. MP-D is gives the best results. For ranges up to 8, assigning different reference sets to each object results in significant reduction of object comparisons for both selection strategies. For both same and different reference sets, MP performs slightly better than MV. This is due to the fact that MP is using the knowledge of input query distribution. The results for protein database are given in
Figure 4-8. Comparison of the proposed methods for DNA database for queries with varying ranges.

Figure 4-9. MP-D outperforms others for all query ranges. For most of the query ranges, MP methods perform better than MV methods. Similar results for text database for varying query ranges have been observed. These experiments show that assigning different reference sets to each object gives better pruning results than the traditional approach of assigning same reference to all objects.

Overall, the results show that assigning different reference sets to each object gives better pruning results than the traditional approach of assigning the same references to all database objects.

4.5.2.2 Impact of number of references \( (k) \)

The goal of this experiment is to understand the behavior of our methods for different number of references, \( k \). It compares MV-S, MP-S, MV-D and MP-D, by fixing the query range and varying the number of references assigned, \( k = 2, 4, 8, 16 \) and 32. Both protein and DNA databases are used. The plots are in log scale for DNA database.
Figure 4-9. Comparison of the proposed methods for Protein database for queries with varying ranges.

The number of object comparisons for DNA database is given in Figure 4-10. For all four methods, the number of object comparisons decreases dramatically with increase in $k$. As the number of references increases from 2 to 32, the number of objects compared drops by factors of 5 to 20 between the methods MP-D and MP-S. The results for protein database is given in Figure 4-11. MP-S and MV-S compare more number of objects for all references. With the increase in the number of references, there is a gradual decrease in the number of object comparisons. MV-S strategy outperforms MP-S strategy at $k = 8, 16$ and $32$ in protein database and MP-S outperforms MV-S for all values of $k$ in DNA object database. The experiments using text database gave similar results as the DNA object database with MP-D giving the best results. This shows that with increase in number of references, the memory can be utilized better by assigning subset of references to each database object.

4.5.3 Comparison with Existing Methods

This Section compares MV-D and MP-D with Omni, FV, M-Tree [29], DBM-Tree [92], Slim-Tree [86], DF-Tree [85], Sparse [21], BRAND and BINC [23].
Figure 4-10. Comparison of the proposed methods for DNA database with a varying number of references.

Figure 4-11. Comparison of the proposed methods for Protein database with a varying number of references.
4.5.3.1  Impact of query range ($\epsilon$)

This experiment compares the behavior of the proposed method with several existing methods for different query ranges. For the protein database, the query range is varied from $\epsilon = 60$ to 420. $\epsilon$ is tested for values from 2 to 32 for the DNA database and $\epsilon$ from 2 to 32\% of the largest distance for the image database. 32 references are used for MP-D, BRAND, BINC and Omni, since these methods allow us to choose the number of references. Sparse chooses the number of references itself. The number of references used by Sparse is 586 for the DNA database, 598 for the protein database and 69 for the image database. The plots are given in log-scale for the protein and image databases.

Table 4-3. Comparison with Tree-based index structures.

<table>
<thead>
<tr>
<th>QR</th>
<th>M-Tree $I_C=50\ ms$</th>
<th>Slim-Tree $I_C=15\ ms$</th>
<th>DBM-Tree $I_C=14\ ms$</th>
<th>DF-Tree $I_C=480\ ms$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9946</td>
<td>10228</td>
<td>7313</td>
<td>8041</td>
</tr>
<tr>
<td>4</td>
<td>13426</td>
<td>14390</td>
<td>10963</td>
<td>10773</td>
</tr>
<tr>
<td>8</td>
<td>20147</td>
<td>19691</td>
<td>17507</td>
<td>19301</td>
</tr>
<tr>
<td>16</td>
<td>22865</td>
<td>21176</td>
<td>19977</td>
<td>24861</td>
</tr>
<tr>
<td>32</td>
<td>22892</td>
<td>21192</td>
<td>19997</td>
<td>25021</td>
</tr>
<tr>
<td>Omni</td>
<td>FV</td>
<td>MV-D</td>
<td>MP-D</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>228</td>
<td>247</td>
<td>205</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>1202</td>
<td>1264</td>
<td>273</td>
<td>208</td>
</tr>
<tr>
<td>8</td>
<td>12677</td>
<td>6208</td>
<td>2648</td>
<td>1126</td>
</tr>
<tr>
<td>16</td>
<td>19927</td>
<td>16088</td>
<td>18541</td>
<td>18296</td>
</tr>
<tr>
<td>32</td>
<td>20000</td>
<td>19811</td>
<td>19840</td>
<td>19836</td>
</tr>
</tbody>
</table>

The results of six existing methods along with MV-D and MP-D for the DNA object database are given in Table 4-3. $QR$ denotes the query range. $ss$ and $ms$ denote the running time in seconds and minutes respectively. Number of references for Omni, DF-Tree, MV-D and MP-D are 16. With the increase in query range, the number of object comparisons increases for all the methods. The tree based methods compare more objects than a simple sequential scan with the increase in query range. This is due to the comparison of query object with the objects in the intermediate nodes of the tree structures. For ranges 8 to 32 Omni performs more object comparison than FV, MV-D and MP-D. With the increase in the range from 2 to 8, MP-D reduces
the number of objects up to a factor of 6 to 100. Even for higher ranges, MP-D reduces the number of object comparisons up to a factor of 2. Thus MP-D outperforms all other methods for most of the query ranges.

$I_C$ denotes the index construction time. For tree-based methods it refers to the time taken to construct the tree structure. $I_C$ of Omni gives the time taken to generate 16 references using this method. $I_C$ of MP-D gives the time taken to generate 200 references for the reference set using maximum pruning method with sampling based optimizations. This includes the time taken for selecting the references and the index construction time, where the reference to data object distances are recorded. The index construction step is a one time cost for the databases. The databases will have thousands of queries in a short period of time with very little updates. Number of object comparisons form a major part of the overall computation in search performance of these databases. In the remaining experiments, Omni and FV are used for comparison as they perform the best among all competitors.

The results for the DNA database are given in Figure 4-12. Not surprisingly, with increasing in query range, the number of object comparisons increases for all the methods. MP-D has up to 40 times fewer comparisons than Omni, BRAND and BINC and up to 10 times fewer comparisons than Sparse. Due to its large reference set, Sparse has more comparisons than other methods at a range of 2. The results for range queries on the protein database are given in Figure 4-13. Sparse has more comparisons than all other methods. BINC performs slightly better than BRAND. For all query ranges, MP-D has up to two times few comparisons than the other methods. The results for the image database are given in Figure 4-14. MP-D has up to 5 times fewer comparisons than Sparse and up to three times fewer than BRAND.

4.5.3.2 Impact of number of references ($k$)

The goal of this experiment is to compare the behavior of the proposed methods with existing methods using different numbers of references. The performance of Omni, BRAND, BINC are compared with MP-D by fixing the query range for a varying number of references $k = 2, 4, 8, 16$ and 32. The DNA, protein and image databases are used. The query range is 8
Figure 4-12. Comparison with other methods on DNA database for queries with varying ranges.

Figure 4-13. Comparison with other methods on protein database for queries with varying ranges.
for the DNA database, 300 for the protein database and 8% of the maximum distance value for the image database. Since Sparse uses a fixed number of references for the static databases, the results of the other methods are compared with different numbers of references. The plots are given in log-scale.

The number of object comparisons for all four methods for the DNA database is given in Figure 4-15. BINC performs better than BRAND and Omni performs better than both BRAND and BINC. For all reference values, MP-D outperforms the other methods. As the number of references increases, the number of comparisons required by MP-D is smaller by up to a factor of 20 compared to Omni, BRAND and BINC.

The results for the protein database are given in Figure 4-16. With fewer references BINC has more comparisons than BRAND. MP-D reduces the number of comparisons by a factor of 2 compared to Omni and outperforms BRAND and BINC for all ranges of references.

The results for the image database are given in Figure 4-17. For a varying number of references, BINC requires fewer comparisons than BRAND. Omni requires more comparisons than
the other methods. MP-D has up to three times fewer comparisons than Omni and outperforms BRAND and BINC.

4.5.3.3 Impact of input queries

This experiment evaluates the proposed method using the DNA database when the distribution of queries differs from that of the sample query objects used in reference selection. Three query sets from three different species (Mouse, Zebrafish and Butterfly) which are taxonomically distant from the species E. coli are used. The number of references for Omni, BRAND, BINC and MP-D is 8. Sparse used 586 references for the DNA database.

The results are given in Figures 5-1 to 4-20. For all query ranges, MP-D outperforms the other methods, even though it has been trained on a species that is different than the query species. This suggests that MP-D is at least somewhat robust to changes in the query distribution or inaccuracies in the training distribution.
Figure 4-16. Comparison with other methods on protein database for a varying number of references.

Figure 4-17. Comparison with other methods on image database for a varying number of references.
Figure 4-18. Comparison with other methods on DNA database for queries from Heliconius Melpomene with varying query ranges.

Figure 4-19. Comparison with other methods on DNA database for queries from Mus Musculus with varying query ranges.
Next, the scalability in database size of Omni, Sparse, BRAND, BINC and MP-D is tested. Four DNA object databases with 5,000, 10,000, 15,000 and 20,000 objects each have been used. The number of references used by Omni, BRAND, BINC and MP-D is 32. The numbers of references selected by Sparse are 379, 478, 534 and 586 for the database sizes 5000, 10000, 15000 and 20000 respectively. The query range is 8. The plot is given in log-scale.

The results for the three methods are given in Figure 4-21. BRAND has the maximum number of comparisons for all database sizes. With an increase in the size of the database, MP-D outperforms all other methods. Even with its large reference set, Sparse has up to two times more comparisons than MP-D. MP-D has up to 20 times fewer comparisons than Omni, BRAND and BINC. With an increase in database size, the numbers of comparisons required by Omni, BRAND and BINC increase at a faster rate compared to MP-D.

Figure 4-20. Comparison with other methods on DNA database for queries from Danio Rerio with varying query ranges.

4.5.3.4 Scalability in database size

Next, the scalability in database size of Omni, Sparse, BRAND, BINC and MP-D is tested. Four DNA object databases with 5,000, 10,000, 15,000 and 20,000 objects each have been used. The number of references used by Omni, BRAND, BINC and MP-D is 32. The numbers of references selected by Sparse are 379, 478, 534 and 586 for the database sizes 5000, 10000, 15000 and 20000 respectively. The query range is 8. The plot is given in log-scale.

The results for the three methods are given in Figure 4-21. BRAND has the maximum number of comparisons for all database sizes. With an increase in the size of the database, MP-D outperforms all other methods. Even with its large reference set, Sparse has up to two times more comparisons than MP-D. MP-D has up to 20 times fewer comparisons than Omni, BRAND and BINC. With an increase in database size, the numbers of comparisons required by Omni, BRAND and BINC increase at a faster rate compared to MP-D.
4.5.3.5 Scalability in string length

The goal of this experiment set is to compare the behavior of the proposed method with existing methods for increasing lengths of string databases. The methods Omni, FV and MP-D have been compared using four DNA database of 10,000 strings and with string lengths $= 25, 50, 100$ and 200. The number of references used in the methods Omni and MP-D is 32 and the query range is 8. The number of string comparisons for different string lengths is given in Figure 4-22. All of the methods show reduction in the number of string comparisons with increase in the string lengths. For shorter strings, the range of 8 is large relative to the string length. In these cases the MP-D outperforms Omni and FV by a factor of 2. For long strings, the range of 8 is low. For these strings, MP-D reduces the strings comparison by a factor of 20 compared to FV and Omni. As the string length increases, the Omni outperforms FV. Separate scalability experiments on protein databases were not performed. Protein databases having string lengths up to 500 are used in the experiments given in Figures 4-13 and 4-16. These results show that the proposed methods scale well to the different lengths of protein strings.
<table>
<thead>
<tr>
<th>Sequence Length</th>
<th>Number of Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Omni</td>
</tr>
<tr>
<td>50</td>
<td>FV</td>
</tr>
<tr>
<td>100</td>
<td>MP-D</td>
</tr>
<tr>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-22. Scalability in string length.
CHAPTER 5
REFERENCE-BASED INDEXING FOR DYNAMIC DATABASES

Databases often updated by inserting new objects. For example, on the average, the GenBank database [10] (ftp://ftp.ncbi.nih.gov/genbank/) has 100,000 insertions of new DNA objects each day. Applying the MP algorithm presented in Chapter 4 to select the best reference set after each insertion is a possible approach, but it may be infeasible due to its cost. This section addresses this problem by proposing two incremental variations of the MP: the Single Pass (SP) and the Three Pass (TP) variations. Since neither algorithm re-computes the index from scratch, both must make assumptions involving the change (or lack thereof) of the various gain statistics used by the MP algorithm over time.

5.1 Overview of SP and TP

5.1.1 Basic Approach

Both SP and TP assume that MP has been used to construct an index for the data that will be included in the database when it is first made available for public use.

Each newly inserted database object is considered as a candidate reference by SP and its gain is computed by passing over the database once. If the gain from the new object is more than that of any reference in the reference set, then the reference set is updated with the new object.

In TP, the gain associated with the new object is computed using SP. Then the gains of all the objects in the database are updated based on whether they can prune the new object with their assigned references. In the final step, if the candidate reference with maximum gain is not already in the reference set, its gain is recomputed and the candidate is added to the set of references. In this method, the objects in the database are scanned three times; hence the name Three Pass algorithm.

5.1.2 Maintaining the Query Distribution

Just as MP uses a query set $Q$ in order to compute the quality of the resulting index with respect to a given query load, so do both of the SP and TP. However, since in a realistic environment the query workload may change over time, it makes little sense for the SP and TP to employ a static query workload. Instead, they use a training query set $Q$ that is allowed to evolve
over time via a reservoir sampling algorithm [93]. The reservoir algorithm is a classic, one-pass sampling algorithm with the key characteristic that at all times, the set of objects maintained by the algorithm is a true random sample of all of the objects seen thus far. The reservoir algorithm is used by both SP and TP, to maintain in an online fashion a random sample of all of the queries that have been observed thus far. This set is then used as the query set $Q$ in order to optimize the index. In this way, as the query distribution changes, both algorithms tend to include examples of both the newer and the older queries in their training set, meaning that the index can evolve over time in order to take into account an evolving query distribution.

5.2 Single-pass Algorithm in Depth

This section presents SP, first of the two incremental variations of the MP.

The gains of the objects that MP has selected as references are given as input to the SP algorithm. After each insertion of a new database object, the new object’s gain is computed by assuming that every existing database object will use it as a reference. If the gain associated with using the new object is more than that of any of the existing reference objects, then the reference set is updated with the new object and its gain.

The SP algorithm is given in Algorithm 5. The algorithm takes the existing reference set $V$, the sample query set $Q$, and gains $G[1..|V|]$, where $G[i]$ is the gain associated with the reference $V[i]$, as input. Given the new database object $X$, the algorithm first computes its gain by including $X$ in the candidate set (Steps 2.a to 2.d). This step is similar to Step 2 of the MP in Algorithm 2. The reference with minimum gain, $e$, is selected in Step 4. If the gain from the new reference is more than that of $e$, then the reference set is updated with the new reference.

5.2.1 Computational Complexity

Assuming that each distance computation takes $O(t)$ time, the total time taken for object comparisons is $O(t((N + 1) + |Q|))$. Step 2 of the algorithm replaces each reference of the database object with the new candidate reference over all sample queries. This takes $O(Nm|Q|)$. Thus the overall time taken for an insertion operation is $O(t((N + 1) + |Q|) + Nm|Q|)$. By
**Algorithm 5**: Single Pass Algorithm.

applying the first optimization of MP given in Algorithm 3 and by using a sample size of \( f \), the time complexity of SP can be reduced to \( O(t(f + |Q|) + fm|Q|) \).

### 5.3 Three-pass Algorithm in Depth

Every new object inserted into the database is considered as a candidate reference by SP, but it ignores the fact that as new database objects are inserted, the distribution of the objects in the database may change over time. The gains of the other objects in the database can also be improved if they can prune the newer objects. Thus, over time it may become desirable to include in the reference set one of the objects previously dismissed as being unsuitable.
The TP algorithm attempts to address this shortcoming. When a new database object is inserted, the first pass of this method computes the gain for the new object using the SP. In the second pass, the gains for other objects in the database are updated by considering each of them as a candidate reference to the new database object. A candidate with maximum gain is selected. In the third pass, the gain of the candidate is recomputed using its assigned references by adding it to the reference set of each database object. If this gain is greater than any of the existing references, the candidate is added to the reference set.

The TP algorithm is formally presented in Algorithm 6. The algorithm first computes gain with the newly-inserted object \( X \) as a candidate reference using Step 2 of the SP. Step 2 updates the gains of other objects in the database similar to the Step 2 of MP in Algorithm 2. Step 3 locates the object with maximum gain. Step 4 recomputes the gain of the object using Step 2 of the SP given in Algorithm 5. If the new object has gain greater than any of the existing references, it is included in the reference set (Steps 5 and 6).

### 5.3.1 Computational Complexity

The algorithm needs access to distances between the new object and all the database objects. Step 2.c needs the query-to-object distances. With each distance computation taking \( O(t) \), the total time taken for object comparisons is \( O(t((N + 1) + |Q|)) \). Step 2 of the algorithm considers each object in the database as a candidate reference and updates its gain. This takes \( O(Nm|Q|) \) time. SP is used twice in Steps 1 and 4. This takes \( O(2fm|Q|) \). Thus the overall time taken for an insertion operation is \( O(t((N + 1) + |Q|) + Nm|Q| + 2fm|Q|) \).

### 5.4 Experiments

In perhaps the most common application of reference-based indexing, a moderately-sized database is made available via the Internet to an extended user community. The user community submits both queries and new objects to be included into the database. For example, consider the GenBank database of DNA objects. On average, this particular database processes 100 thousand object insertions each day. Given this sort of application, natural questions are:
input: Database $S$, $|S| = N$.
Reference set $V$, $|V| = m$.
Sample queries $Q$, $|Q| = q$.
New Database object $X$.
Gains from $m$ references, $G[1..|V|]$.

output: Set of references $V = \{v_1, v_2, \ldots, v_m\}$.

1 Compute $G[X]$ using the SP.
2 for each $[s_i, X], \forall s_i \in S$ do
3   Let $g_o$ be the number of queries for which $X$ is pruned using references in $V$.
4   $P[a] = 0, 1 \leq a \leq m$. /* Initialize gain for $ith$ reference $V[*]$*/
5   for each $[e, q]$ pair, $\forall e \in V$ and $\forall q \in Q$ do
6     $V' = V - \{e\} \cup \{s_i\}$.
7     if $\text{PRUNE}(V', q, X) = 1$ then
8        $P[e]++$.
9     end
10  end
11 end
12 if $\text{MAX}(P) > g_o$ then
13   $G[i] += \text{MAX}(P) - g_o$.
14 end
15 Let $w = \text{argmax}_i(G[i])$.
16 Compute $G[w]$ using the SP.
17 Let $e = \text{argmin}_i(G[V[1..m]])$.
18 if $G[e] < G[w]$ then
19   $V = V - \{V[e]\} \cup \{w\}$.
20 end

**Algorithm 6: Three Pass Algorithm.**

- Which of the proposed methods (MP, SP, or TP) is most suitable for a dynamic environment where both insertions and queries must be processed?
- Does the suitability of each method changes as the specific characteristics of the database change?

There are many factors that need to be considered when answering these questions. MP can be used in a dynamic environment by simply rebuilding the index from scratch after every new insertion (or periodically, if it is not too problematic that the index become stale). Almost certainly, this will result in a database that gives superior query-processing speed compared to SP.
or TP which use less information during index construction. However, the cost associated with frequent index rebuilds may be problematic.

SP and TP will expectedly give inferior query processing capability compared to MP, yet will be able to process insertions more easily. All of this is complicated by the fact that all of the proposed algorithms require that the distance from each new database object to each existing database object be computed as the database grows, which adds a very significant fixed cost to every new database insertion for each algorithm.

These considerations must all be taken into account when choosing which of the three methods to use. The goal of this experimental section is to carefully benchmark each of the three methods in order to be able to point out under what circumstances each method may or may not be preferred.

5.4.1 Query Performance

The goal of this particular experiment is to simulate a real-life environment such as GenBank, where the database is growing in size due to insertions, and where queries are being asked simultaneously to the database growth.

5.4.1.1 Experimental setup

One important consideration when designing this experiment is to re-create the realistic situation where the distribution of queries as well as recently inserted data is not stationary. In a realistic environment, one may expect that the distribution of the data objects that are stored in the database (as well as the queries asked) change as the application and user base evolves over time. Such a non-stationary distribution may be expected to favor the “massive-rebuild” option presented by the MP method, where the index is simply reconstructed from scratch periodically. The reason for this is that both SP and TP save computation by assuming that the index does not change too much over time. If the distribution of data and/or queries does change significantly, then this assumption may be problematic and one might expect the query performance to deteriorate. The goal is to test what the effect of such a non-stationary distribution is on the quality of the three methods.
In order to test the effect of such considerations on query performance, the following experimental setup is used. Given the DNA data set of 40,000 objects, the database objects are first ordered (as we will describe later in this section). Then, the first 4,100 objects from the ordering are considered. A random selection of 100 of these are considered as part of a sample query set and the remaining 4,000 as part of the initial set of objects in the database. An index is constructed for this initial database using MP and the sample query set. The remaining objects are processed in the DNA database by their ordering, one object at a time. The first twenty-six objects from the ordering are inserted into the database. For SP and TP, the index is updated after each insertion. For MP, the index is reconstructed from scratch after the twenty-sixth insertion. Then, the twenty-seventh object is selected as the query object for the search. For this query object, the number of object comparisons required to answer the query using each of the three indexing strategies are computed. For SP and TP, this query is then added to the query sample using the reservoir algorithm [93]. Then, the next twenty-six objects (in order) are inserted. The
next object is used as a query over all of the inserted data. This process is repeated until 1,000 queries and 26,000 insertions have been performed.

Two different orderings of the DNA data are considered. The first ordering, a Hilbert ordering, is obtained as follows. The percentage of each letter in the DNA object defines a multi-dimensional vector (four dimensional since DNA object contains four different letters). The objects are then ordered according to the Hilbert ordering of their resulting vectors. Because of the Hilbert ordering, the distribution of the data that are inserted will vary over time. The first few objects will have a high percentage of a single letter and low percentages of the rest of the letters, but that percentage will drop for objects inserted later on, as the database is processed and other letters become prominent. For example the DNA objects have \{A, C, G, T\} as alphabet set. The Hilbert ordering of this database has more than 40% A’s in the first few objects. This percentage decreases to 10% towards the end of the ordering. Because of the experimental setup,

Figure 5-2. Comparison of the proposed methods on DNA database with random-ordered data and query distributions.
this simulates the situation where queries are always more oriented towards the most-recently inserted data, which may be a realistic scenario.

For the second experiment over the DNA data set, the data is ordered randomly. Thus, there is no drift of the data and query distribution over time in this case. The same experiment is also performed over the randomly ordered images from the retinal image data set. Here the initial data set contained 1000 images and 100 sample query images. Then a random image is used as query for every 10 images inserted. This process is continued until 3500 images have been inserted into the database.

For the image data set, $\epsilon = 8\%$ of the largest distance and $k = 16$ are used. For the two orderings of DNA database, $\epsilon = 8$ and $k = 32$ are used.
5.4.1.2 Experimental results

The number of comparisons required to answer each query as a function of database size (or equivalently, time) for each of the various methods over various databases is given in Figures 5-1 to 5-4. In addition to MP, SP, and TP, we present the results obtained by running the same experiment using the Sparse method proposed by Brisaboa et.al. [21] for dynamic databases. The other methods considered experimentally in this paper such as OMNI and BINC are not able to handle dynamic updates, and so are not tested in these experiments.

The various plots depicted are fairly jagged due to the large number of queries asked and the high variance of index performance on any given query, but the results show that the incremental algorithms require may require more than three times more comparisons than MP. Not surprisingly, SP required more comparisons than TP for all three data sets, though the difference between the two methods is negligible for the randomly-ordered DNA data set. This particular data set also showed the closest performance among all three methods. This is also not
Figure 5-5. Distance computation time of DNA database.

surprising, given that the data and query distributions are stationary over time. Perhaps the most surprising finding is that for the randomly ordered image database, there is a very clear separation in performance of the various methods over time.

For the DNA databases, the Sparse method failed to generate a good set of references and it must scan almost the entire database. For the image database, Sparse requires up to three times more comparisons than MP and up to two times more comparisons than TP.

5.4.2 Distance Computations

The previous experiment tested the ability of the various methods to process queries efficiently, but in an dynamic environment insert processing speed is also important.

The first experiment regarding insertion processing speed is meant to test the magnitude of a cost that is common to all three algorithms: the cost of computing the distance of a new database object to all existing database objects. For the two different distance measured considered
Figure 5-6. Distance computation time of image database.

(ED and EMD), the time take to process the necessary distance computations associated with insertions numbered 1000 to 3000 is computed.

The timing results for the DNA database (using ED) are given in Figure 5-6. With the increase in size from 1000 to 3000 the time taken to process each additional insertion increases from 0.12 to 0.36 seconds. The time required for the image data set (using EMD) is given in Figure 5-5. With the increase in size from 1000 to 3000, the time taken increases from 2500 seconds to 7500 seconds. As one might expect, times scale linearly with database size for both distances, but there is a very significant difference in the magnitude of the cost for both methods.

5.4.3 Impact of Index Construction Time

After demonstrating the fixed costs that all three methods must pay in order to process an additional insertion, the actual index construction times of the three methods are compared.

The time taken for index construction by the two incremental methods with increase in database size for the image data set is given in Figure 5-7, using $\epsilon = 8\%$ of the largest distance
Figure 5-7. Index construction ($I_C$) times of the methods SP and TP DNA and image databases.

and $k = 32$. The time taken by TP varies from 0.06 second for 1000 objects to 0.13 second for 3000 objects. Similarly the time taken by SP varies from 0.02 second to 0.04 second. The time taken to construct the index for MP is given in Figure 5-8. It varies from 175 seconds for 1000 objects to 675 seconds for 3000 objects and increases linearly. Thus the incremental algorithms are orders of magnitude faster than MP. We obtained similar results for the methods SP, TP and MP using the DNA database.

5.4.4 Analyzing the Results

The experiments suggest that there is no easy answer to the question: Which is better, MP, SP, or TP? Depending upon the application characteristics, either MP or one of the incremental methods may be superior.

For distance metrics of intermediate computational cost such as the ED, the incremental methods such as SP and TP seem preferable. The distance computation cost of ED is significant,
but not debilitating. Specifically, it has a low cost (Figure 5-5) compared to the index construction cost of the MP (Figure 5-8) which dominates – the time required to rebuild the index using MP is around 3000 times the cost of computing all of the distances to a newly inserted database object. The gain from MP’s superior query speed (up to two times faster compared to the incremental algorithms) does not seem to justify its costs. For example, the statistics associated with the DNA objects in the GenBank [10] database show that for every new object that is inserted into the database, the database has around one new query. With a 1 : 1 query to update ratio, the update cost is just as important as query cost, making MP far less attractive.

On the other hand, for distance metrics of high computational cost such as EMD, applying MP is preferred. The index construction time for MP (Figure 5-8) is low compared to the costly EMD distance computation (Figure 5-6). Even though the incremental algorithms have an index construction cost that is negligible (Figure 5-7), their query performance is up to three times slower than MP. The difference in their query times is much greater than than the index
construction of MP (300 seconds more than MP). Hence for the image database applying MP is probably a better option.

If an incremental algorithm is selected, choosing from among SP and TP is somewhat difficult. The latter generally has superior query performance – though not over the DNA data set with a random ordering. The former has better insert processing performance. A rule of thumb might be that if inserts are more common, choose SP, if queries are more common, choose TP.
CHAPTER 6
GENERALIZED NEAREST NEIGHBORS FOR SIMILARITY JOINS

This chapter presents a generalized framework for Nearest Neighbor queries called Generalized Nearest Neighbors (GNN) to answer the similarity join queries.

Finding the broadness of data is needed in many applications such as life sciences (e.g., detecting repeat regions in biological sequences [46] or protein classification [24]), distributed systems (e.g., resource allocation), spatial databases (e.g., decision support system or continuous referral system [55]), profile-based marketing, etc.

In this dissertation, a new database primitive, called the Generalized Nearest Neighbor (GNN) which naturally detects data broadness is defined. Given two databases $R$ and $S$, the GNN query finds all the objects in $S^t \subseteq S$ that appear in the $k$-NN set of at least $t$ objects of $R$, where $t$ is a cutoff threshold. The objects in the result set of a GNN query are broad. Here, $S^t$ is the set of objects that the user focuses on for broadness property. If $R = S$, then it is called mono-chromatic query. Otherwise, it is called bi-chromatic query.

The trivial solution to a GNN query is to run a $k$NN query for each object in $R$ one by one, and accumulate the results for each object in $S$. However, this approach suffers from both excessive amount of disk I/Os and CPU computations. When the databases do not fit into the available buffer, a page that will be needed again might be removed from buffer while processing a single $k$NN query. CPU cost also accounts for a significant portion of the total cost since $k$NN is determined for each object in $R$ from scratch.

Lets assume that each of the databases is larger than the available buffer. Three solutions are proposed that arrange the data objects into pages. Each page represents a set of objects represented by their minimum bounding rectangle (MBR). Two R*-trees [9] are constructed, one built on $R$ and other on $S$. They predict a set of candidate pages from $S$ that may contain $k$NNs for each MBR of $R$ with the help of R*-trees. Each candidate page is assigned a priority based on its proximity to that MBR of $R$ and is stored in a Priority Table (PT). The first algorithm, pessimistic approach, fetches as many candidate pages as possible from $S$ for each page of $R$. The second algorithm, optimistic approach, fetches one candidate page at a time from $S$ for each page of $R$. The third algorithm, hybrid approach, combines the advantages of the first two algorithms.
page of $R$. The third algorithm dynamically decides the number of pages that needs to be fetched for each page of $R$ by analyzing query history. It reduces the CPU and I/O cost significantly through three optimizations by dynamically pruning 1) pages of $S$ that are not in the $k$-NN set of sufficiently many objects in $R$, and 2) pages of $R$ whose nearest neighbors do not contribute to the result 3) objects in candidate MBRs of $S$ that are too far from the MBRs of $R$. The method further reduces these costs by pre-processing the input databases using a packing technique called Sort-Tile-Recursive (STR) [76].

### 6.1 Problem Definition

Let $R$ and $S$ be two databases. The GNN query is defined by a 5-tuple $\text{GNN}(R, S, St, k, t)$, where $St \subseteq S$, and $k$ and $t$ are positive integers. This query returns the set of tuples $(s, R_s)$, where $s \in St$, $R_s \subseteq R$ is the set of objects that have $s$ as one of their $k$-NN, and $|R_s| \geq t$. The Euclidean distance is used as the distance measure in this paper unless otherwise stated.

Assume that the white and black points in Figure 6-1 show the layout of 2-D databases $R = \{r_1, \cdots, r_5\}$ and $S = \{s_1, \cdots, s_5\}$ respectively. Consider the following query:

$$\text{GNN}(R, S, St = \{s_1, s_2, s_5\}, 2, 3).$$
This translates as: “Find the objects in $S_I$ that are in the 2-NN set of at least three objects in $R$”. In Figure 6-1, the circles centered at each $r_i \in R$ covers the 2-NN of $r_i$, $\forall i$. Only $s_2$ and $s_4$ are covered by at least three circles. $s_4$ can be ignored since $s_4 \notin S_I$. The set of nodes that have $s_2$ in the 2-NN is $\{r_1, r_2, r_3\}$. Therefore, the output of this query is $\{(s_2, \{r_1, r_2, r_3\})\}$. Note that the data points in $S - S_I$ cannot be ignored prior to GNN query. In other words $\text{GNN}(R, S, S_I, k, t) \neq \text{GNN}(R, S_I, S_I, k, t)$. For example, in Figure 6-1, removal of $s_3$ and $s_4$ prior to GNN query changes the 2-NNs of $r_2$, $r_3$ and $r_4$. As a result $s_1$ becomes one of the 2NNs of $r_2$ and $r_3$. Hence $s_1$ is incorrectly classified as broad.

A nice property of the GNN query is that both mono-chromatic and bi-chromatic versions of the standard $k$-NN, ANN and RNN queries are its special cases. Following observations state these cases. One can prove these from the definition of the GNN query. Note that the goal of this paper is not to find different solutions to each of these special cases. The goal is to solve a broader problem which can not be solved trivially using these special cases.

**Observation 1.** $\text{GNN}(\{r\}, S, S, k, 1)$ returns the $k$-NN of the object $r$ in $S$. If $r \in S$, then it corresponds to the mono-chromatic $k$-NN query. Otherwise, it corresponds to the bi-chromatic $k$-NN query.

**Observation 2.** $\text{GNN}(R, S, S, 1, 1)$ returns the ANN of $R$ in $S$. If $R = S$, then it is the mono-chromatic case, otherwise bi-chromatic case.

**Observation 3.** $\text{GNN}(R, S, \{s\}, 1, 1)$, where $s \in S$, returns the RNN of the object $s$ in $R$. If $R = S$, then it is the mono-chromatic case, otherwise bi-chromatic case.

### 6.2 Overview of the Algorithm

Let’s assume that each of the databases is larger than the available buffer. The three proposed solutions arrange the data objects into pages. Each page represents a set of objects represented by their minimum bounding rectangle (MBR). Two R*-trees [9] are constructed, one built on $R$ and other on $S$. They predict a set of candidate pages from $S$ that may contain $k$-NNs for each MBR of $R$ with the help of R*-trees. Each candidate page is assigned a priority based on its proximity to that MBR of $R$ and is stored in a Priority Table (PT). The first algorithm,
pessimistic approach, fetches as many candidate pages as possible from $S$ for each page of $R$.

The second algorithm, optimistic approach, fetches one candidate page at a time from $S$ for each page of $R$. The third algorithm dynamically decides the number of pages that needs to be fetched for each page of $R$ by analyzing query history. It reduces the CPU and I/O cost significantly through three optimizations by dynamically pruning 1) pages of $S$ that are not in the $k$-NN set of sufficiently many objects in $R$, and 2) pages of $R$ whose nearest neighbors do not contribute to the result 3) objects in candidate MBRs of $S$ that are too far from the MBRs of $R$. The method further reduces these costs by pre-processing the input databases using a packing technique called Sort-Tile-Recursive (STR) [76].

6.3 Predicting the Solution: Priority Table Construction

Let $R$ and $S$ be two given databases, and $A$ and $B$ be the sets of MBRs that contain the objects in these databases. The computation of the candidate set of MBR pairs one from $A$ and the other from $B$ that needs to be inspected to calculate a given GNN query is discussed. Let’s assume that the databases are packed and indexed prior to the GNN query. The packing of the database is discussed in more detail in Section 6.6. This is a one time cost per database; the same index will be used for all the queries. STR [76] ordering is used for a total ordering of the data. Throughout this paper R*-Tree is used to index the databases. Other index structures can be used to replace the R*-tree. For simplicity, the capacity of each MBR of the R-tree is chosen as one disk page and use leaf level MBRs to prune the solution space.

Given two MBRs $B_1$ and $B_2$, $\text{MAXDIST}(B_1, B_2)$ and $\text{MINDIST}(B_1, B_2)$ are defined as the maximum and minimum distance between $B_1$ and $B_2$. The following lemma establishes an upper bound to the $k$-NN distance to the objects in a set of MBRs.

**Lemma 1.** Let $A$ be the MBR of a set of objects and $a \in A$ be an object. Let $B = \{B_1, \cdots, B_{|B|}\}$ be the set of leaf level MBRs of an index structure built on a database. Assume that the MBRs in $B'$, where $B' \subseteq B$, contain at least $K$ objects. Let $\varepsilon$ denote the distance of object $a$ to its $k$th NN in $B$, then

$$\varepsilon \leq \max_{B \in B'} \{\text{MAXDIST}(A, B)\}, \forall k, 1 \leq k \leq K.$$
Proof follows from the observation that all objects in $B'$ appear in $B$ too. For a given positive integer $k$, let $m$ be the integer, $1 \leq m < |B|$, for which
\[
\sum_{i=1}^{m-1} |B_i| < k \leq \sum_{i=1}^{m} |B_i|,
\]
where $|B_i|$ is the number of objects in $B_i$. Let $\text{MAXDIST}(A, B_i) \leq \text{MAXDIST}(A, B_{i+1}), \forall i$ $1 \leq i < |B|$, where $|B|$ is the cardinality of $B$.

From Lemma 1, the $k$-NN distance of the objects in $A$ to the objects in $B$ is at most $\text{MAXDIST}(A, B_m)$. Hence, if $\text{MAXDIST}(A, B_m) < \text{MINDIST}(A, B)$, $B \in B$, then $B$ does not contain any object from the $k$-NN set of any object in $A$. Therefore, $B$ can be pruned away from $B$ without any false dismissals during the computation of the $k$-NNs of the objects in $A$. From these observations, given a GNN query, $\text{GNN}(R, S, S' \subseteq S, k, t)$, for each $A \in A$, a priority list of the candidate boxes in $B$ is computed as follows:

- For each $A \in A$,
  
  **Step 1:** Compute $\text{MAXDIST}(A, B_m)$ for the given value of $k$ as discussed above.
  
  **Step 2:** Find the MBRs, $B \in B$ for which $\text{MINDIST}(A, B) \leq \text{MAXDIST}(A, B_m)$.
  
  **Step 3:** Assign priorities to these MBRs in increasing $\text{MINDIST}(A, B)$ order.

The algorithm for Step 1 takes a MBR $A$, the root node of an R*-tree, and an integer $k$ as input. The root node is stored using a min-heap. The node with the smallest MAXDIST to $A$ is extracted from this heap. If the MINDIST of this node to $A$ is more than the threshold, then it is omitted. Otherwise it is inspected. If it is an internal node, then its children are inserted into the min-heap. Otherwise, it is inserted into the candidate set, which is maintained using a max-heap. If the candidate set contains more objects than necessary, then the MBR with the largest MINDIST value is removed from the candidate set. Although the worst case time complexity of this step is $O(|B|)$ (i.e., the entire index is traversed), the amortized complexity is only $O(\log(|B|))$. Step 2 is computed using the classic range search algorithm on R-trees. Therefore, the amortized time complexity of this step is also $O(\log(|B|))$. This step eliminates all the leaf level MBRs that only contain irrelevant points. Naturally, if an MBR contains at least one relevant point, it is detected in Step 1 and that MBR will be processed by the strategies proposed.
Figure 6-2. A sample Priority Table for two databases \( R \) and \( S \).

in Section 6.4. Step 3 takes \( O(C \log C) \) time where \( C \) is number of candidate MBRs found at Step 2.

The candidate MBRs for all the MBRs in \( A \) are stored in a **Priority Table (PT)**. Figure 6-2 depicts the PT constructed for the GNN(\( R, S, St, k, t \)) query. Here, \( r_i \) and \( s_i \) correspond to MBRs for \( R \) and \( S \). Each row/column corresponds to a page or \( R/S \) on disk. The numbers at each cell show the priority of that column for that row. Lets assume that \( St = \{s_1, s_3, s_4, s_5, s_7\} \) in this example. In this figure, each row and column corresponds to an MBR for \( R \) and \( S \) respectively. For simplicity, two assumptions are made without affecting the generality: 1) The objects in \( R \) and \( S \) are located sequentially on disk. 2) Each row and column of the PT (i.e., each MBR) corresponds to one disk page. The numbers at each row show the priority of the candidate MBRs in \( S \) for the corresponding MBR in \( R \). For example in row 1, the MBRs \( s_1, s_3 \) and \( s_7 \) are in the candidate set of \( r_1 \), such that \( s_3 \) has the highest priority and \( s_1 \) has the lowest priority. This is depicted in Figure 6-3. Here \( r_1 \in R \) and \( S = \{s_1, \cdots, s_7\} \). Objects in \( m_1 \) are within MAXDIST distance from \( r_1 \). If an MBR of \( S \) is not in the candidate set of an MBR in \( R \), then the corresponding cell is unnumbered.
Given a query, GNN\((R, S, S' \subseteq S, k, t)\), our search methods reduce the solution space by pruning the PT. Following two optimizations can be made to reduce search space by inspecting the PT:

**Optimization 1.** (Column Filter) Let \(s_i\) correspond to an MBR in \(S'\). If the total number of objects in the MBRs in \(R\) which have \(s_i\) in their candidate set is less than \(t\), then that column can be removed from \(S'\).

For example, in Figure 6-2, \(s_5\) is in the candidate set of only \(r_4\). If the total number of objects in \(r_4\) is less than \(t\), then \(s_5\) can be removed from \(S'\) safely. The correctness of Column Filter can be proven from the fact that an object in a column, \(s_i\), can be in the \(k\)-NN set of the objects in the rows only that have \(s_i\) in the candidate set.

**Optimization 2.** (Row Filter) If a row does not contain any candidate MBRs in \(S'\), then it can be removed from PT.

For example, in Figure 6-2, rows \(r_3\) and \(r_8\) do not have any candidates in \(S'\). Therefore, these rows can be omitted safely. If \(s_5\) is pruned from \(S'\) due to Column Filter, then the row, \(r_4\), can also be ignored.
6.4 Static Search Strategies

PT defines the MBR pairs that (potentially) need to be compared to answer a given GNN query. This section develops two methods to compute a GNN query, GNN($R$, $S$, $S_t \subseteq S$, $k$, $t$), given the PT of the databases $R$ and $S$. These methods are referred as Fetch All (FA) and Fetch One (FO). Lets assume a limited buffer space $B$ throughout this section. That is, the sizes of both $R$ and $S$ are larger than $B$.

6.4.1 Fetch All

The first method uses a pessimistic strategy: to process each page (i.e., MBR) of $R$, (i.e., one row in PT), it reads as many candidate pages from $S$ as possible into buffer at once starting from the one with the highest priority. For example, for $r_1$, FA reads $s_1$, $s_3$, and $s_7$. FA runs in 3 phases: (1) find maximal clusters that fit into buffer, (2) reorder the clusters to maximize the overlap and (3) read the pages for each cluster and process the contents. Next, each phase is elaborated.

6.4.1.1 Creating clusters

The clusters are created by iteratively adding rows into the current cluster, starting from the first row, until its size becomes $B$. When the cluster becomes large enough, a new cluster begun. For example, if $B = 6$ pages, then the clusters for the PT in Figure 6-2 are $C_1 = \{r_1, r_2, s_1, s_3, s_4, s_7\}$, $C_2 = \{r_3, r_4, s_2, s_5, s_6, s_8\}$, $C_3 = \{r_5, s_1, s_2, s_4, s_8\}$, and $C_4 = \{r_6, r_7, r_8, s_1, s_2, s_6\}$. The total cost of this step is linear in the number of candidate pages since each candidate page is visited only once.

6.4.1.2 Ordering clusters

The order for reading the clusters affects the total amount of disk I/O. This is because, if consecutive clusters have common pages, these pages will be reused and they do not need to be read again. For example, if $C_3$ is read after $C_1$, then $s_1$ and $s_4$ will be reused, saving two disk reads. Given a read schedule of clusters, the total amount of disk reads saved by reusing buffer is equal to the sum of the common pages between consecutive clusters. For example, if
the clusters are read in the order of $C_1$, $C_3$, $C_2$, $C_4$, then total savings adds up to 6 pages (i.e., $|C_1 \cap C_3| + |C_3 \cap C_2| + |C_2 \cap C_4| = 6$).

One can show that the Traveling Salesman Problem (TSP) can be reduced to the problem of finding the best schedule for reading clusters. Intuitively, the proof is as follows. Each vertex of TSP maps to a cluster. Each edge weight $w_{i,j}$ between clusters $C_i$ and $C_j$ is computed as the number of overlapping pages between $C_i$ and $C_j$. The best schedule on this graph is the Hamiltonian Path that maximizes the sum of edge weights. Since TSP minimizes the sum of edge weights, the weight of each edge $w_{i,j}$ is updated as $w'_{i,j} = w_{max} - w_{i,j}$, where $w_{max}$ is the largest edge weight. This guarantees that the new edge weights are non-negative. Then a new node $v$ is created and is connected to all nodes by zero-weight edges. The optimal schedule is the path with the smallest sum of edge weights which begins at vertex $v$ and visits all nodes once.

A greedy heuristic is used to find a good schedule as follows: We start with an empty path. While there are unvisited vertices, we insert the next edge with the smallest weight into the path if it does not destroy the path. Finally, the disconnected paths are attached randomly if there are any.

6.4.1.3 Processing clusters

Once the cluster schedule is determined, the contents of each cluster are iteratively read into buffer using optimal disk scheduling [77]. The procedure used to process each cluster after it is fetched into buffer is given in Algorithm 7. For each row in the cluster, the algorithm searches the $k$-NN of each object starting from the box with the highest priority (Steps 1 and 2). The results obtained at this step are used to prune the candidate set (Step 3). After the candidate set is pruned, Optimizations 1 and 2 are applied to PT in order to further reduce the solution space.

6.4.2 Fetch One

FA reads many redundant pages if only a small percentage of the candidate pages contain actual $k$-NNs. FO uses optimistic approach to avoid this problem. FO iteratively reads one page per row as long as there are more candidates.
**Algorithm 7**: Process Buffer Algorithm.

The pseudocode for the FO algorithm is given in Algorithm 8. The algorithm splits buffer equally for each of the databases. This is because, one candidate page is read per row starting from the highest priority (Step 1). Therefore, the number of pages from each database in the buffer will be equal at all times if all the candidate pages are distinct. After searching each candidate page (Step 2), PT is further pruned by eliminating the pages that are farther than the \( k \)-th NN found so far (Step 3), and using Optimizations 1 and 2 (Step 4).

For example, for the PT in Figure 6-2, let buffer size be 6 pages, then FO reads \( \{r_1, r_2, r_3\} \) and \( \{s_3, s_4, s_6\} \) into buffer. Assume that the third candidate of \( r_1 \) is pruned at the end of this step. Next, \( \{s_7, s_8\} \) are read to replace \( \{s_4, s_6\} \). Although it is the second candidate of \( r_2 \), \( s_3 \) is not read at this step since it is already in buffer. Assume that the third candidate of \( r_2 \) is pruned at the end of this step. Since none of the rows \( \{r_1, r_2, r_3\} \) have any remaining candidates FO does not need to read any more pages for these rows. Therefore, \( \{r_1, r_2, r_3\} \) is replaced with \( \{r_4, r_5, r_6\} \), and the search continues recursively.

**Algorithm 8**: Fetch One Algorithm.
**Algorithm 9**: Fetch Dynamic Algorithm.

### 6.5 Dynamic Strategy

FO reads only the necessary pages (i.e., MBRs) to compute a given GNN($R$, $S$, $St$, $k$, $t$) query since it reads one page at a time starting from the highest priority and stops when the distance to the next MBR is more than the distance to the $k$th NN found so far. However, this does not guarantee that the total I/O cost is minimized. This is because FO incurs a random seek cost every time a new page is fetched from disk. Since a random seek is significantly more costly than a page transfer, reading a few redundant pages sequentially at once may be faster than FO. Thus, neither FO nor FA ensures the optimal I/O cost. The number of pages read at each iteration, $f$, that minimizes the I/O cost depends on the query parameters and the distribution of the database. A good approximation to this number can be obtained by sampling the MBRs of $R$.

The third method, *Fetch Dynamic (FD)* adaptively determines the value of $f$ as follows. It starts by guessing the value of $f$. It then reads the first cluster using this value. As it finds the $k$-NNs of all the objects in the first cluster, it computes the optimal value of $f$ for that cluster. It then uses this value of $f$ to choose the next cluster. After processing each cluster, it iteratively updates $f$ as the median of the number of pages needed for all of the rows processed so far. Note that, the choice of the initial value of $f$ has no impact in the performance after the first step, since $f$ is updated immediately after every iteration. As more rows are processed in each iteration, $f$ adapts to the query parameters and data distribution.
The pseudocode of the FD algorithm is given in Algorithm 9. The algorithm first assigns an initial value for \( f \) \((1 \leq f \leq \text{candidate size})\). 20\% of the average number of candidates of \( R \) is used as the initial guess. Let \( B \) denote the buffer size. While there are unprocessed rows, FD reads \( \left\lfloor \frac{B}{f+1} \right\rfloor \) pages \((r_i)\) from \( R \) and \( f \) pages \((s_{r_i})\) from \( S \) with the highest priority for each \( R \) page in buffer (Step 2.a). Thus, if all the candidates are distinct, buffer is filled with pages from \( R \) and \( S \). Steps 2.b processes each candidate page \( s_{r_i} \). The processed pages \((r_i s)\) are removed from buffer at Step 2.c. The algorithm continues with Steps 2.a to 2.c until all the rows in buffer are exhausted. Then Optimizations 1 and 2 are applied (Step 2.d). The value of \( f \) is updated at Step 2.e as the median of the number of candidates of the processed pages in \( R \).

### 6.6 Further Improvements for GNN Queries

So far the two optimizations, row filter and column filter to trim both I/O and CPU costs are discussed. This section discusses further optimizations to cut down both CPU and I/O costs of FA, FO, and FD.

#### 6.6.1 Adaptive Filter

The third optimization follows from the following observation. A given MBR \( r \) can be expanded by \( d_{\text{max}} \) in all dimensions. If a candidate MBR \( s \) overlaps with this expanded MBR, the distances between all pairs of points from \( r \) and \( s \) are computed. (Steps 2 and 3 of Algorithm 7) This incurs \( O(t^2) \) comparisons if each MBR contain \( O(t) \) points. This cost is reduced in two ways. First, instead of expanding by \( d_{\text{max}} \), different dimensions can be adaptively expand by different amounts. Second, the \( t^2 \) comparisons are avoided by pruning unpromising points from \( S \) in a single pass. More formally, first all points in a candidate MBR \( s \) that are contained in the expanded MBR of \( r \) are found. Next, the distances between all those points and all points of \( r \) are computed. Let \( t_l, t_r \leq t \), be the number of points in \( s \) that are contained in the expanded MBR of \( r \). The CPU cost for the comparison of MBR pairs drops from \( O(t^2) \) to \( O(t + t \cdot t_l) \). This is summarized as the third optimization, Adaptive Filter.

**Optimization 3.** (Adaptive Filter) Let \( p \) be a point in MBR \( r \). Let \( d \) be the \( k \text{NN} \) distance of \( p \) to the points in MBR \( s \). Let \( k \text{NN-sphere of } p \) denote the sphere with radius \( d \), centered at \( p \). Let \( M \)
denote the MBR that tightly covers the kNN-spheres of all the points in r. A point can not be a kNN of a point in r if it is not contained in M.

In Figure 6-4, \( m_1, m_2 \) are the expanded MBRs for the r without using and using the Adaptive Filter respectively. \( s_1, s_2 \subseteq S, s_1 \) is the MBR of the points \{q_1, q_2, q_3, q_4\}. The expanded MBR of r in the worst case is given by \( m_1 \). When adaptive bounds are used, the expanded MBR \( m_2 \) is obtained. In the former case, two MBRs \( s_1 \) and \( s_2 \) intersect with \( m_1 \). Thus, 3 disk I/Os (r, \( s_1 \), \( s_2 \)) and 32 comparisons are made. However, only \( s_1 \) intersects with \( m_2 \) in the latter case. Hence MBR \( s_2 \), which do not have any point inside \( m_2 \), can be pruned. This reduces the I/O cost to 2 page reads (r, \( s_1 \)) and the CPU cost to 16 comparisons. However, Optimization 3 states that a point in S is considered only if it is inside \( m_2 \). Therefore, each point in \( s_1 \) can be scanned once to find such points. These points are then compared to the points in r to update k-NNs. Thus the CPU cost reduces to 12 comparisons (4 for scanning \( s \) and 8 for comparison of the points in r with \( q_1 \) and \( q_2 \)).

6.6.2 Partitioning

Optimization 3 is improved further by partitioning the MBR r along selected dimensions. Dimensions with high variances are selected for the partitioning. It starts from the dimension with the highest variance. The MBR is split along this dimension into two MBRs, such that each
Partitioning improves the performance in two ways. First, since each of the partitions is smaller than the original MBR, the pruning distance \(d_{\text{max}}\) along each dimension is reduced. This reduces the I/O cost. Second, without partitioning, an object in MBR \(s\) is compared to all the objects in \(r\) if the extended MBR of \(r\) contains it. However, with partitioning, an object in \(s\) is not compared to the objects in partitions whose extended MBR does not contain it. Thus, CPU cost is reduced by avoiding unnecessary comparisons. Note that as the number of partitions increases, the number of point-MBR comparisons increases. When the number of partitions becomes \(O(t)\) (i.e., the number of objects per MBR), the number of such comparisons becomes \(O(t^2)\). Thus, partitioning becomes useless. In the experiments, they are partitioned along at most eight dimensions for the best performance.

In Figure 6-5, \(v_1\) and \(v_2\) are two partitions of MBR \(r\) and \(m_3\) and \(m_4\) are their extended MBRs. \(s_1 \subseteq S\) is the MBR of the points \(\{q_1, q_2, q_3, q_4\}\). Horizontal dimension is used to partition the MBR \(r\) into two partitions. \(v_1\) and \(v_2\) are the MBRs of these partitions having \(m_3\) and \(m_4\) as extended MBRs. MBR \(s_2\), which do not have any points inside \(m_2\), can be pruned. Each point in \(s_1\) is scanned once to find the candidate points for the partitions \(v_1\) and \(v_2\). Only \(q_1\) is present in
the extended MBR of $v_2$, reducing the CPU cost to 10 (8 for comparing points in $s_1$ with $v_1$ and $v_2$ and 2 for comparing the points in $v_2$ with $q_1$).

### 6.6.3 Packing

The performance of R-Tree based methods can be improved by using packing algorithms which group similar objects (objects within a close neighborhood) together. The Sort-Tile-Recursive (STR) method [76] is employed for packing the R-Tree, built on the databases. Let $N$ be the number of $d$-dimensional objects in a database, $B$ be the capacity of a node in R-Tree and let $P = \lceil \frac{N}{B} \rceil$. STR sorts objects according to the first dimension. Then the data is divided into $S = \lceil P^{\frac{1}{d}} \rceil$ slabs, where a slab consists of a run of $n.\lceil P^{\frac{d-1}{d}} \rceil$ consecutive objects from the sorted list. Now each slab is processed recursively using the remaining $d - 1$ coordinates. It has been shown in [76] that for most types of data distributions STR-Ordering performs better than space-filling-curve based Hilbert-Ordering [51].

### 6.7 Experiments

Two classes of databases are used in the experiments.

**Image databases:** Each of Image1 and Image2 contains 60-dimensional feature vectors of 34,433 satellite images. Two databases are created from from Image1 and Image2 by splitting each 60-dimensional vector into 30 two-dimensional vectors. Each of the resulting databases contains 1,032,990 data points.

**Protein structure databases:** Each of Protein1 and Protein2 contains 288,156 three-dimensional feature vectors for secondary structures of proteins from Protein Data Bank (ftp://ftp.rcsb.org/pub/pdb) as discussed in [24].

In addition to FA, FO, and FD, three of the existing methods are implemented: sequential search (SS) and the R-tree-based NN method of Roussopoulos et. al., (RT) [70] and Mux-Join [19]. To implement the buffer restrictions into RT, half of the available buffer is used for $R$ and the other half for $S$. In order to adapt these methods to GNN, GNN($R$, $S$, $St$, $k$, $t$), a $k$-NN search is performed for each object in $R$. SS is included in the experiments, as it is better than many complicated NN methods for a broad set of data distributions [15]. The source codes of
GORDER [97] and RkNN [84] are obtained from their authors. However, at its current state, it is impossible to restrict memory usage of GORDER to a desired amount. Therefore, GORDER is used in only one of the experiments where it is possible.

In all the experiments, \( S' = S \) is used unless otherwise stated. 4 kB page size is used in all the experiments. The experiments ran on an Intel Pentium 4 processor with 2.8 GHz clock speed.

### 6.7.1 Evaluation of Optimizations

This section inspects the performance gain due to Optimizations 1, 2 and 3 and the improvements in Section 6.6. The GNN query is performed by varying the size of \( S' \) from 0.5 % to 8 % of \( S \), by selecting pages of \( S \) randomly. In this experiment, FD used \( k = 10, t = 3,000 \), and buffer size = 10 % of the database size. The queries are run on the two dimensional image database.

The CPU and I/O time of FD with four different settings of Optimizations 1, 2 and 2 (obtained by turning these optimizations on and off) on two-dimensional image databases for different sizes of \( S' \) are given in Figure 6-6. According to the results, the main performance gain is obtained from Optimizations 2 and 3, yet there is a slight performance gain from
Table 6-1. Comparison with an Optimal solution.

<table>
<thead>
<tr>
<th>Buffer Size (%)</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>11245</td>
<td>10980</td>
<td>10244</td>
<td>10252</td>
</tr>
<tr>
<td>FO</td>
<td>14601</td>
<td>13425</td>
<td>12710</td>
<td>12393</td>
</tr>
<tr>
<td>FD</td>
<td>16706</td>
<td>13825</td>
<td>11702</td>
<td>10789</td>
</tr>
<tr>
<td>FA</td>
<td>155727</td>
<td>73238</td>
<td>23885</td>
<td>10328</td>
</tr>
</tbody>
</table>

Optimization 1. The reason that the Optimization 1 has a smaller impact can be explained as follows. \( t \) is only 0.3% of the total number of objects in \( R \). Thus, Optimization 1 can eliminate a page of \( S \) only if it is in the candidate set of less than 3 pages of \( R \). The impact of Optimization 1 is larger when the ratio of the average number of candidate pages to \( t \) is lower. This happens when \( t \) is large or \( k \) is small. Optimization 2 has a high impact when \( St \) is smaller. This is because fewer rows in PT have candidates in \( S \) for small \( St \). Another way to obtain high filtering rate from Optimization 2 is to reduce the average number of candidate pages per row by choosing a small value for \( k \). Optimization 3 effectively reduces the CPU and I/O cost for different sizes of \( St \). We can also see that for higher percentages of \( S \), the impact of this optimization remains constant and is independent of the size of \( S \). This can be understood from the fact that for a fixed
value of $k$ and at higher percentages of $S$, every MBR $r \in R$ has same number of candidate MBRs from $S$. This results in constant reduction in CPU and I/O costs.

The performance gains on top of Optimizations 1, 2, and 3 (Unpartitioned algorithm) by partitioning the MBRs and by using the STR-method based packing algorithm are compared in Figure 6-7. Here, CPU and I/O time of FD with three different settings Unpartitioned, Partitioned, and Packed (along with partition) on two-dimensional image databases for different sizes of $S_t$. Here, packing is applied along with partitioning. Partitioning reduces the I/O cost up to factor of 3 and CPU costs by orders of magnitude. The tighter bounds of the extended MBRs of the partitions resulted in a reduction of the pruning distance. This explains the I/O and CPU performance gains from the partitioned algorithm. Packing utilizes the distribution of data and groups similar objects in MBRs that have common parent and a better organization of the R*-Tree index structure. This results in a lower value for the parameter $f$ in FD and hence has better performance gains. Packing reduces the I/O cost up to 10 times and CPU cost by orders of magnitude faster than an unpartitioned algorithm. It outperforms partitioned algorithm by up to a...
factor of 2 and 6 in I/O and CPU costs respectively. From here on all optimizations are used in all of the proposed methods.

Scheduling the pages is known as paging problem [63]. Chan [25] proposed heuristic based $O((R_p + S_p)^2)$ algorithms ($R_p$ and $S_p$ are the number of pages in the two databases) for Index-based Joins. For large databases, however, these heuristics are not efficient. An online scheduling algorithm can be evaluated using competitive analysis [1]. In competitive analysis, an online algorithm is compared with an optimal off-line algorithm which knows all the candidate pages in advance. An algorithm is $c$-Competitive if for all sequences of page requests, $C_A \leq c \bar{C} + b$, where $C_A$ is the cost of the given algorithm, $\bar{C}$ is the cost of the off-line algorithm, $b$ is a constant, and $c$ is the competitive ratio.

The performance of online methods are compared with its off-line version, named Oracle. For each MBR $r \in R$, Oracle provides the set of MBRs from S such that every MBR in this set contains at least one $k$-NN of at least one object in $r$. Then the number of I/Os of Oracle is optimized using the heuristic discussed in Section 6.4. The lower bound is computed as the
optimal number of I/Os as the total number of pages in R and S. The purpose of this experiment is to observe how the I/O cost of the online methods compare to that of an off-line method and the minimum possible I/O cost. Table 6-1 compares the performance of Oracle with the proposed methods.

Since each database has 5064 pages, the lower bound is computed as the number of disk I/Os as 10128 (5064+5064). The competitive ratio of FA is smallest for large buffer sizes (1.008 for 40% buffer) and for FO it is smallest for small buffer sizes (1.3 for 5% buffer). FD has a smaller competitive ratio (1.5 for 5% buffer to 1.05 for 40% buffer). It can be concluded that our methods perform very close to the off-line method.

Table 6-2. Comparison with GORDER.

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>1000</th>
<th>500</th>
<th>200</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Size (%)</td>
<td>175</td>
<td>108</td>
<td>88</td>
<td>85</td>
</tr>
<tr>
<td>Time (seconds)</td>
<td>305</td>
<td>535</td>
<td>1519</td>
<td>4259</td>
</tr>
</tbody>
</table>

Table 6-3. Comparison with RkNn.

<table>
<thead>
<tr>
<th>$k$</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>RkNN</td>
<td>2620</td>
<td>11750</td>
<td>84145</td>
<td>175495</td>
</tr>
<tr>
<td>FD</td>
<td>101</td>
<td>101.76</td>
<td>101.3</td>
<td>101.83</td>
</tr>
</tbody>
</table>

6.7.2 Comparison of Proposed Methods

This section compares FA, FO, and FD to each other for different parameter settings.

6.7.2.1 Evaluation of buffer size

Here, the performance of FA, FO, and FD are compared when the buffer size varies from 5 to 40% of the total size of $R$ and $S$. We use two-dimensional image database with $k = 10$ and $t = 500$.

The I/O time and the running time of our methods are given in Figure 6-8. For lower buffer sizes, FA retrieves all the candidate MBRs for every row and hence I/O cost takes up most of the total time. We can observe this from the performance of FA at buffer size 5% and is dominated by the I/O cost. As buffer size increases, the cost of all three strategies drop since more pages can
be kept in buffer at a time. For small buffer sizes FO has the lowest cost since it does not load unnecessary candidates. As buffer size increases, FA has the lowest cost since it keeps almost entire $S$ in buffer. However, in all these experiments, the cost of FD is either the lowest or very close to the lower of FA and FO. This means that FD can adapt to the available buffer size.

### 6.7.2.2 Evaluation of the number of NN

The next experiment compares the performance of FA, FO, and FD for different values of $k$. A 10% buffer size is used and $t = 500$ for the two-dimensional image database.

The I/O and the running times are given in Figure 6-9. The costs of all these methods increase as $k$ increases. For different values of $k$ FO has the lowest cost and FA has the highest cost, due to the small buffer size (10%). Even when it does not have the lowest cost, FD is very close to FO. This means that FD can adapt to the parameter $k$.

### 6.7.3 Comparison to Existing Methods

This section compares FD to five existing methods SS, RT, Mux-Index, RkNN, and GORDER for different parameter settings. Two-dimensional image and protein databases are
used in the experiments. The experimental results comparing FD with well known methods for special cases are presented. Memory Usage and Running times (seconds) of GORDER on image database with varying grid sizes. FD runs in only 11.03 seconds for the same database using 20% buffer.

6.7.3.1 Evaluation of buffer size

In this experiment set, the values of $k$ and $t$ are fixed, and vary the buffer size. The two-dimensional image database is used and $k = 10$. The running times of GORDER with different amounts of memory usage and that of FD with 1.6 MB memory are computed. We measured the actual memory usage of the methods using the top command of Linux. Although the buffer size is set (an input parameter to GORDER) to 20% of the total database size, we observed that GORDER uses significant amount of memory (up to 175% of the database size) for additional book keeping. In order to reduce the actual memory usage GORDER is run with grid numbers 1000, 500, 200, and 100. However the actual memory usage of GORDER is always much larger than 20% of the total database size (i.e., 8 MB). For different memory settings, the running time of GORDER varied from 300 to 4000 seconds while for the same query, FD running times
varied from 10 to 13 seconds (see Table 2. According to these experiments, FD runs an order of magnitude faster than GORDER even when it uses much smaller buffer. It is impossible to reduce the actual memory usage of GORDER to 20% at its current implementation. Therefore, in order to be fair, it is not included in the remaining experiments.

The I/O and the running times of SS, RT, Mux-index [19] and FD for different buffer sizes on two-dimensional image and protein databases are given in Figures 6-10 and 6-11. $k = 10$ and $t = 100$ are used. FD is the fastest of the three methods in all settings. It can be seen that for small buffer sizes RT is dominated by I/O cost. As buffer size increases, CPU cost of RT dominates. Sequential scan is dominated by the CPU cost in all the experiments. The I/O cost of FD is a fraction of that of RT. FD also reduces CPU cost aggressively through Optimizations 1 to 3 and partitioning. In all the experiments, the total time of FD is less than the I/O time of RT or SS alone. Mux-Index is dominated by I/O costs in all experiments. This is because for each block in $R$ it fills the buffer with blocks from $S$. Because of the nature of GNN queries, one needs to load pages multiple times while working with limited amount of memory, independent of the method.
used, naive (sequential scan) or more sophisticated (RT and Mux-Index). FD performs only the necessary leaf comparisons and uses the near optimal buffering schedule, thus reduces both the CPU and I/O cost effectively.

6.7.3.2 Evaluation of the number NN

Here, the performance of FD, SS, Mux, RkNN and RT are compared for different values of $k$. 10% buffer size is used and $t = 500$ for two-dimensional image and protein databases. The RkNN was evaluated by querying for 100 random query points for different values of $k$.

The I/O and the running times are given in Figures 6-12 and 6-13. The cost of SS is almost the same for all values of $k$. It increases slightly as $k$ increases due to maintaining cost of the top $k$ closest objects. The costs of RT, Mux and FD increase as $k$ increases since their pruning power drops for large values of $k$. The running times of RT, Mux and FD do not exceed SS as $k$ increases. FD runs significantly faster than others. Depending on the value of $k$, FD runs orders of magnitude faster than RT, SS and Mux. The I/O cost increases much slower for FD. This is because FD adapts to different parameter settings quickly to minimize the amount of disk reads.

Table 3 present the running times of FD and RkNN for 100 query points. While the running time
of RkNN increases at faster rate and is not scalable for higher values of $k$, the running times of FD, including the time taken for the creation of priority table for each $k$, for the same query set is almost constant and is order of magnitude faster than RkNN.

### 6.7.3.3 Evaluation of database size

In this experiment, the performance of FD, SS, Mux, and RT are observed for increasing database sizes. Smaller databases are created from the original two-dimensional image database by randomly choosing 50, 25, and 12.5% of all the vectors. The buffer size is fixed to 10% of the original image database, $k = 10$, and $t = 500$.

The I/O and the running times are given in Figure 6-14. As $R$ and $S$ grows, the running time of FD increases almost linearly. This is because when both databases are doubled, the average number of candidate pages per row in the PT stays almost the same. On the other hand, the total running time of SS increases quadratically since it has to compare all pairs of data points. The running time of RT is dominated by I/O cost and increases faster than that of FD and slower than that of SS. Like SS, the running time Mux increases quadratically since it fills the buffer
with blocks from \( S \) and is dominated by I/O costs. Thus, the speedup of FD over SS, Mux and RT increases as database size increases. This means that the proposed method scales better with increasing database size.

### 6.7.3.4 Evaluation of the number of dimensions

In this experiment, the performance of FD, SS, Mux, and RT are observed for increasing number of dimensions. Databases of \( d = 2, 4, 8, 16 \) dimensions are created by choosing the first \( d \) values of the feature vectors from the original 60-dimensional image databases. The buffer size is fixed to 10\% of the total size of \( R \) and \( S \), \( k = 10 \), and \( t = 500 \). The I/O and the running times are given in Figure 6-15. As the number of dimensions increases, the running time of SS increases linearly. On the other hand, the running times of RT and Mux increases faster. This is also known as the *dimensionality curse*. For all the methods CPU time increases with the increase in dimension and is significantly larger for 16 dimensions. However even at 16 dimensions FD is 1.3 times faster than the sequential scan, up to 3.5 times faster than RT and up to 1.2 times faster than Mux-Index.
CHAPTER 7
CONCLUSION

Similarity search in database systems is becoming an increasingly important task in modern application domains such as artificial intelligence, computational biology, pattern recognition and data mining. With the evolution of information, applications with new data types such as text, images, videos, audio, DNA and protein sequences have began to appear. Despite extensive research and the development of a plethora of index structures, similarity search is still too costly in many application domains, especially when measuring the similarity between a pair or objects is expensive.

In this dissertation, new indexing techniques to improve the performance of similarity search are proposed. Given a metric database and a similarity measure, the queries we consider are classified under two categories: similarity search and similarity join queries. Several novel search and indexing strategies are presented for each category.

Chapter 4 considered the problem of similarity search in static databases with complex similarity measures. A family of reference-based indexing techniques were developed. Two novel strategies were proposed for selecting references. Unlike existing methods, these methods select references that represent all parts of the database. The first one, Maximum Variance (MV), maximizes the spread of database around the references. The second one, Maximum Pruning (MP), optimizes pruning based on a set of sample queries. Sampling methods were used to improve the running times of the index construction. A novel approach to assign the selected references to database objects was also proposed. The method maps of different sets of references to each database object dynamically rather than using the same references. According to our experiments, our methods perform much better than existing strategies. Among the our methods, Maximum Pruning with dynamic assignment of reference sequences performed the best. The total cost (number of sequence comparisons) of our methods was up to 30 times less than its competitors.

Chapter 5 considered the problem of similarity search in dynamic databases with complex similarity measures. For dynamic databases with frequent updates, two incremental versions of
the MP-Algorithm, the Single Pass (SP) and Three Pass (TP) algorithms were proposed. Since neither algorithm re-computes the index from scratch, both must make assumptions involving the change (or lack thereof) of the various gain statistics used by the MP algorithm over time.

Experiments suggested that depending upon the application characteristics, either MP or one of the incremental methods may be superior. For distance metrics of intermediate computational cost such as the ED, the incremental methods such as SP and TP seemed preferable. On the other hand, for distance metrics of high computational cost such as EMD, applying MP is preferred. If an incremental algorithm is selected, choosing from among SP and TP is somewhat difficult. The latter generally had superior query performance – though not over the DNA data set with a random ordering. The former had better insert processing performance. A rule of thumb might be that if inserts are more common, choose SP, if queries are more common, choose TP.

Chapter 6 considered the problem of similarity join queries in large databases. A new database primitive called Generalized Nearest Neighbor (GNN) was proposed. GNN queries can answer a much broader range of problems than the $k$-Nearest Neighbor query and its variants the Reverse Nearest Neighbor and the All Nearest Neighbor queries. Based on the available memory and the number of nearest-neighbors, either CPU or I/O time can dominate the computations. Thus, one has to optimize both I/O and CPU cost for this problem.

Three methods were proposed to solve GNN queries. These methods arrange the given two databases into pages and compute a priority table for each page. The priority table ranks the candidate pages of one database based on their distances to the pages from the other database. The first algorithm, FA, uses pessimistic approach. It fetches as many candidate pages as possible into available buffer. The second algorithm, FO, uses optimistic approach. It fetches one candidate page at a time. The third algorithm, FD, dynamically computes the number of pages that needs to be fetched by analyzing past experience. Three optimizations, column filter, row filter and adaptive filter were proposed to reduce the solution space of the priority table. Packing and partitioning strategies which provide significant performance gains were also developed.
These optimizations reduce the CPU cost of the $k$-NN searches and eliminates additional I/O costs by pruning the MBRs which do not have a $k$-NN.

According to the experiments, FA is the best method when the buffer size is large and FO is the best method when the buffer size is small. FD was the fastest method in most of the parameter settings. Even when it was not the fastest, the running time of FD was very close to that of the faster of the FA and the FO. FD was significantly faster compared to sequential scan and the standard R-tree based branch-and-bound $k$-NN solution to the GNN problem.
REFERENCES


[31] Dantzig, G.B.: Application of the simplex method to a transportation problem. In Activity Analysis of Production and Allocation p. 359373 (1951)


BIOGRAPHICAL SKETCH

Jayendra Gnanaskandan Venkateswaran was born and brought up in Chennai, India. He received his Bachelor of Engineering from Coimbatore Institute Of Technology (CIT), one of the most prestigious and oldest engineering college in India, in 2001. Jayendra majored in computer engineering and obtained a distinguished record. Jayendra received his Master of Science from University of Missouri-Rolla in May, 2003. He majored in computer science. His master’s thesis was on packing methods for SR-Tree index structure. Along with his masters’ advisor, Dr. S.R.Subramanya, he has published his work at the 21st Annual ACM Symposium on Applied Computing (SAC), Dijon, France, April 2006.

Jayendra joined the Doctor of Philosophy (Ph.D) program in Computer and Information Science and Engineering at the University of Florida-Gainesville in the fall 2003. While pursuing his graduate degree, Jayendra worked as a graduate research assistant. He received his Doctor of Philosophy in Computer Engineering from the University of Florida in December 2007. Along with his advisors Dr. Tamer Kahveci and Dr. Christopher Jermaine, he has published his research at the top database conferences Very Large DataBases (VLDB), Extended Database Technology (EDBT) and VLDB Journal.

Jayendra’s research focus is in the area of database searches, with special interests in database indexing and querying, text mining, algorithms and bioinformatics.