Thesis

Scalable approximate k-NN in multidimensional big data

MOHAMED, Hisham

Abstract

This thesis studies the scalability of the similarity search problem in large-scale multidimensional data. Similarity search, translating into the neighbour search problem, finds many applications for information retrieval, visualization, machine learning and data mining. The current exponential growth of data motivates the need for approximate and scalable algorithms. In most of existing algorithms and data-structures, there is a trade-off to be found between efficiency, complexity, scalability and memory efficiency. To address these issues, we explore recent techniques for similarity search. One remarkable recent technique is the Permutation-Based Indexing. Data objects are represented by a list of pivots, ordered with respect to their distances from the object. The similarity between two objects is then estimated based on these lists, following the idea that neighbouring objects have the same neighbourhood. In this thesis, we introduce a formal representation of the permutation-based indexing model. In particular, we introduce different strategies for selecting the pivots, pursuing the goal of high efficiency and low [...]

Reference

MOHAMED, Hisham. *Scalable approximate k-NN in multidimensional big data*. Thèse de doctorat : Univ. Genève, 2014, no. Sc. 4712

UNIVERSITÉ DE GENÈVE FACULTÉ DES SCIENCES
Département d'Informatique Professeur Stéphane Marchand-Maillet

Scalable Approximate k-NN In Multidimensional Big Data

THÈSE

présentée à la Faculté des sciences de l'Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par

Hisham MOHAMED
du
Caire (Égypte)

Thèse no 4712

GENÈVE
Repro-Mail - Université de Genève
2014

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Doctorat ès sciences
Mention informatique

Thèse de *Monsieur Hisham MOHAMED*

intitulée :

"Scalable Approximate *k*-NN In Multidimentional Big Data"

La Faculté des sciences, sur le préavis de Monsieur S. MARCHAND-MAILLET, professeur associé et directeur de thèse (Département d'informatique), Monsieur S. VOLOSHYNOVSKYY, professeur associé (Département d'informatique), Monsieur P. ZEZULA, professeur (Faculty of Informatics, Masaryk University, Brno, Czech Republic) et E. BRUNO, docteur (Firmenich S.A., Genève, Suisse), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 8 septembre 2014

Thèse - 4712 -

Le Doyen

N.B. - La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".

To my family....

Acknowledgment

It would be impossible to write this thesis without the help and support of many people around me. This work would not be possible without the help, support, guidance and encouragement of my supervisor, Prof. Stéphane Marchand-Maillet. I am really thankful to him for the friendly working environment and for being patient with me.

I would like to thank my lab-mates in the Computer Vision Multimedia Laboratory for their constructive comments during the weekly CVML meetings and for the nice memories we had. Special thanks to my jury committee, Prof. Pavel Zezula, Dr. Eric Bruno and Prof. Sviatoslav Voloshynovskiy for their suggestions and comments. I will forever be thankful to Prof. Moustafa Ghanem. Thanks for all your support. Rest in peace, hope you are in a better place now.

I would like to thank my parents and my sister for their ultimate support and patience. Last but not the least, I would like to thank my wife, Aziza Merzouki for being patient, supportive and for her tolerance with me.

I would like to thanks Allah, for allowing me to finish this work. Any missing or wrong information in this work is entirely my own responsibility.

Abstract

This thesis studies the scalability of the similarity search problem in large-scale multidimensional data. Similarity search, translating into the neighbour search problem, finds many applications for information retrieval, visualization, machine learning and data mining. The current exponential growth of data motivates the need for approximate and scalable algorithms.

In most of existing algorithms and data-structures, there is a trade-off to be found between efficiency, complexity, scalability and memory efficiency. To address these issues, we explore recent techniques for similarity search. One remarkable recent technique is the Permutation-Based Indexing. Data objects are represented by a list of pivots, ordered with respect to their distances from the object. The similarity between two objects is then estimated based on these lists, following the idea that neighbouring objects have the same neighbourhood.

In this thesis, we introduce a formal representation of the permutation-based indexing model. In particular, we introduce different strategies for selecting the pivots, pursuing the goal of high efficiency and low query response time. We propose a scalable, fast and memory-efficient data structure for nearest neighbour search. We provide models for permutation-based indexing on shared memory architecture, including CPU and GPU. In addition, we propose several distributed models for permutation-based indexing using MPI and MapReduce. In doing so, we provide an enhanced programming model using MPI to speedup further the MapReduce strategy. We analyse the proposed techniques and algorithms using standard public large-scale datasets containing millions of objects, and give comparisons to the most recent data structures and algorithms for similarity search.

Résumé

Cette thèse étudie la scalabilité du problème de recherche par similarité sur des données multidimensionnelles à grande échelle. La recherche par similarité, traduite en un problème de recherche de voisinage, trouve de nombreuses applications dans les domaines de la recherche d'informatique, la visualisation, l'apprentissage automatique et l'exploration de données. La croissance exponentielle de la quantité de données motive le besoin d'algorithmes approximatifs et scalables.

Dans la plupart des algorithmes et structures de données existants, un compromis doit être trouvé entre l'efficacité, la complexité, la scalabilité et l'efficacité en terme de mémoire. Afin d'aborder ces questions, nous explorons les techniques récentes de recherche par similarité. Une technique importante est l'indexation basée sur les permutations. Chaque donnée est représentée par une liste de pivots, ordonnés en fonction de leurs distances à l'objet. La similarité entre deux objets est alors estimée sur la base de ces listes, selon l'idée que des objets voisins partagent le même voisinage.

Dans cette thèse, nous introduisons une représentation formelle du modèle d'indexation basée sur les permutations. En particulier, nous introduisons différentes stratégies de sélection des pivots, dans le but d'atteindre une efficacité élevée et un faible temps de réponse aux requłtes. Nous proposons une structure de données scalable, rapide et efficace en termes de mémoire pour la recherche des plus proches voisins. Nous fournissons des modèles d'indexation basée sur les permutations sur des architectures à mémoire partagée, y compris CPU et GPU. De plus, nous proposons plusieurs modèles distribués pour l'indexation basée sur les permutations, utilisant MPI et MapReduce. Nous fournissons aussi un modèle de programmation amélioré utilisant MPI afin d'accélérer la stratégie MapReduce. Nous analysons les techniques et algorithmes proposés en utilisant des ensembles de données à grande échelle, standards et public, contenant des millions d'objets, et nous les comparons aux structures de données et algorithmes de recherche par similarité les plus récents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Big Data

Nowadays, due to the growth of the technology, there is a mass production of data, available in digital form (high volume). This current available data is not unified, it appears in different formats and types. The diversity of the data is based on the type of information it contains such as documents, text, image, video and audio or even on its source such as data coming from sensors (high variety). Furthermore, with the expansion of internet and the World Wide Web, the majority of this data becomes publicity available for a wide range of users (high velocity). That creates a great demand to store, analyse and manage this Big Data (high volume, high velocity, and/or high variety) to offer benefits for users. Nevertheless, this remains challenging, due to the volume and the variety of the data. Current research and methodologies classify the available data into two main domains, namely *structured data* and *unstructured data*.

The term *structured data* generally refers to the data that has a predefined length and format. Until a recent time, the structured data was the only way to store information. This data is usually stored in traditional databases (relational databases and spreadsheets), where the data is stored in tables. The relations between these tables are defined through a database schema. Using a structured query language (SQL), these traditional databases can be explored easily in an efficient way.

The term *unstructured data* refers to the data that does not have a specified consistent format. Unstructured data is found everywhere and we are dealing with it everyday. Now, it represents the majority of the available data. Example of unstructured data includes images, videos, audio, scientific data, sensor data and social media data. The domination and the size of this unstructured data gives a great motivation to manage and organize it based on its content, which is known as content-based management systems (CBMS). CBMS is composed of two computationally expensive operations. The first operation is *feature extraction*. Feature extraction aims to transform the unstructured data into numerical representation usable for other processing. An important assumption of the feature extraction process is that the extracted features will carry sufficient information to represent the content adequately and to enable the compu-

tation of distance for measuring content similarity. The second operation is a combination of storing, analysing and searching the data based on its content (features). Content-based search becomes a daily used operation by a wide range of users in different domains. This operation relies on the notion of similarity for search.

Similarity search aims to extract the most similar objects to a given query. Translating the similarity search into the nearest neighbour (NN) search problem finds many applications for information retrieval, visualization, machine learning and data mining. The context of large-scale unstructured data imposes to find approximate solutions. Approximate similarity search relaxes the search constrains to get acceptable results at a lower cost (eg. computation, memory, time).

Permutation-based indexing is one of the most recent techniques for approximate similarity search in large-scale domains. The data objects are represented by a list of their closest pivots, which are ordered with respect to their distances from the object. The same principles are applied in many domains such as machine learning.

As the data reaches the petabyte-scale in many domains, the current sequential approximate similarity algorithms, whatever their efficiency, cannot handle such volumes. Hence, the role of parallel and distributed computing in all of its forms becomes more important than before by offering large-scale processing capabilities.

In this work, we focus of the problem of approximate k-NN in terms of effectiveness and scalability based on permutation-based indexing and parallel computing. The current state of the art work has not proposed a sufficient answer to this problem and it is still under study from different perspective. Hence, we aim by this work to propose some solutions that improves the scalability and efficiency of approximate similarity in multidimensional big data based on permutation-based indexing.

1.2 Thesis Layout

The thesis is divided into two parts. In the first part, we review the state of the art of similarity search. We then propose our techniques and algorithms with evaluation with large-scale standard datasets. More specifically:

- Chapter 2 is divided into two parts. The first part provides a review on the metric model of similarity, the principles of indexing and searching, the principles of approximate similarity search and the related data structures. The second part is dedicated to permutation-based indexing and its related data structures. Then, at the end of the chapter, we detail all the datasets and the evaluation techniques that will be consistently used in the thesis.

- Chapter 3 describes our perspective on the encoding model for permutation-based indexing. In addition, we propose strategies for selecting the reference points. We then evaluate our proposed

strategies on synthetic and real datasets.

- Chapter 4 proposes two novel fast and effective data structures to handle the permutation lists, which are the *Metric Suffix Array* (MSA) and the *Metric Permutation Table* (MPT). We evaluate our proposed data structures via a series of experiments using large datasets and compare them with the current existing approaches.

In the second part, we review the basics of parallel computing. We then propose multicore and distributed algorithms for our proposed techniques. More specifically:

- Chapter 5 is divided into two parts. The first part reviews the concepts of parallel computing on shared and distributed memory architecture with an overview on all the related programming models such as MPI, MapReduce, OpenMP and CUDA. The second part proposes a novel optimization data exchange policy for MapReduce using MPI programming model. We then evaluate our proposed policy on large scale datasets comparing to the recent implementation of MapReduce based on different applications.

- Chapter 6 proposes different algorithms and techniques to construct the permutation lists using multi-core architectures (CPU and GPU). These proposed algorithms can be adapted to any data structure or algorithm that are based on permutation-based indexing. In addition, we propose a multi-core implementation of our MPT data structure. We then evaluate our proposed models on standard large scale datasets.

- Chapter 7 proposes different algorithms and techniques to construct the permutation lists on a distributed environment based on different programming models (including our enhanced implementation of MapReduce). For evaluation, we apply our proposed algorithms on two different data structures using large scale datasets.

- Chapter 8 shows our experience with building a prototype for distributed large scale similarity search system for big databases using our proposed data structures.

- The work is then concluded in Chapter 9 by discussing our contributions and giving an overview of the perspectives of the work.

1.3 Thesis Contributions

Our aim is to design scalable data structures and algorithms for approximate similarity search that can work efficiently and effectively for big datasets. At the same time, these structures have to suitable for various types of applications and domains. In order to meet these objectives, this thesis proposes:

- A detailed guidance of objects encoding using permutation with different methodologies to select the permutations (Chapter 3).
- Two novel data structures that are based on metric spaces, that makes it extensible and suitable for a wide range of applications and algorithms (Chapter 4). This work is published in [1, 2, 3].
- An enhanced MapReduce model based on MPI (Section 5.4). This work is published in [4, 5, 6].
- Multicore (CPU and GPU) strategies for permutation-based indexing (Chapter 6). Part of this work is published in [7].
- Distributed strategies for permutation-based indexing that makes it scalable (Chapter 7). This work is published in [8, 9].
- A competitive study with the current existing approaches based on public large-scale datasets.

Chapter 2

Related Work

In this chapter, we review the basics of similarity search and its related techniques. In addition, we explain in detail the data structures that we will use as baseline to compare to our proposed algorithms in the next chapters. The chapter is organized as follows. In section 2.1, we give an introduction about metric spaces and similarity queries. In section 2.2, we review the exact search technique and one of its most famous data structure, namely the k-d Tree, and show why it is not suitable for high dimensional datasets. In section 2.3, we give an introduction about approximate similarity search and discuss the available approximation techniques. In section 2.4, we give an introduction about the permutation-based indexing with its related data structures. In section 2.6, we explain our evaluation strategy for all the experiments done later in the thesis and present the datasets that will be used for evaluating our proposals.

2.1 Similarity Search

Similarity search [10] aims to extract the objects most similar to a given query. It is a fundamental operation for many applications, such as information retrieval, machine learning and computer vision. Many associated software are available [11, 12, 13, 14].

Similarity search is performed by ranking the database objects with respect to their similarity or dissimilarity to the query. The similarity or the dissimilarity between two objects is defined through some distance function, where low values correspond to a high degree of similarity [15, 16]. This distance measure generally (at least to some extent) follows the *metric space postulates*.

2.1.1 Metric Spaces

Definition 1. *Given a metric space $\mathcal{M} = (\mathcal{D}, d)$, where $\mathcal{D}$ is a domain of objects and d is a distance function $d : \mathcal{D} \times \mathcal{D} \rightarrow \Re$ that follows the metric space postulates, $\forall x, y, z \in \mathcal{D}$:*

$$\textit{identity}: \quad x = y \iff d(x, y) = 0,$$

$$\textit{non-negativity}: \quad d(x,y) \geq 0,$$

$$\textit{symmetry}: \quad d(x,y) = d(y,x) \quad \textit{and}$$

$$\textit{triangle inequality}: \quad d(x,z) \leq d(x,y) + d(y,z)$$

The distance function quantifies the proximity of objects in a given space.

2.1.2 Distance Functions

The distance function is specified according to the application and it is well surveyed in [15]. The most well known distances are the *Minkowski Distances* and the *Edit Distance*.

Minkowski Distances

The Minkowski Distances form the L_p metric functions, which are suitable for vectors of any dimension and defined by the parameter p. Given m-dimensional vectors $\mathbf{x} = (x_1, \ldots, x_m)$ and $\mathbf{y} = (y_1, \ldots, y_m)$, these metric distance functions are defined as follows:

$$L_p(\mathbf{x}, \mathbf{y}) = (\sum_{i=1}^{m} |x_i - y_i|^p)^{\frac{1}{p}}, \tag{2.1}$$

Where L_1 is the *Manhattan distance*, L_2 is the *Euclidean Distance* and L_∞ is the *Maximum Distance*. Figure 2.1 illustrates the L_1, L_2 and L_∞ distances with their unit ball in $\Re^2$.

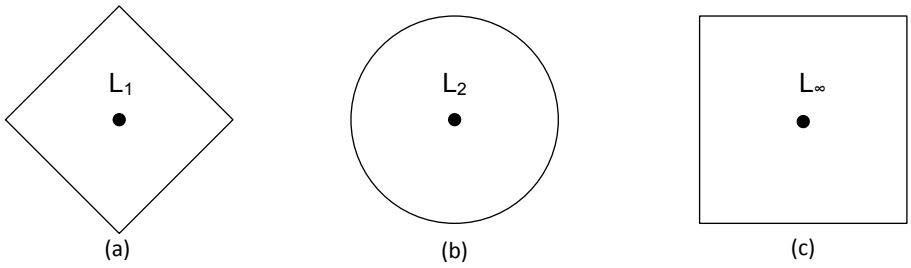


Figure 2.1: The set of points at a constant distance from the central point for L_1, L_2 and L_∞ distance functions respectively.

Edit Distance

The Edit Distance d_{edit} function is suitable for string and sequence comparison [17]. Given two sequences $\mathbf{x}$ and $\mathbf{y}$ of lengths n and m, respectively. The distance between $\mathbf{x} = (x_1 \ldots x_n)$ and $\mathbf{y} =$

$(y_1 \dots y_m)$ represents the minimum number of symbols that have to be *inserted*, *deleted* and *substituted* to transfer **x** onto **y**. Each operation has a cost: w_{ins}, w_{del} and w_{sub} represent the *insertion*, *deletion* and *substitution* costs, respectively. The distance between two sequences is the minimum possible sum of the costs of the set of operations mapping **x** onto **y**.

For example, given two words **x** = *'combine'* and **y** = *'combination'*, where $w_{ins} = 1$, $w_{del} = 1$, $w_{sub} = 3$, the optimal Edit Distance $d_{edit} = 6$, because 1 deletion (deleting 'e') and 5 insertions (adding 'ation') transforms **x** into **y**. We may also transform **x** into **y** by a cost 7 with 1 substitution (replacing 'e' with 'a') and 4 insertions (adding 'tion'), but this is not the optimal distance.

2.1.3 Similarity Queries

A similarity query is based on a query object q submitted to the database to rank objects in their order of decreasing similarity. The criterion for getting the similar objects is mapped onto proximity, as measured by the distance function $d(,)$. The number of these retrieved objects is based on the search scenario. There are different type of queries, which are well discussed in [18, 15]. The most common search scenarios are the *Range Query* and *k-Nearest Neighbour Query*.

Range Query

The range query $R(q, \rho)$ aims to retrieve all the similar objects to query q within a given distance range ρ, formally:

$$R(q, \rho) = \{o \in \mathcal{D} \text{ s.t } d(q, o) \leq \rho\}$$

These retrieved objects are ranked with respect to their distance from the query. Figure 2.2(a) shows an example of a range query.

k-Nearest Neighbour Query

Specifying a certain range for search requires information about the scale of the domain. In addition, it is difficult if the statistical distribution of the data is not precisely known. A common alternative is the *k-Nearest Neighbour Query (k-NN)*. *KNN* aims to find the k most similar objects to q, formally:

$$\textit{K-NN}(q, k) = \{S \subset \mathcal{D} \text{ s.t } |S| = k \text{ and } d(q, o_i) \leq d(q, o_j), \forall o_i \in S, o_j \in \mathcal{D} \setminus S\}$$

Unlike the range query procedure, this technique retrieve all the closest k objects, whatever their distance range from the query. Figure 2.2(b) shows an example of K-NN query.

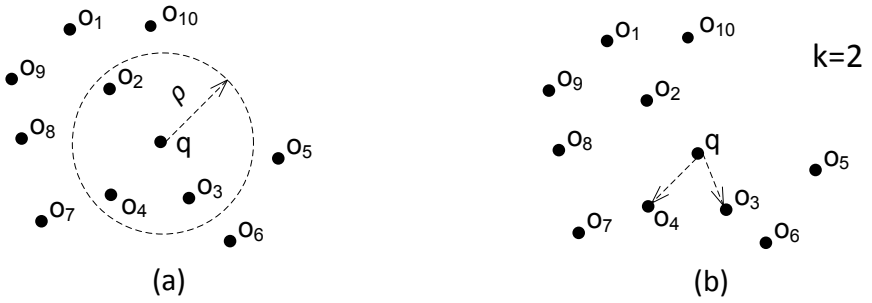


Figure 2.2: a) Range Query $R(q, \rho)$ b) k-Nearest Neighbour Query for $k = 2$; *K-NN*$(q, 2) = \{o_3, o_4\}$

2.2 Exact Similarity Search

The naive (baseline) search technique is called the *exhaustive search* approach. For a given query, the distance between the query and all the database objects is calculated while keeping track of which objects are similar (close) to the query.

There are two challenges with this technique. The first problem is that the distance function can be computationally expensive, especially if the dimension m increases. The second problem is the scalability. If the database size N increases, since all the objects need to be scanned against the query, this is not applicable with large scale datasets.

Hence, several data structures were proposed to decrease the number of objects to compare to the query. These data structures are based on space partitioning and decomposition techniques. These techniques most often record the space structure hierarchically in trees. The most common such trees are the *k-d tree* [19], the LSD tree [20] and BV-Tree [21].

2.2.1 k-d Tree

A *k-d tree* is a space-partitioning data structure for organizing objects in a *k*-dimensional space. The *k-d tree* is similar to the binary search tree (BST) [22] in its structure, except for the splitting plane. The binary search tree is built for one dimensional objects and the splitting plane is done with respect to one dimension. In contrast, the *k-d tree* alternates among the available m dimensions. The nodes from the first level of the tree are split into left and right with respect to the first dimension, the second level with respect to the second dimension and so on.

A *k-d Tree* has two types of nodes, *internal* and *leaf* nodes. An internal node of the *k-d tree* has two values, the splitting dimension, and the splitting value. If the splitting dimension is x_1, then all points whose x_1 coordinates are less than the splitting value are stored in the left sub-tree and in the right sub-tree if it is larger than the splitting value. Internal nodes are used to store the splitting information only. The objects are stored in the leaves of the tree. Figure 2.3 shows an example of *k-d tree* for set of 8 points

in 2D.

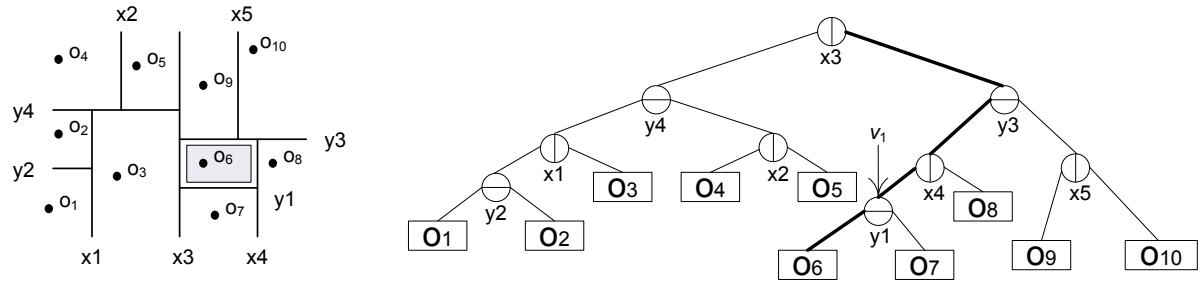


Figure 2.3: *k-d Tree* for 8 points in two dimensions. Horizontal and vertical lines in the nodes refer to the direction of splitting. The splitting values $x_1, x_2, ..$ and $y_1, y_2, ..$ attached to each node divide the plane by vertical and horizontal lines.

The $k-$NN query in *k-d tree* can be answered by top-down traversal of the tree in the same way that it would be inserted. As we explained before due to the alternative splitting, each node corresponds to the smallest hyper-rectangular region bounding for the points under it. As shown in figure 2.3, the region bounding the points under the node v_1 is determined by the splitting values x_4, y_3 and x_3 stored at the ancestor nodes. The region bounding the leaf o_6 can be identified by reading the splitting values y_1, x_4, y_3 and x_3 of the four ancestors nodes.

2.2.2 Curse Of Dimensionality

Several studies [23, 24, 18] pointed out that using the above type of structure for exact similarity search is not efficient for high dimensional data (eg. $m > 10$). The efficiency decreases in high dimensional datasets, due to the *curse of dimensionality* [25].

Organizing and searching data often relies on detecting areas where objects form groups with similar properties. When the dimension increases, the volume of the search space increases, which leads to make the data becomes sparse. Accordingly, exact search methods such as *k-d Tree* becomes less local in high dimensional spaces. This means that the k-NN points are less concentrated and it becomes harder to group them based on space partitioning.

For example, assuming a uniform distribution of objects to construct a hypercube around an object **x** to get a fraction φ of the observation space, the edge length of the cube should be $l = \varphi^{\frac{1}{m}}$ (such that the volume $\varphi = l^m$). If $m = 1$ and $\varphi = 10\%$, then $l = 0.1$. That means that in order to capture 10% of the related data, the searching algorithm should cover 10% of the range of each coordinate. If $m = 10$, then $l = 0.79$. If the dimension increases to 10, the searching algorithm should cover 79% of the range of each coordinate [26]. In other words, as the dimension increases, a large part of the database needs to be scanned. Hence, the performance becomes similar to that of exhaustive search.

In [27], the authors claim that the *curse of dimensionality* affects the quality of similarity search.

Given a point q, the relative variance of the distance value to its neighbours points converges to zero when the dimension m goes to infinity

$$\lim_{m \to \infty} \frac{Var^2[d(q,x)]}{E[d(q,x)]} = 0$$

In a high dimensional space, when this ratio is satisfied, if the distance to the nearest neighbour is 1, then the distance to the farthest nearest neighbour is $1+\varepsilon$, where ε is an arbitrarily small positive value. Hence, the nearest neighbour search may be meaningless in very high dimensional space due to computational imprecision.

In order to accelerate and improve the search, there has been proposals for *approximate* search solutions. Such proposals aim to provide fast and scalable response time while accepting some imprecision in the results. This is known as the approximate similarity search [28, 29, 30].

In [31], the authors propose to measure the *intrinsic dimensionality* γ in metric spaces for approximate similarity search:

$$\gamma = \frac{E^2[d(q,x)]}{2Var^2[d(q,x)]}$$

The intrinsic dimensionality grows with the mean and decreases with the variance of the distances between points distributed uniformly in the metric space. More details on the use of this intrinsic dimensionality in metric spaces are in available in [31].

2.3 Approximate Similarity Search

Nowadays, approximate similarity search is the most commonly used technique for searching in multidimensional spaces. Most similarity search systems are based on distance function that do not fully represent the similarity, as the human brain does [32]. As a results, in modern systems, relevance feedback may be used to improve interactively the efficiency of similarity. Hence, this is an example where the searching time at each feedback iteration should not be too long, in order to keep the user focused on the previous iterations of the search, and as a result, that improves the quality of the results [9, 33].

The goal of approximate similarity search is to relax the search constrains to get acceptable results at a lower cost (computation, memory, time). Current approximate similarity search algorithms are sorted into three categories, which are the *Space Transformation*, *Space Partitioning* and *Pre-Computed Distances*. These three techniques are based on computing distances. Therefore, these strategies may be used within metric spaces of any dimension. However, they are fragile to the curse of dimensionality. These techniques are discussed in detail, with their related data structures, in the next sections.

2.3.1 Space Transformation

The aim of this technique is to change the metric space into another, where searching is much simpler. The transformation may be done by transforming the data space or by transforming the distance function, or both of them. There are several algorithms based on this technique such as *Locality-Sensitive Hashing* (LSH) [34, 35, 28], *FastMAP* [36], *MetricMAP* [37], *Integrated Progressive Search* [38] and *VA-File* [24].

Local Sensitivity Hashing (LSH)

Locality-Sensitive Hashing was first proposed in [34] based on random projections, [39, 40]. The main idea is to map each object onto a hash value (based on a hash family), so that the objects that are close to each other have a higher probability of colliding (to be located in the same bucket) than the objects that are far from each other.

Definition 2. *A function family $\mathcal{H} = \{h : \mathcal{D} \rightarrow U\}$ is called (ρ,$c\rho$,$\mathcal{P}_1$,$\mathcal{P}_2$)-sensitive for distance function d if for any $o, q \in \mathcal{D}$*

$$\textit{if } d(o,q) \leq \rho \textit{ then } \mathcal{P}_r[h(q) = h(o)] \geq \mathcal{P}_1,$$

$$\textit{if } d(o,q) > c\rho \textit{ then } \mathcal{P}_r[h(q) = h(o)] \leq \mathcal{P}_2$$

where, ρ is a distance value, c is a constant and $\rho < c\rho$ and $0 > \mathcal{P}_1 > \mathcal{P}_2 > 1$.

Close objects within a distance ρ should have a high chance to be hashed with the same hash value than the objects that are located at a greater distance $c\rho$. In [41], authors proposed the LSH family for L_p norms (section 2.1.2) based on p-stable distribution (Gaussian distribution). Each function is defined as:

$$h_{a,b}(v) = \lfloor \frac{a.v + b}{w} \rfloor \tag{2.2}$$

where a is a random vector chosen from a p-stable distribution, b is a real number chosen from the range $[0, w]$ and v is the vector that will be hashed. Once the hash family $\mathcal{H}$ is ready, it can be used to index the database objects for efficient searching.

The basic LSH indexing algorithm works as follows [34].

- Define an integer k and a set of hash functions $\mathcal{G} = \{g : \mathcal{D} \rightarrow U^k\}$ built on the combination of k hash functions ($h : \mathcal{D} \rightarrow U$). For $g \in \mathcal{G}$, $g(v) = (h_1(v), \ldots, h_k(v))$, where $h_i \in \mathcal{H}, 1 \leq i \leq k$. In other words, a function g is a concatenation of k hash functions k_i. The main motivation for this concatenation is to improve the collision probability.

- Define an integer z and choose uniformly z functions $g_i \in \mathcal{G}$. Each chosen g_i function is used to construct one hash-table. As a result, z hash tables are created.

- For each database object o, compute the hash value $g_i(o)$, for $i = 1, \ldots, z$ and insert o into the corresponding hash bucket.

For similarity search, for a given query q, $g_i(q)$ is computed for all $i = 1, \ldots, z$. Then, all the objects o located in the same hash bucket as q are pre-selected and re-ranked by their direct distance calculation $d(q, o)$ to the query.

The main bottleneck is the large number of hash tables that is needed in order to cover the space. Also, the values k and z affect the search quality. Later, several algorithms have been proposed to improve the efficiency of the LSH algorithm such as *Multi-Probe LSH* [42] and *LSH Forest* [35].

LSH Forest [35] was proposed to improve the values of the k and z parameters and the indexing structure. Instead of assigning a fixed hash value of size k to each object, the hash value is not with a fixed length in order to give a distinct representation to each object. Then, a prefix-tree is used to represent all the hash values, where each leaf node points to a distinct object and the hash tables are replaced with the prefix tree. Extending this, the LSH forest consists of z prefix-trees.

2.3.2 Space Partitioning

Techniques based on the *space partitioning* approach aim to reduce the search space. Using the original distances and some probabilistic bounds, parts of the data are eliminated because they are likely to be far from the query object. Space partitioning is done through three different ways, *Pivoting*, *Local Partitioning* and *Pre-computed Distances*.

Pivoting

The main idea is to select a number of pivot points from the space. Using these pivots and the distances between them, the data is categorised in different classes. The basic schema is to choose n pivots $r_j \in \mathcal{D}$. Objects $o_i \in \mathcal{D}$ are classified based on their distances to the pivots. For a query q, all the n distances $d(r_j, q)$ are computed and the search region is identified.

There are two fundamental principles for pivoting, which are the *Ball Partitioning principle* and *Hyperplane Partitioning principle*. The *Ball Partitioning principle* aims to divide the data domain $\mathcal{D}$ into two subsets S_1 and S_2 using a spherical cut with respect to a pivot r_j, where $r_j \in \mathcal{D}$. Each object $o_i \in \mathcal{D}$ is mapped to its corresponding subset, with respect to its distance to the pivot r_j. More formally,

$$o_i \in S_1 \leftarrow \{d(r_j, o_i) \leq \rho\},$$

$$o_i \in S_2 \leftarrow \{d(r_j, o_j) > \rho\},$$

where ρ is the radius of the sphere centred by r_j. The value of ρ is determined depending on the technique (see below). Several data structures follow that approach such as BK-Tree [43], Fixed Queries Array [44]

and Vantage-point trees (VP-Trees) [45, 46].

Hyperplane Partitioning is similar to the ball partitioning technique, but instead of dividing the space using a spherical cut, the space is divided using an hyperplane. Two pivots $r_1, r_2 \in \mathcal{D}$ are chosen. Each object $o_i \in \mathcal{D}$ is mapped to its corresponding subset depending on its distance from the selected pivots. More formally,

$$o_i \in S_1 \leftarrow \{d(r_1, o_i) \leq d(r_2, o_i)\},$$

$$o_i \in S_2 \leftarrow \{d(r_1, o_i) > d(r_2, o_i)\},$$

Several data structures are based on this technique such as the Bisector Tree [47] and the Generalized Hyperplane Tree [46].

VP-Tree

The VP-Tree is based on the *Ball Partitioning principle*. The main idea is to divide the data space into small regions (buckets) based on their distances to pivots r_j, where the buckets are organized in a balanced tree structure.

The algorithm works as follows. For a given dataset $\mathcal{D}$, a pivot point $r_j \in \mathcal{D}$ is chosen randomly. The distance $d(r_j, o_i),\ \forall o_i \in \mathcal{D}$ are computed to define the median value ρ of these distance. The sphere centred at r_j with radius ρ divides the space into two partitions S_1 and S_2 such that, each partition contains approximately half of the database points. The first level of the tree points to these two partitions: the points located in the pivot sphere are placed into the left sub-tree ($d(o_i, r_j) \leq \rho$) and the other points that are placed into the right sub-tree ($d(o_i, r_j) > \rho$). The median strategy is applied recursively on each subset and that leads to a balanced tree. A bucket is a leaf node that contains the points close to a certain pivot. Figure 2.4 illustrates the VP-Tree structure.

For K-NN queries $KNN(q, k)$, the tree is traversed from the root by calculating the distances $d(q, r_j)$ between the query q and the pivot in each level r_j until a leaf node is reached. Then, the objects at the leaf node are ranked based on their actual distance to the query (direct distance calculation).

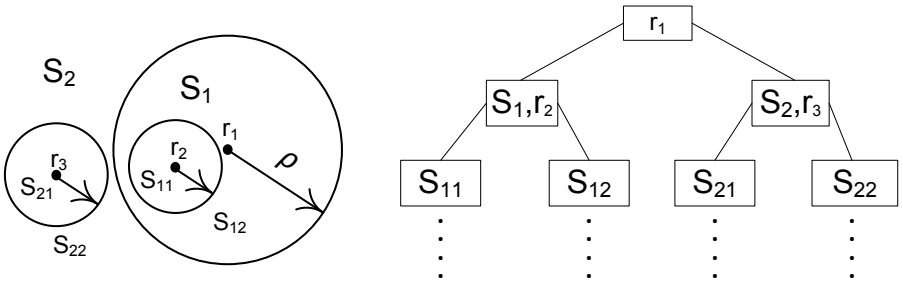


Figure 2.4: VP-Tree

In [48], authors proposed *VPT** as a distributed implementation of the VP-Tree to handle large scale data. The *VPT** is based on peer to peer (*P2P*) [49] paradigm. *P2P* is a decentralised network. Such that, all the peers (compute nodes in the network) offer the same functionality. In addition, there is a communication protocol that facilitates intra-system navigation and data exchange. The core idea of *VPT** is to define the appropriate peers, that has the required data based on *P2P* communication protocol. More details are available in [48].

Local Partitioning

The aim of this approach is to divide the database into groups, where all the elements within a certain group are close to each other. A representative pivot from each group is chosen. At query time, the distance between the query q and all the pivots is calculated, the closest is identified and the objects associated to that representative are ranked with respect to their distance to the query. The M-Tree [50] and M-Index [51] are based on this approach.

M-Tree

The M-Tree is a dynamic tree that supports insertion and deletion. It is a balanced tree that is based on the B-Tree [52, 53]. The M-Tree is composed of two types of nodes, the internal nodes and the leaf nodes. Each internal node stores l tuples, where l is a value predefined by the user. Each tuple consists of four parameters

$$< o_i, \rho_{o_i}, d(o_i, o_i^p), ptr >,$$

where o_i is an object, ρ_{o_i} is the covering radius of the object, $d(o_i, o_i^p)$ is the distance between the object and its parent object o_i^p and ptr is a pointer to a child node.

All the database objects are stored in the leaves. Each leaf node consists of pairs, where each pair contains two parameters

$$(o_i, d(o_i, o_i^p))$$

where o_i is the database object and $d(o_i, o_i^p)$ is the distance between the database object and the parent object. A parent object is the pivot r in the parent node. Hence, the leaf nodes store the database objects, while the internal node store information that guide to the appropriate leaf node, most probably similar to the query.

Figure 2.5 shows a M-Tree of $l = 3$ tuples. The nodes that belong to a certain branch contain all the objects that are located in the covering radius of the parent object. For example, for object o_1, its covering radius is 4.1. All the objects located within this distance from the objects are located in its branch. That is, o_{10}, o_6, o_3.

For a range search $R(q, \rho)$, the algorithm traverses the tree in a depth-first manner using the stored distances between the objects and their parents. All the objects out of the covering radius of the object

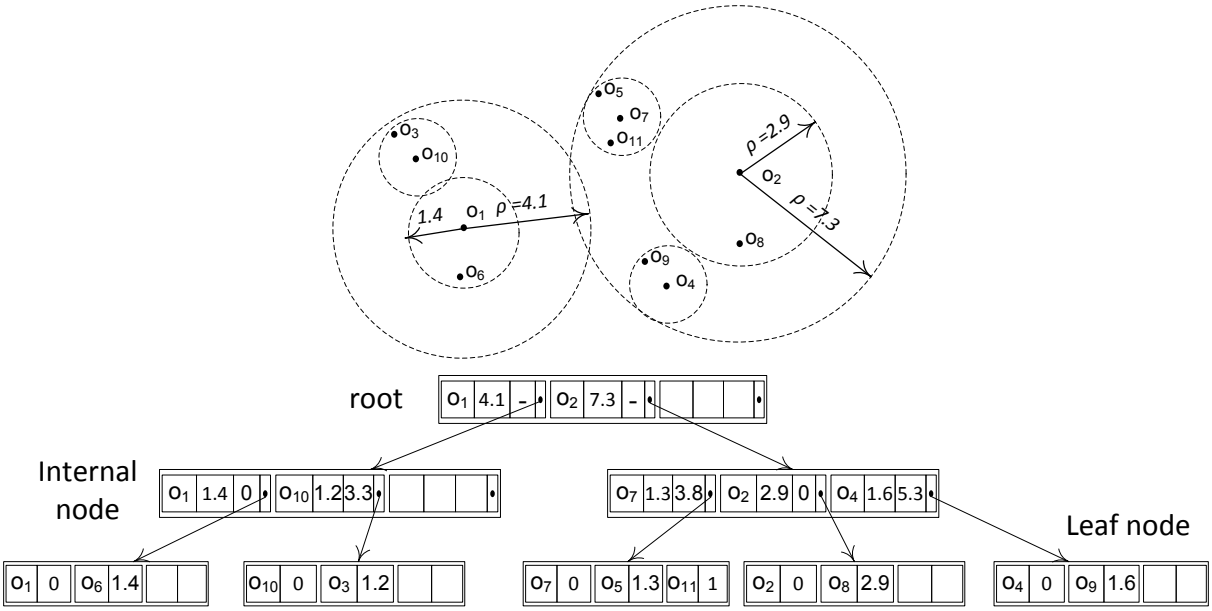


Figure 2.5: M-Tree

are ignored, because they are not located in the range of the query. A sub-tree is ignored if

$$d(q, o_i^p) - d(o_i, o_i^p) > \rho + \rho_{o_i}^p,$$

Otherwise, if,

$$d(q, o_i^p) - d(o_i, o_i^p) \leq \rho + \rho_{o_i}^p,$$

the sub tree is visited. The algorithm continues recursively until it reaches to the appropriate leaf nodes. All the objects in the leaf nodes are checked, based on the saved distance with respect to their parent. An object is ignored if $d(q, o_i^p) - d(o_i, o_i^p) > \rho$. Then, the objects are ranked based on their distance to the query.

The main strength of the M-Tree is that the algorithm reduces the number of distance operation for answering a query, because the distances are locally saved in the tree. That helps to reduce the searching time, but affects the memory.

2.3.3 Pre-computed Distances

As the distance calculation is an expensive operation, some proposals aim to calculate the distances between the database objects and use these pair-wise distances for answering queries. Given a data domain $\mathcal{D}$ of N objects. A table of size $N \times N$ is allocated to store the pairwise distances between all objects in the database. Due to the rapid increase in the data size, $n < N$ of pivots are selected and the pairwise distances between the database objects and these pivots are stored in a table of size $N \times n$. Then,

pairwise distances that are not stored are estimated using the pre-computed distances, while searching. The most known such algorithm is the AESA [54]. The technique is efficient and unbeatable, but it is impractical due to the huge memory requirement.

AESA

The structure is a $N \times N$ matrix holding all the pair-wise distances between all the database objects. Due to the symmetry property of the metric spaces, the matrix is symmetric and the diagonal is 0. Accordingly only $(N-1)(N-2)/2$ distance values are stored. For k-NN similarity search, a random object o_i from the database is selected. The distance $d(q, o_i)$ between the query and the selected object is calculated. An object o_j is ignored if $|d(q, o_i) - d(o_i, o_j)| > \rho$. The process is recursively repeated until only a small set of objects remains. These objects are ranked with respect to their actual distance to the query object. The proposed algorithm is almost quadratic in space. In [55], an enhanced algorithm based on choosing n pivots was proposed.

2.4 Permutation-Based Indexing

Permutation-based indexing (PBI) is a recent technique for approximate similarity search. We consider this approach as a hybrid approach between the *Space Transformation*, *Space Partitioning* and the *Pre-computed Distances* techniques. A Permutation-based index is based on predicting similarity between the database objects according to how they order a distinguished set of anchor objects [56]. PBI selects n reference points and the distance between each object and these references are calculated, similar to AESA. However, permutation-based indexing does not use the distance information between the objects and the pivots while searching. The distance information is just used for the indexing process.

The core idea of permutation-based indexing was proposed by Chavez et al. in [56]. Given a collection of N objects o_i in a domain $D = \{o_1 \dots o_N\}$ and a distance function $d(.,.)$, a set of n reference objects $R = \{r_1, r_2, \dots r_n\}$ is selected from D. Each object o_i is then represented by an *ordered list* $L(o_i, R)$. The ordered list for each object contains the set of reference points sorted by their distance to the object o_i. More formally, $L(o_i, R)$ is the permutation of $(1, \dots, j, \dots, n)$ according to the distance function $d(,)$. In our notation, $L(o_i, R)_{|r_j}$ returns the position of the reference object r_j within the ordered list $L(o_i, R)$. For example, $L(o_i, R)_{|r_j} = 5$ means that r_j is the 5th closest reference point to the object o_i.

The reference points and the ordered lists for the database objects are the only information stored. For a given query q, an ordered list $L(q, R)$ is computed with respect to the reference point set R. The similarity between the query and database objects is measured by comparing the permutation lists.

Comparing two ordered lists can be performed through different ways [57, 58, 59, 60]. In [56], the authors compare three different approaches. They measure the effect of each similarity function on the

retrieval performance. They use *Spearman Rho*, *Kendal Tau*, and *Spearman Footrule Distance* [56, 57]. The *Spearman Rho* (Def.3) and *Spearman Footrule Distance* (Def.4) measure the differences between two ordered lists by adding the differences between the position of every reference in the first and the second ordered list. The *Kendal Tau* (Def.5) calculates the number of swaps that are needed to convert one ordered list into the other.

Definition 3. *Given two ordered lists $L(o, R)$ and $L(q, R)$ and $n = |R|$, Spearman Rho is defined as*

$$S_\rho(o, q) = \sum_{j=1}^{n} (L(o, R)_{|r_j} - L(q, R)_{|r_j})^2$$

.

Definition 4. *Given two ordered lists $L(o, R)$ and $L(q, R)$ and $n = |R|$, Spearman Footrule Distance is defined as*

$$d_{\mathrm{SFD}}(o, q) = \sum_{j=1}^{n} |L(o, R)_{|r_j} - L(q, R)_{|r_j}|$$

.

Definition 5. *Given two ordered lists $L(o, R)$ and $L(q, R)$ and $n = |R|$, Kendal Tau is defined as*

$$d_K(o, q) = \sum_{r_i, r_j \in R} K_{r_i, r_j}(L(o, R), L(q, R)),$$

such that

$$K_{r_i, r_j}(L(o, R), L(q, R)) = \begin{cases} 0, & \text{if } L(o, R)_{|r_i} < L(o, R)_{|r_j} \text{ and } L(q, R)_{|r_i} < L(q, R)_{|r_j} \\ 1, & \text{otherwise} \end{cases}$$

The experiments performed in [56] show that the *Kendal Tau* and *Spearman Rho* give similar performance, slightly better than the *Spearman Footrule Distance*. Due to the complexity of the *Kendal Tau*, they used the *Spearman Rho*. In recent algorithms, *Spearman Footrule Distance* is used as it avoids costly multiplications and the difference in performance with *Spearman Rho* is still not meaningful.

We will study in detail permutation-based indexing. In particularly, a formal model for permutation based indexing is given in chapter 3.

2.4.1 Related Data Structures

Metric Inverted Files (MIF)

Amato and Savino [61, 62] introduced the metric inverted files (MIF). The MIF is the first data structure that was proposed for permutation-based indexing. The MIF is a disk based data structure, that stores

the permutations in an *inverted file* [63, 64].

Inverted files are the backbone of most current text search engines. Inverted files are composed of two elements, namely the *vocabulary* and the *inverted list*. The *vocabulary* is a lexicographically sorted list containing a list of unique terms appearing in the text corpus. Every term is then associated with an *inverted list* (posting list) containing the references to the documents where this term appears. Further, for each document, a weight is given that reflects the importance of the term into the document, in the context of the complete collection. The *TF-IDF* weighting scheme [65] is used in information retrieval and text-mining. This weight is a statistical measure used to evaluate how important a word is to a document in a corpus.

To map that model to the permutation-based indexing, the reference objects from R are acting as the vocabulary. The inverted list for a reference $r_j \in R$ stores the object-id o_i and the ranking of this reference object relative to all objects in the database, $L(o_i, R)_{|r_j}$. Figure 2.6(b) shows an example of the metric inverted files for the ordered lists in Figure 2.6(a).

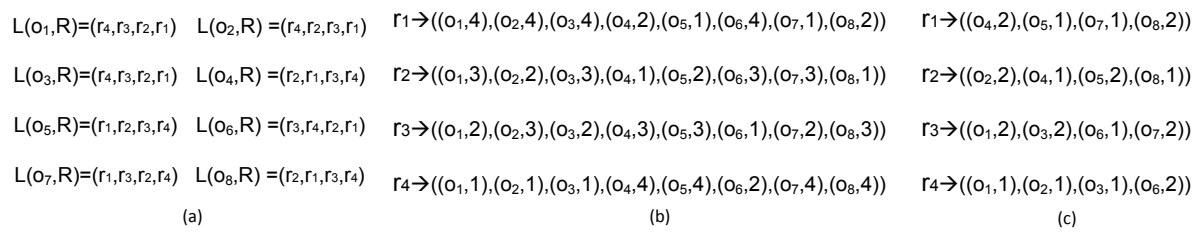


Figure 2.6: a) Ordered lists b) Metric inverted files using full permutation list. c) Metric inverted files using the nearest 2 references.

The similarity between a given query and the database objects is defined sequentially using the SFD function (Def.4). In algorithm 2.1, for a given query q, an accumulator is assigned to each object o_i and initialized to zero (lines 1-2). The posting list δ for every reference point is accessed and the accumulator is updated by adding the difference between the position of the current reference point r_j in the ordered list of the query and the position of the reference point in the ordered list of the objects (saved in the posting list), using the SFD equation (lines 4-8). After checking the posting lists of all the reference points, the objects are sorted based on their accumulator value. Objects with small accumulator values are the most similar to the query object (lines 9-10).

To further reduce the storage, each object is encoded using the only nearest reference points. Accordingly, the length of the posting lists decreases and that improves the searching time. Figure 2.6(c) shows the metric inverted files using the nearest references.

The data structure is efficient and scalable, but still requires 8 bytes for each pair in the inverted list. With large scale databases, the process becomes inefficient.

Algorithm 2.1 Metric Inverted Files: MIF

IN: Query: q,
Reference Object list of n elements: R,
Posting lists assigned to each reference object for N objects;
OUT: Sorted Objects list: out
1. Create a list of accumulators $A[1 \dots N]$
2. Set accumulators values to 0
3. For each $r_j \in R$
4. Let δ be the posting list for the reference object r
5. Set $i \longleftarrow 0$
6. For each pair $(o, L(o_i, R)_{|r_j}) \in \delta$
7. Set $A[i] = A[i] + |L(o_i, R)_{|r_j} - L(q, R)_{|r_j}|$
8. $i \longleftarrow i + 1$
9. Sort(A)
10. $out \leftarrow A$

Brief Index For Proximity Searching

The quality of permutation-based indexing is based on the number of the reference points. However, more references means a larger memory is required. To decrease the memory usage, authors in [66] proposed the *brief permutation index*. The algorithm proposes a smaller representation of the ordered lists, in order to reduce the required memory. After generating the ordered lists $L(o_i, R)$, each list is encoded using fixed size shorter bit-string. Encoded ordered lists are then stored to be used to answer users' queries.

Once a query is received, its order list is also encoded in the same way as the database objects. Then, in a sequential manner, the encoded query vector is compared with the database vectors using the binary Hamming distances [67]. The technique is more efficient in terms of memory and searching time compared to the MIF. In addition, the Hamming distance calculation is cheaper than the SFD. However, its performance is limited in large databases since the Hamming distance still should be computed between the query and all the database objects.

In [68], the same authors proposed a version with LSH. The first step generating the ordered lists for all the database objects and encoding them is the same. Then, using the brief encoding, a family of hashes for each binary string is computed for representing the objects. Finally, LSH is built using this family of hash functions.

To answer a query, the same procedure is followed. First, compute the ordered list and generate the brief binary representation. Then, retrieve all the objects that have the same hash and refine them using the Hamming distance. The efficiency of the algorithm is not high due to the two-level of encoding.

Permutation Prefix Index (PP-Index)

In [69], authors proposed an attractive technique called the *prefix permutation index* (PP-Index). The algorithm aims to decrease the memory usage by reducing the number of reference points and perform some direct distance calculation with only a small portion from the database. The PP-Index only stores a prefix of length l (where $l < |R|$) of the permutations in the main memory in a Trie Tree [22]. The data objects are saved in data blocks on the hard disk. Objects which share the same prefix are saved close to each other in the same data block. Figure 2.7(b) shows the PP-Index data structure for the ordered lists in figure 2.7(a).

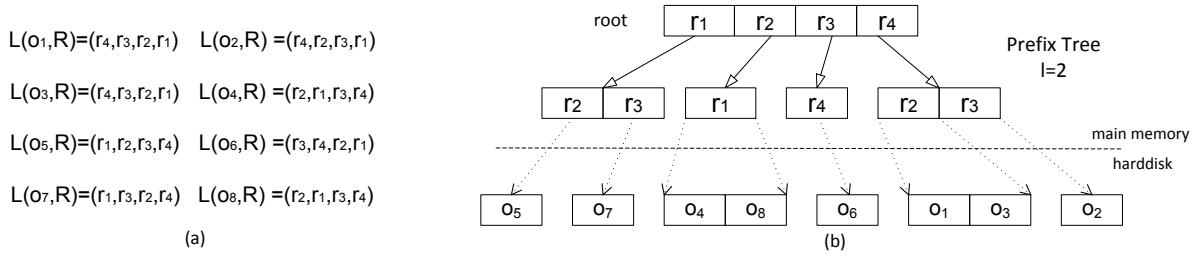


Figure 2.7: a) Ordered lists b) PP-Index data structure for $l = 2$.

When a query is submitted, all the objects in the block sharing the same prefix are retrieved. To rank them, a direct distance calculation is performed between the query and these objects, to sort them with respect to their actual distance value from the query.

The PP-Index requires less memory than the previous data structures and uses the permutations prefix in order to quickly retrieve the relevant objects, in a hash-like manner. The indexing time of the PP-Index data structure is high, essentially due to constructing the Trie Tree.

The proposed technique initially gives a lower recall value. An improvement is made using multiple queries technique. When a query is received, the ordered list for the query is computed and the nearest prefix permutation of length l is defined. Different combinations of this prefix list are created and then submitted to the tree as new queries. Finally, the objects are ranked with respect to their distances to the submitted query. This however affects the searching time, especially if the prefix is large.

Succinct Nearest Neighbour Search (NAPP)

In [70], authors propose the *Neighbourhood Approximation* (NAPP). The main idea of the NAPP is to get the objects located in the intersection region of t reference objects shared between the query and database objects for a certain length l of the ordered list, such that $t < l$. For a given a query q, the objects similar to the query are the objects where $L(o_i, R) \cap L(q, R) \neq \phi$ and $|L(o_i, R) \cap L(q, R)| \geq t$ for ordered lists of length l.

The data structure related to this algorithm is similar to the MIF except for two features. The first difference is that only 4 bytes are required for each element in the posting list, because the positions of the references in the ordered list of the object are not sorted. The second difference is that the posting lists are compressed. Each list is compressed using the sparse bitmap representation [71].

The indexing process is as follows. After computing the ordered list for all objects, the objects are stored in the posting lists of reference points if these reference points appear in their ordered lists (Figure 2.8(a)). For each inverted list, ranges of consecutive integers are defined. These consecutive integers are replaced by the differences between them (Figure 2.8(b)). Then, the group of differences are gathered in pairs by similarity. The first element is the difference value and the second element is the number of elements that have the same difference value (Figure 2.8(c)). The indexing time of this technique is high especially due to the compressing time. The searching time and the efficiency depend on the indexing technique and the threshold values.

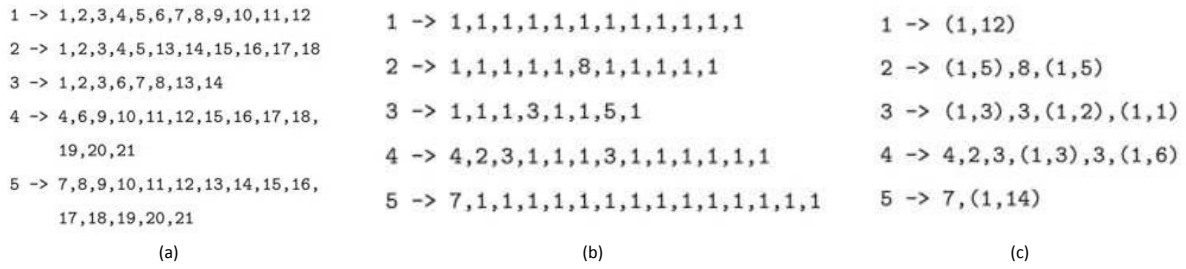


Figure 2.8: a) Inverted index b) Inverted index with differences c) Inverted index with grouped differences. The figure is copied from [70]

2.4.2 Similar techniques based On permutations

There are several algorithms that are based on permutations and the relation between the objects and a set of pivots, but with different methodologies, such as *i-Distance* [72], *M-Index* [51] and *Product Quantization* [73]. These techniques use the distance information rather than the rank only or apply some data decomposition in order to index the database objects and answer the users queries efficiently. The main difference between them and permutation-based indexing is that in permutation-based indexing, the similarity is based on the ranks. The distance information is not considered as a parameter in the representation of the objects and there is no decomposition of the original data.

i-Distance

The main idea of *i-Distance* [72] is to cluster the domain into n clusters. Each cluster C_j is then represented by a reference point r_j. Every database object is then assigned to a numeric key $iDist$ according

to its distance from its cluster's reference point. This numeric key is calculated for each object o_i as follows:

$$iDist(o_i) = d(r_j, o_i) + j \times \lambda,$$

where $o_i \in C_j$ and λ is a large constant value to separate the objects within each cluster. The database objects are then stored in a B^+-Tree [22] based on their $iDist$ keys.

Figure 2.9(a) shows the indexing process. For example o_1 and o_2 have the same $iDist$ key as they are located at the same distance from the reference point r_1.

For a k-NN query, the algorithm handles it as a range query with small radius ρ, then the search sphere expands gradually until the k-NN are found. This is performed by measuring the distance between the query q and the reference points. A cluster C_j is accessed if q is located within this cluster. Figure 2.9(b) shows the search process. More details are available in [72].

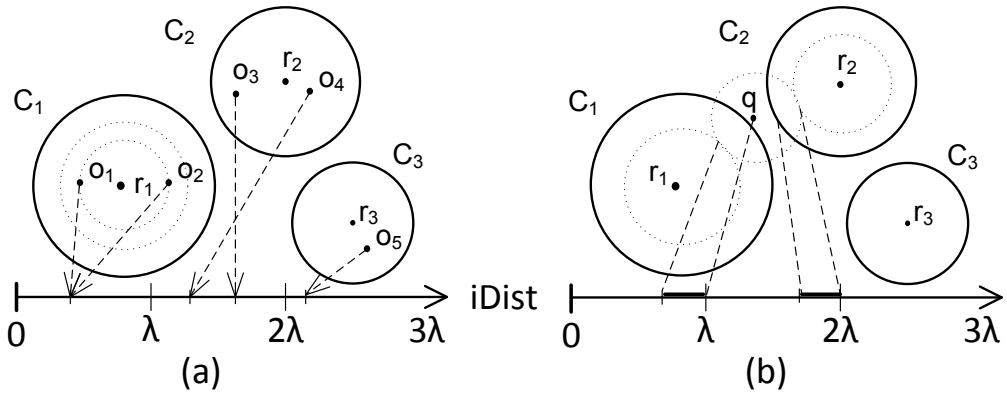


Figure 2.9: *i-Distance* principles a) Indexing b) Searching

M-Index

The *M-Index* [51] is based on the *i-Distance* principles. The data structure is composed of multi-levels. In the first level, similar to *i-Distance*, n reference points are chosen. Using these reference points, Voronoi partitioning is applied to divide the space into n regions C_j. A partition C_j contains all the objects close to its reference point r_j. Figure 2.10(a) illustrates level one partitioning. In the second level, each Voronoi cell is divided into a number of sub-regions $C_{j,k}$ based on the distance between the objects within the cell and the second nearest reference. An object $o_i \in C_j$ is assigned to a sub-cluster $C_{j,k}$ if $d(o_i, r_j) < d(o_i, r_k) < \cdots < d(o_i, r_n)$. Then, the algorithm repeats these steps l times. Accordingly, objects that see the same l nearest references will be in the same cluster. Afterwards, a key is assigned to each object o_i based on it's cluster information as follows:

$$key(o_i) = d(r_j, o_i) + \sum_{k=0}^{l-1} j \times n^{l-1-j},$$

where r_j is the closest reference point to object o_i. The objects within a cluster $C_{j,k}$ are mapped to interval $[j \times n + k, j \times n + k + 1]$. Then, similar to the *i-Distance*, the objects are stored in a B^+-Tree by their key values.

Figure 2.10(b) shows level 2 partitioning. When searching, the algorithm accesses the suitable cluster based on the distance between the query and the cluster's reference points. More details are available in [51].

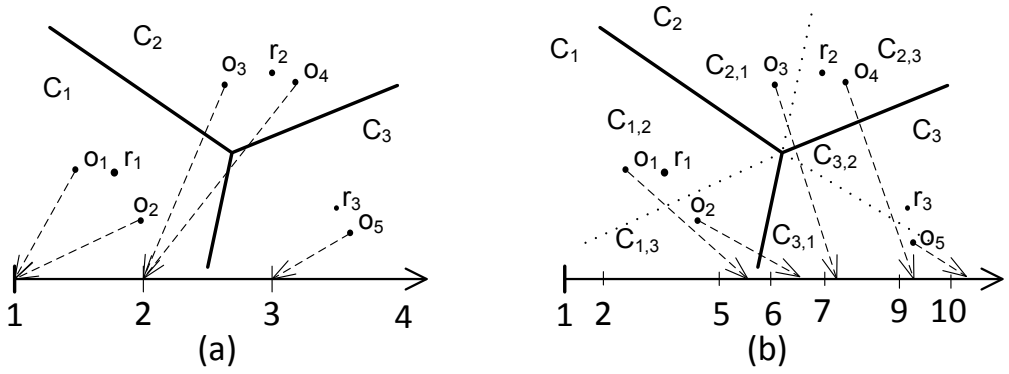


Figure 2.10: *M-Index* principles ($l = 2$, $n = 3$) a) Level 1 indexing b) Level 2 indexing.

In [74], the same authors proposed *M-Chord* to handle large scale data. *M-Chord* is a distributed implementation of the M-Index data structure based on the *Chord* [75] algorithm. *Chord* is a *P2P*[49] algorithm that allows to find efficiently the node that stores a particular data item. The main idea of *M-Chord* is to distributed the pivots between the compute nodes. Then, using the *Chord* algorithm, the location of the data is defined. More details are available in [74].

Product Quantization

Similar to *i-Distance* and *M-Index*, the *Product Quantization* [73] choose n references. Database objects are then assigned to their nearest reference point. That creates a Voronoi-like partitioning of cells C_n. For each database object o_i of dimension m, the vector is divided into l distinct sub-vectors of dimension $\bar{m} = \frac{m}{l}$. The sub-vectors are then represented by distinct reference points. Then, these coded sub-vectors are concatenated and saved to represent the object. While searching, objects that have the same code as the query are retrieved.

It is clear from the three above approaches, that they are not comparable to permutation-based in-

dexing. The main difference is that the permutation-based indexing is based only on the ranking of the reference point and no distance information is assigned to the references like in *i-Distance* [72] and *M-Index* [51]. And no vector decomposition or coding is done like in *Product Quantization* [73]. Nevertheless, they belong to the same family of reference based indexing.

2.4.3 Reference Points Selection

All the above mentioned permutation-based indexing data structures may be based on a random selection of the reference (pivots). No clear criterion to select references or a way to define the optimal number of references for efficient permutation-based indexing is provided in the original references.

The strategy to set the number and the position of the reference points (pivots) may be found in several studies [76, 77, 78, 79, 80, 81]. In general, good pivots are the pivots that are far away from each other and cover the space of objects. Pivot selection affects the efficiency of the approximate similarity search algorithms. The selection of reference points can be categorized into three approaches are *random selection*, *incremental selection* and *local optimal selection*.

In this section, we give one example for each of these three approaches, which are Random groups [79], Farthest-First Traversal (FFT) [81] and k-Medoids (kMED) [76]. Then, we show the Balancing Pivot-Position Occurrences (BPP), which is the first work [82] that aims for efficient reference points selection for permutation based indexing.

Random groups

The basic idea of random selection is to select the reference points randomly and perform the indexing using these random references. A more advanced technique is called *random groups*. It aims to increases the mean distance between the random points that were chosen. Randomly l groups of n references are selected. Then, the mean distance is calculated with respect to each group. The group that has the maximum mean distance value is selected.

Farthest-First Traversal

Farthest-First Traversal (FFT) belongs to the family of *incremental selection*. FFT [81] was proposed as a 2-approximation algorithm for the *k-center* NP-hard problem [83]. The *k-center* problem is defined as follows.

Definition 6. *Given a metric space $\mathcal{M} = (\mathcal{D}, d)$ and an integer n, find a subset $R \subseteq \mathcal{D}$ of size n while minimizing*

$$\max_{o_i \in \mathcal{D}} \min_{r_j \in R} d(o_i, r_j)$$

In other words, given N objects, find n references such that the distance to the nearest reference is the smallest. Algorithm 2.2 shows how the FFT algorithm proceeds in a greedy manner following the heuristic of spreading at maximum the reference points. It selects a random reference point r_j from the data domain and puts it in the output set (lines 1-2). Then, the algorithm selects another reference point which is the farthest from the already selected reference (line 3). The algorithm continues until the number of references is achieved. The complexity of the algorithm is $O(nN)$.

Algorithm 2.2 Farthest-First Traversal Algorithm

IN: $\mathcal{D}, n$,
OUT: R
1. Get a random reference point $r_j \in \mathcal{D}$
2. Set $R = \{r_i\}$
3. while $|R| < n$
4. $\quad j = \underset{o_i \in \mathcal{D}}{\operatorname{argmax}}\, d(o_i, R)$
5. $\quad R = R \cup \{r_j\}$

k-Medoids

k-Medoids belongs to the family of *local optimal selection*. k-Medoids [76] is similar to the k-Means [84] algorithm in selecting density-based representative. k-Medoids tries to minimize the average distance between the database objects. The main difference between k-Means and k-Medoids that the k-Medoids uses one of the objects as a representative of the center of the cluster rather than computing the centroid for each group. k-Medoids was proposed as a way for selecting pivots in [79, 85, 86, 31, 87, 88, 89].

Balancing Pivot-Position Occurrences

Balancing Pivot-Position Occurrences (BPP) [82] is a way to organize the already selected pivots in the ordered lists. The main idea is that, if a certain reference is located at the same position in different ordered lists, this reference points encodes little discriminative information.

The first $\hat{n}$ references are selected randomly from the dataset $\mathcal{D}$, where $\hat{n} > n$. The ordered lists $L(o_i, R)$ for objects $o_i \in \mathcal{D}$ are calculated. At each iteration, the algorithm evaluates the effect on ordered lists of removing a reference point r_j from R. A reference point r_j is removed if it appears many times at the same position in different ordered lists. The algorithm continues until $|R| = n$. The complexity of the algorithm is $O(\hat{n}|S|)$, where $S \subset R$.

As mentioned before, the algorithm does not propose a way to place the reference points in the dataset, it proposes a way to decimate a randomly selected set based on efficiency, with a criterion close to the IDF principle. The computational time of the algorithm is very high especially for large datasets and large set of references.

2.5 Discussion

In [61, 90, 70, 66], permutation-based indexing empirically proved to be an efficient technique for searching in multidimensional spaces. However, some important research questions are still not answered to ascertain its efficiency.

1. What is the encoding model behind permutation-based indexing ?
2. What is the optimal way to select the reference points ?
3. What is the efficient way to store the permutations ?

In the next chapters, we propose some answers to these questions.

2.6 Our Evaluation Strategy

In all of our experiments, we will measure and compare our proposed techniques and algorithms in terms of performance (memory usage, indexing time and searching time) and relevance (Recall and Position Error [15]).

2.6.1 Performance

The performance measure is based on the type of the algorithm; sequential or multi-core or distributed. The sequential time and multi-core indexing and searching time is measured from the beginning of the execution until the end of the execution. For memory usage, we measure the amount of memory at the end of the indexing process. This is the memory that is allocated and used to answer the queries.

For the distributed time, processes index and search in parallel through their data portions. Accordingly, the reported time is the maximum indexing or searching time that is reported at a single process during the execution. That leads to two other terms which are the *speedup* and the *efficiency*.

Speedup

The speedup refers to how much the parallel implementation is faster than the sequential implementation:

$$S_P = \frac{Sequential\ time}{Parallel\ time} \tag{2.3}$$

The optimal speedup is the linear speedup, where it equals to the number of cores that participate in the parallelization process.

Efficiency

The efficiency E estimates how well the processors are utilized in the parallelization process:

$$E = \frac{S_P}{P} \tag{2.4}$$

The optimal efficiency is equal to 1.

2.6.2 Effectiveness

Given a query q, the recall (RE) and position error (PE) are defined as [15]:

$$RE = \frac{|M \bigcap M_A|}{|M|}, \tag{2.5}$$

$$PE = \frac{\sum\limits_{o \in M_A} |P(o, M) - P(M_A, o)|}{|M_A|.N}, \tag{2.6}$$

where M and M_A are the ordering of the k-top ranked objects from q for exact similarity search and approximate similarity search respectively. $P(o, M)$ indicates the position of o in M and $P(o, M_A)$ indicates the position of o in M_A. A RE$= 0.3$ indicates that the output results of the searching algorithm contain 30% of the k closest objects retrieved by the exact search. A PE$= 0.001$ indicates that the average shifting of the best k objects ranked by the approximate search algorithm, with respect to the rank obtained by the exact search, is 0.1% of the dataset. An efficient algorithm should return a $RE = 1$ and $PE = 0$. Unless indicated otherwise, the average RE and PE are measured in all the experiments based on at least 100 different queries q that are selected randomly from the dataset.

2.6.3 Datasets

Our datasets consists of vectors in the m-dimensional space. We use large scale public standard datasets:

- DS_1: Color histograms (32-dimensional vector) from an image database of 50,000 images. The dataset is available on the UCI knowledge Discovery Archive [1].
- DS_2: Color histogram (112-dimensional vector) extracted from an image database of 112,544 images. The dataset is available on the SISAP website [14].
- DS_3: About 9-millions color features (84-dimensional) extracted from the 12-million ImageNet corpus [91][2].

[1] http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html
[2] The feature dataset comprises 9'175'426 elements of 84 dimensions

- DS_4: About 9-million shape features (21-dimensional) extracted from the 12-million ImageNet corpus [91][3].

- DS_5: The full CoPhIR dataset [92], consists of 106-million five MPEG-7 features (Scalable Colour, Colour Structure, Colour Layout, Edge Histogram and Homogeneous Texture). These five features are combined to build 280-dimensional vectors. The dataset is saved in XML files, where the size of these XML files is more than 240 Giga bytes. After extracting the features from the XML, the database size is about 75 Giga bytes.

- DS_6: We also created three datasets from the CoPhIR dataset of different sizes from the first three descriptors to construct vectors of size 208.

  - 1-million (478 MB)
  - 10-million (4.7 GB)
  - 106-million (50 GB)

- DS_7: The BIGANN dataset consists of 1 billion SIFT features [93]. The dimension of SIFT features is 128. The features are extracted from 1 million images.

2.6.4 Queries

A query q is a vector of in the m-dimensional space where the database objects live. The distance function between a query q and the database objects is defined using the classical euclidean L_2 distance. From all the datasets that were mentioned in the previous sections, a set of 100 queries were selected randomly. A ground truth is calculated between these queries and the database objects using exhaustive search.

2.7 Summary

Similarity search is known to be achievable through at least three ways. The first technique is called *exhaustive search*. For a given query, the distance between the query and the database objects is calculated while keeping track of which objects similar (near) to the query. The main problem with this technique, that it does not scale with large collections.

The second technique is the *exact search*. It is based on space-partitioning data structures for organizing points in m-dimensional space. With this space partitioning techniques, the number of objects which are compared to the query is reduced, which improves the searching time. This type of techniques

[3]The feature dataset comprises 9'175'426 elements of 21 dimensions

is not efficient for high dimensional datasets, due to the curse of dimensionality. As the dimension increases a large part of the database needs to be scanned. Hence, the performance becomes similar to the exhaustive search.

The third technique is *approximate search*. It aims to reduce the amount of data that needs to be accessed. There are three common approaches to do that. The first approach is the *space transformation* techniques. Space transformation techniques change the metric space into another, where searching in the new space is much simpler. The second approach is the *space partitioning* techniques. The partition is generally done using pivots. It is similar to the exhaustive search technique, but with some omission and decomposition techniques. It reduces the amount of data that need to be accessed. The third approach is to pre-compute the distances between the objects. The *pivoting* approach is the most common.

Recently, a new technique which is based on permutations is proposed, which is called *permutation-based indexing*. The permutation-based indexing belongs to the approximate similarity search family. It is a hybrid model between space transformation and space partitioning techniques. Permutation-based indexing is based on selecting references (pivots). Permutation-based indexing aims to predict the similarity between the database objects according to how they order their distances towards this set of distinguished anchor objects.

Several data structures [61, 90, 70, 66] were proposed to index the permutations in an efficient way, but still some open issues affect the quality and efficiency of the searching algorithm. The main questions are how many references should be chosen and how can they be stored efficiently. We study these questions in the next chapters.

Chapter 3

Permutation-Based Indexing Encoding Model

As a recent technique for approximate similarity search, permutation-based indexing aims to predict the similarity between the database objects according to how they order their distances towards a set of distinguished anchor (reference) objects. There are several data structures that was proposed to handle these objects and these reference points. However, the justification model for its encoding and the selection of reference points are not well studied. In this chapter, we aim to give a sound formal model for permutation-based indexing with a methodology to select reference points.

In section 3.1 and 3.2, we propose a model that describe formally how the permutation-based indexing encode objects position. In section 3.3, we study the encoding with respect to the nearest reference objects. In section 3.4, we propose different techniques for selecting the reference points for efficient searching. We then evaluate our proposed selection strategies on real and synthetic datasets in section 3.5.

3.1 Encoding Model

Given $\mathcal{D} = \{o_1, \ldots, o_N\}$ a collection of N m-dimensional objects $o_i \in \mathcal{R}^m$ and a distance function $d(.,.)$ operating on objects and following the metric space postulates, we choose a set of n $(0 < n \leq N)$ reference objects $R = \{r_1, \ldots, r_n\} \subset \mathcal{D}$ where, for every j, $r_j = o_i$ for some i.

Definition 7. *(Ordered list) Given $o_i \in \mathcal{D}$, we define the ordered list of object o_i as the mapping*

$$L(o_i, R) : \{r_1, \ldots, r_n\} \rightarrow \{r_{j_1}, \ldots, r_{j_n}\}$$

Such that $\forall k < n$*:*

$$\begin{cases} \quad d(o_i, r_{j_k}) < d(o_i, r_{j_{k+1}}) \\ or \quad d(o_i, r_{j_k}) = d(o_i, r_{j_{k+1}}) \;\; j_k < j_{k+1} \end{cases}$$

In other words, $L(o_i, R)$ is the list of reference objects sorted in increasing distance values from o_i. In case of a tie on distances, the reference objects of lower index appears first in the list. Figure 3.1(b) gives the ordered lists $L(o_i, R)$ for $n = 4$ for D and R illustrated in Figure 3.1(a).

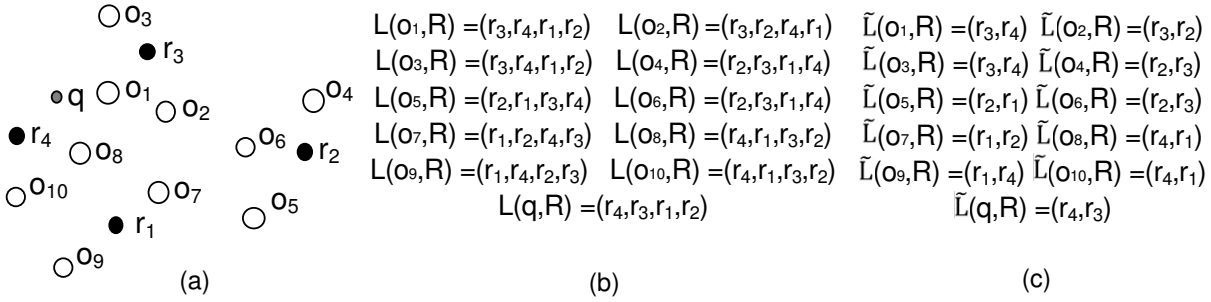


Figure 3.1: Data configuration (a) ∘ data objects o_i; • reference objects r_j; the gray circle is the query object q (b) Ordered lists for all the objects $L(o_i, R)$ using full permutations (c) Pruned ordered lists for all the objects $\tilde{L}(o_i, R)$ with $\tilde{n} = 2$ (see section 3.3)

We define $L(o_i, R)_{|r_j} = k$, as the position k of a reference point r_j in the ordered list of the object o_i. Thus:

- $L(o_i, R)_{|r_j}$ is the position of reference object r_j in the ordered list of o_i;
- if $L(o_i, R)_{|r} = k$, then $r_{j_k} = r$ and r is the k^{th} neighbour of o_i in R.

Based on this encoding, we can define a new distance approximation using any distance function that can be computed between ranks (ordered lists). Due to the complexity of the *Spearman Rho* compared to the *Spearman Footrule Distance* and the minor difference in performance between them (as discussed in section 2.4)[56], we use the *Spearman Footrule Distance* (d_{SFD}) based on set R:

$$d_{\mathrm{SFD}}(q, o_i) = \sum_{j=1}^{n} |L(q, R)_{|r_j} - L(o_i, R)_{|r_j}| \tag{3.1}$$

It has been shown that this distance can be used to resolve the k-NN similarity queries since d_{SFD} approximates, in terms of ranking, continuous distance functions for the search of k-NN. In other words,

$$d(q, o_i) \overset{\mathrm{rank}}{\simeq} d_{\mathrm{SFD}}(q, o_i) \tag{3.2}$$

Hence, *Permutation-Based Indexing* aims at facilitating and optimising, for any query q the computation of $d_{\mathrm{SFD}}(q, o_i)$ for all $o_i \in \mathcal{D}$.

3.2 Algebraic Foundations

3.2.1 Topology

Computing distances over ordered lists generates distance approximations. From this distance, we can infer an equivalence relationship '$\equiv$' defined as follows.

Definition 8. *(Equivalence Relationship) Given $R \subset \mathcal{D}$ and $q, o_i \in \mathcal{D}$, we note $q \equiv o_i$ if and only if*

- $d_{\mathrm{SFD}}(q, o_i) = 0$
- *equivalently,* $L(q, R) = L(o_i, R)$
- *equivalently,* $L(q, R)_{|r_j} = L(o_i, R)_{|r_j} \quad \forall j = 1 \dots n$

Definition 9. *(Equivalence class - Invariance) The equivalence class of object o_i is*

$$[o_i] = \{p \in \mathcal{R}^m \text{ such that } p \equiv o_i\}$$

The quotient space $\mathcal{D}/\equiv$ is the set of all equivalence classes of the d_{SFD} from $\mathcal{D}$.

The equivalence classes show the extent of the invariance of the $L(o_i, R)$ encoding. The equivalence class is the set of all positions p an object can take without changing its encoding. Similarly, the equivalence classes measure the approximation made by the distance d_{SFD}. The value of the d_{SFD} distance between any pair point of respective classes does not vary.

We will use these properties in section 3.4 to propose strategies to optimise the choice of the set R. Prior to that, we construct a geometric structure for analysing permutation-based indexing.

3.2.2 Geometry

Objects o_i are points of $\mathcal{R}^m$ over which some geometrical properties can be inferred. A m-dimensional space may be partitioned by $(m-1)$-dimensional hyperplanes. In our context, perpendicular bisectors are particular such hyperplanes.

Definition 10. *(Perpendicular bisector) Given $r_k, r_l \in R$, we define δ_{kl} as the $(m-1)$-dimensional perpendicular bisector of the segment $[r_k, r_l]$.*

Proposition 1. *Given two objects $o_i, o_j \in \mathcal{D}$, then*

$$(L(o_i, R)_{|r_k} - L(o_i, R)_{|r_l}).((L(o_j, R)_{|r_k} - L(o_j, R)_{|r_l})) < 0$$

if and only if o_i and o_j are separated by δ_{kl}.

Proof. Traversing δ_{kl} flips the ranking of r_k and r_l. □

There is therefore a direct relationship between the geometrical organisation of the points and the organisation of the ordered list. More generally, neighbouring relationships between objects relate to Voronoi diagrams, which are formed by bisectors δ_{kl}.

Definition 11. *(Voronoi cell) Given $r_k \in R$, we define $V_R(r_k) \subset \mathcal{R}^m$ as the Voronoi cell of r_k with respect to R. $V_R(r_k)$ is the m-dimensional subset:*

$$V_R(r_k) = \{p \in \mathcal{R}^m \text{ such that } d(p, r_k) \leq (p, r_l) \ \ \forall r_l \in R\}$$

$V_R(r_k)$ is a m-dimensional simplex bounded by bisectors δ_{kl}. r_k is then said to be a generator of $V_R(r_k)$. Aggregating all Voronoi cells, defines $\mathcal{V}_R$ Voronoi diagram of set R.

$$\mathcal{V}_R = \{V_R(r_k) \quad \forall r_k \in R\}$$

Definition 12. *(Delaunay Graph) Given R and $\mathcal{V}(R)$, we define $\mathcal{G} = (R, E)$ the Delaunay Graph with vertices $r_k \in R$ and edges E such that:*

$$(r_k, r_l) \in E \ \text{ if and only if } \ V_R(r_k) \ \text{ and } \ V_R(r_l) \quad \text{share a common facet}$$

.

Definition 11, takes a unique object r_k to generate each Voronoi cell. Hence, for all objects $o_i \in V_R(r_k)$, we have $L(o_i, R)_{|r_k} = 1$.

Consider $r_l, r_m \in R$, neighbours of r_k in G, where δ_{kl} and δ_{km} support facets of $V_R(r_k)$. Assume objects $o_i, o_j \in \mathcal{D}$ are on each side of the facet between r_l and r_m. If δ_{lm} is extended within $V_R(r_k)$, it will separate objects o_i for which $L(o_i, R)_{r_l} > L(o_i, R)_{r_k}$ from object o_j for which $L(o_j, R)_{r_l} < L(o_j, R)_{r_k}$. In particular, δ_{lm} isolate a portion of $V_R(r_k)$, where for each object o_i in that region is such that $L(o_i, R)_{|r_k} = 1$, $L(o_i, R)_{|r_l} = 2$ and $L(o_i, R)_{|r_m} = 3$.

Repeating the process, leads to the construction of the ordered 2-order Voronoi diagram, where the generators of the cells at the ordered pairs of reference objects. Figures 3.2(a) and 3.2(b) shows an example of the ordinary Voronoi diagram and the ordered 2-Order Voronoi diagram. Generalising this construction, we obtain the *Ordered k-Order Voronoi Diagram (OOVD)* [94].

Definition 13. *(Ordered k-Order Voronoi Diagram) Given $R_l = (r_{j_1}, \ldots, r_{j_l})$ an ordered subset of R, we define $V_R(R_l) \subset \mathcal{R}^m$ as the OOVD cell of R_l with respect to R. $V_R(R_k)$ is the m-dimensional subset:*

$$o_i \in V_R(R_l) \Leftrightarrow L(o_i, R)_{|(r_{j1}, \ldots, r_{jl})} = (1, \ldots, l)$$

The equivalent classes of the d_{SFD} distance ($\mathcal{D}/\equiv$) are therefore the cells of the ordered n-order Voronoi Diagram of R as shown in figure 3.3.

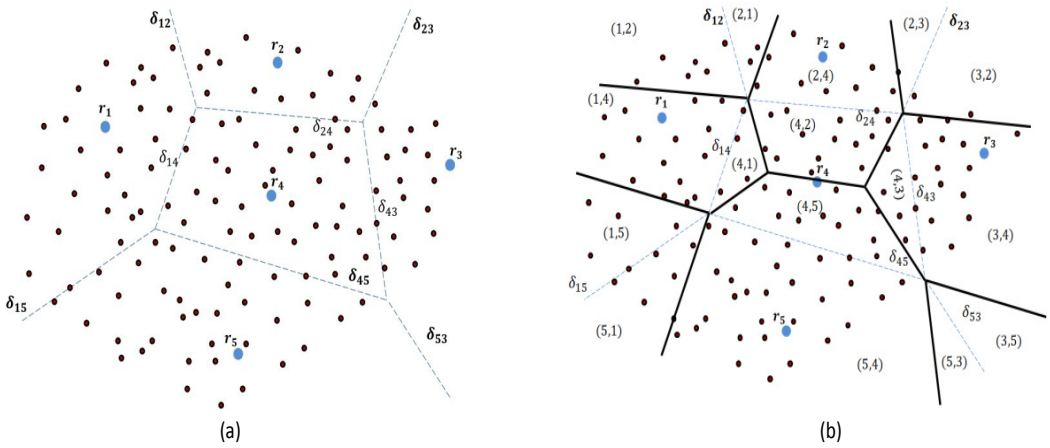


Figure 3.2: (a) The ordinary Voronoi diagram. (b) The ordered 2-Order Voronoi Diagram (the solid lines) and the ordinary Voronoi diagram (the dotted lines).

The above definitions show that the permutation-based indexing encoding model can be related to the construction of an Ordered k-Order Voronoi Diagram [94]. The order of the diagram is defined by the number of reference points.

3.3 Pruning Permutation Lists

It is intuitively clear that objects may be encoded relatively to a reduced set of reference objects only. In the above geometrical model using OOVD, this corresponds to saying that a OOVD cell is essentially defined by bisectors between close points. Using a locality criterion, a point can be encoded by local set of reference objects only. A point can then be represented by its *pruned ordered list*. By definition (Def. 7), the first reference points are the local (closest) points to the object.

In other words, such a locality criterion may be translated on ranking $L(o_i, R)_{|r_j}$ by simply pruning lists $L(o_i, R)$ to a length $\tilde{n}$ into $\tilde{L}(o_i, R)$:

$$\tilde{L}(o_i, R) = (r_j / L(o_i, R)_{|r_j} \leq \tilde{n})$$

This corresponds to considering only the $\tilde{n}$ closest reference objects to encode o_i. However, the set of reference objects R will still typically be chosen to cover the complete object space (see section 3.4).

Figure 3.1(c) gives the pruned ordered lists $\tilde{L}(o_i, R)$ for $\tilde{n} = 2$ and D and R illustrated in Figure 3.1(a). For example in Figure 3.1(a), reference points r_3 and r_4 are the closest reference points to object o_1. As a result, they appear in the pruned ordered list of object o_1 (Figure 3.1(c)).

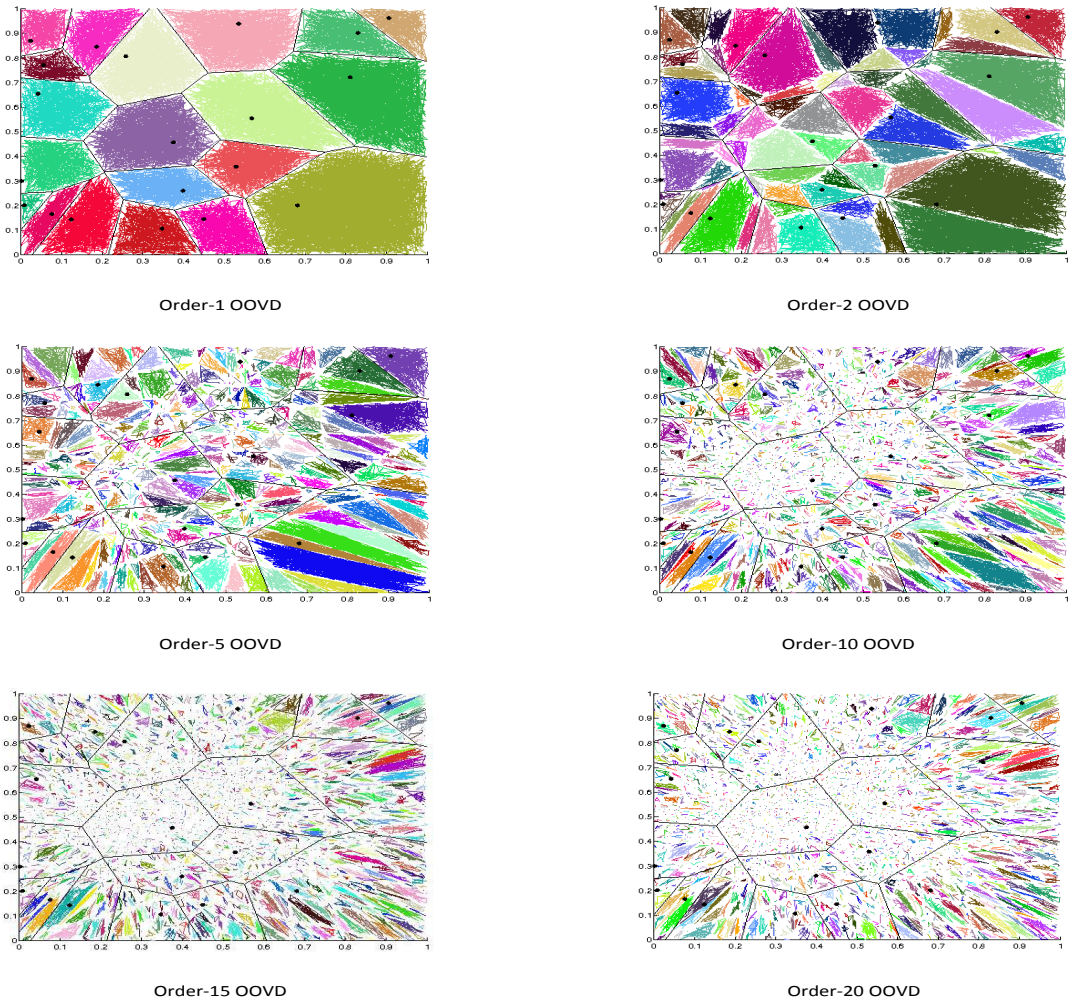


Figure 3.3: Ordered k-Order Voronoi Diagram: The points within the same OOVD cell that have the same color have the same ordered list.

3.3.1 Measuring distances using pruned ordered lists

Given two objects o_i and o_j and considering only the nearest $\tilde{n}$ reference objects, the set R is divided into three subsets (Figure 3.4):

- $\tilde{R}_i$ is the set of nearest $\tilde{n}$ reference objects to o_i;
- $\tilde{R}_j$ is the set of nearest $\tilde{n}$ reference objects to o_j;
- $R \setminus (\tilde{R}_i \cup \tilde{R}_i)$ is the rest of reference objects that do not belong to $\tilde{R}_i$ nor $\tilde{R}_j$.

If a given $r_k \in \tilde{R}_i$ and $r_k \notin \tilde{R}_j$, then r_k is farther than the $\tilde{n}^{th}$ neighbour of o_j. $\tilde{L}(o_i, R)_{|r_k}$ is therefore undefined since the distance between r_k and o_j is seen as "infinity", which can be represented as a large constant value $\mathcal{K}$.

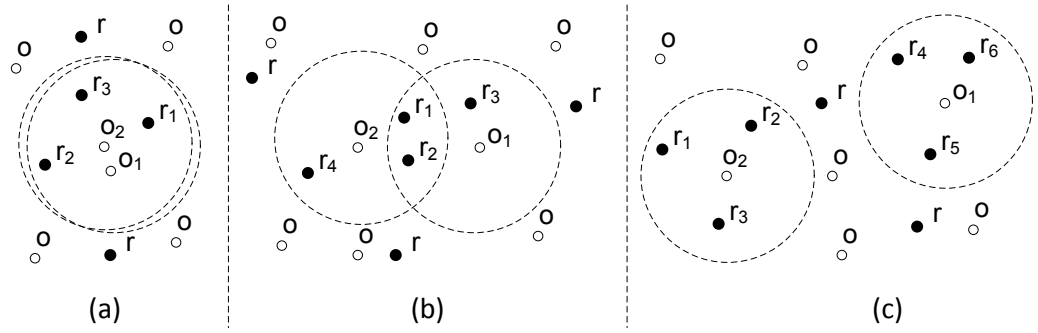


Figure 3.4: (a) Objects share the same reference list (b) Objects share part of the reference list (c) Objects do not share references.

In that case, the d_{SFD} (Eq. 3.1) further translates into:

$$
\begin{aligned}
d_{\mathrm{SFD}}(o_i, o_j) = \sum_{\substack{r_k \in \tilde{R}_i \\ r_k \in \tilde{R}_j}} |\tilde{L}(o_i, R)_{|r_k} - \tilde{L}(o_j, R)_{|r_k}| + \\
\sum_{\substack{r_k \in \tilde{R}_i \\ r_k \notin \tilde{R}_j}} |\tilde{L}(o_i, R)_{|r_k} - \mathcal{K}| + \\
\sum_{\substack{r_k \notin \tilde{R}_i \\ r_k \in \tilde{R}_j}} |\tilde{L}(o_j, R)_{|r_k} - \mathcal{K}| + \\
\sum_{\substack{r_k \notin \tilde{R}_i \\ r_k \notin \tilde{R}_j}} |\mathcal{K} - \mathcal{K}|
\end{aligned}
\tag{3.3}
$$

For a given object o_i, nearest reference objects from o_i will suffice to discriminate it from the surrounding objects, whereas the ordering of reference points far from o_i is seen as carrying little (or less) discriminative information.

3.3.2 Choosing a value for constant $\mathcal{K}$

We look for a constant $\mathcal{K}$ that leads to an overall consistent behaviour of the distance function on pruned lists.

Definition 14. *Given $\tilde{R}_i$ and $\tilde{R}_j$ as the closest reference sets for object o_i and o_j respectively. We define z as the number of shared reference points between the two subsets:*

$$z = |\tilde{R}_i \cap \tilde{R}_j|$$

where $0 \leq z \leq \tilde{n}$.

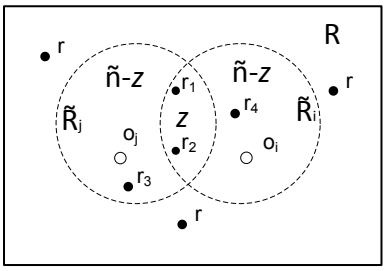


Figure 3.5: z-overlapping for objects o_i and o_j, $z = 2$.

If o_i and o_j are close to each other, their pruned reference sets $\tilde{R}_i = \tilde{R}_j$ share $z = \tilde{n}$ reference points (Figure 3.5). If the two objects are far from each other, $\tilde{R}_i \cap \tilde{R}_j = \phi$ and $z = 0$.

Non-overlapping case

If $z = 0$, equation 3.3 becomes:

$$d_{\mathrm{SFD}}(o_i, o_j) = \sum_{\substack{r_k \in \tilde{R}_i \\ r_k \notin \tilde{R}_j}} (\mathcal{K} - \tilde{L}(o_i, R)_{|r_k}) + \sum_{\substack{r_k \notin \tilde{R}_i \\ r_k \in \tilde{R}_j}} (\mathcal{K} - \tilde{L}(o_j, R)_{|r_k}) = 2 \sum_{i=1}^{\tilde{n}} (\mathcal{K} - i)$$

We then obtain:

$$d_{\mathrm{SFD}}(o_i, o_j) = \tilde{n}(2\mathcal{K} - \tilde{n} - 1) \tag{3.4}$$

which is both a upper bound and a lower bound for the non-overlapping case.

z-overlapping case

We consider that if $r_k \in \tilde{R}_i \cap \tilde{R}_j$ then

$$0 \leq |\tilde{L}(o_i, R)_{|r_k} - \tilde{L}(o_j, R)_{|r_k}| \leq \tilde{n} - 1$$

Therefore, equation 3.3 becomes:

$$d_{\mathrm{SFD}}(o_i, o_j) \leq z(\tilde{n} - 1) + 2 \sum_{i=1}^{\tilde{n}-z} (\mathcal{K} - i) \qquad \text{upper bound}$$

and

$$d_{\mathrm{SFD}}(o_i, o_j) \geq z \times 0 + 2 \sum_{i=z+1}^{\tilde{n}} (\mathcal{K} - i) \qquad \text{lower bound}$$

Deriving these equations, we then obtain:

$$d_{\mathrm{SFD}}(o_i, o_j) \leq z(\tilde{n} - 1) + (\tilde{n} - z)(2\mathcal{K} - \tilde{n} + z - 1) \qquad \text{upper bound} \tag{3.5}$$

and

$$d_{\mathrm{SFD}}(o_i, o_j) \geq (\tilde{n} - z)(2\mathcal{K} - \tilde{n} + z - 1) \qquad \text{lower bound} \tag{3.6}$$

Full Overlapping case

In case of full overlapping $z = \tilde{n}$ equation 3.3 would be:

$$d_{\mathrm{SFD}}(o_i, o_j) = \sum_{\substack{r_k \in \tilde{R}_i \\ r_k \in \tilde{R}_j}} |\tilde{L}(o_i, R)_{|r_k} - \tilde{L}(o_j, R)_{|r_k}|$$

A rough approximation of this equation is:

$$d_{\mathrm{SFD}}(o_i, o_j) = \tilde{n}(\tilde{n} - 1), \tag{3.7}$$

since $0 < \tilde{L}(o_i, R)_{|r_j} - \tilde{L}(o_j, R)_{|r_j} \leq (\tilde{n} - 1)$

Hence, the upper bound of $d_{\mathrm{SFD}}(o_i, o_j)$ when $\tilde{R}_i = \tilde{R}_j$ is $\tilde{n}(\tilde{n} - 1)$, when o_i and o_j share the same pruned ordered list in different order. The lower bound for this case is 0, when o_i and o_j share the same pruned ordered list in the same order.

Value of the constant $\mathcal{K}$

At $z = 0$ lower and upper bound coincide at $d_{\mathrm{SFD}}(o_i, o_j) = \tilde{n}(2\mathcal{K} - \tilde{n} - 1)$. At $z = \tilde{n}$ the lower bound coincide at 0 and the upper bound coincide at $\tilde{n}(\tilde{n} - 1)$. For a consistent behaviour of $d_{\mathrm{SFD}}(o_i, o_j)$ we want:

$$\tilde{n}(\tilde{n} - 1) < \tilde{n}(2\mathcal{K} - \tilde{n} - 1)$$

This imposes $\mathcal{K} > \tilde{n}$. Hence, the first admissible value for $\mathcal{K}$ is $\mathcal{K} = \tilde{n} + 1$. Throughout the thesis, we use this value.

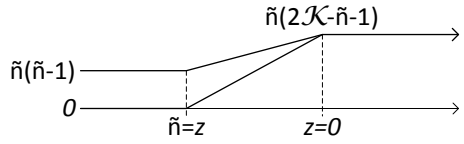


Figure 3.6: Evolution of the distance on pruned ordered list $\mathcal{K}$

3.4 Reference Points Selection

In section 3.2.1, we have shown that the approximation made by the d_{SFD} and the invariance of the encoding $o_i \rightarrow L(o_i, R)$ is related to the size of the equivalence classes of the relation $\equiv$ (Def. 8). In other words, each point will be uniquely defined if each equivalence class contains the point o_i only.

By Definition 13, this is equivalent to say that each $k-$OOVD cell contains one point only. We know by construction that the OOVD is a re-partition if the classical Voronoi diagram (order-1 OOVD). Hence, the above criterion can be approximated by choosing R so as to minimize the size of the Voronoi cells in $\mathcal{V}_R$. The algorithms below implement this criteria with two models for the "size of $V_R(r_j)$".

In section 3.4.1, we use the radius of the covering circle of $V_R(r_j)$ to measure the size of the cell leading to our *global locality* approach. In section 3.4.2, we count the number of points in each cell to better adapt to the (potentially non uniform) distribution of $\mathcal{D}$, leading to our *dense locality* approach.

3.4.1 Global Locality Approach

We look for a set $R \subset \mathcal{D}$ which minimizes the number of points $o_i \in \mathcal{D}$ in each cell of $n-$OOVD of R. We will use a greedy selection approach to build R. Since it would be practically infeasible to build the OOVD at each step of the construction of R, we upper bound the above criteria by minimizing the size of each cell of the Voronoi diagram of R. Hence, the size is approximated by the maximum size of the covering circle of $V_R(r_j)$, which is itself approximated at location of objects o_i since these are the only locations where measurements can be made.

That is, given $R^{(t)}$ as the selection of reference points at iteration t, we define for each $r_j \in R^{(t)}$

$$\rho_j^{(t)} = \max_{o_i \in V_{R^{(t)}}(r_j)} d(o_i, r_j)$$

and our global criteria:

$$\rho_{max}^{(t)} = \max_j \rho_j^{(t)} \tag{3.8}$$

Voronoi diagrams have the local property that, when adding or removing a seed, only cells of neigh-

bours in the Delaunay Graph are affected [1]. We generalise this fact for splitting the largest cell in the diagram:

$$V_{R^{(t)}}(r_j*) \text{ such that } j^* = \operatorname{argmax} \rho_j^{(t)}$$

we split $V_{R^{(t)}}(r_j*)$ by setting $R^{(t+1)} = R^{(t)} \cup \{o_i*\}$, where

$$o_i* = \underset{o_i \in V_{R^{(t)}}(r_{j*})}{\operatorname{argmax}} d(o_i, r_j*)$$

Since we use o_i* as the farthest point from r_j*, o_i* is likely to be close to the border of $V_R(r_{j*})$ and efficiently split this cell. Figure 3.7 shows splitting a cell by the farthest reference point [2].

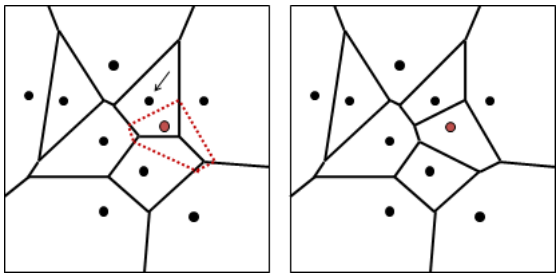


Figure 3.7: Splitting a Voronoi cell.

The above therefore provides a sound justification for a procedure close to the Farthest First strategy approach [79, 95], which we implement in the following algorithm.

Algorithm 3.1 shows the *Global Locality Approach*. In lines 1-4, an initial object is chosen. This object is considered as the first reference point r_1 . The distance between r_1 and the rest of objects is calculated and saved in dis array. All objects are then labelled with the id 1 (array $label$). In line 6-9, the object o_{i*} located at the farthest distance is considered as the next reference point r_j. In lines 10-14, the distance between the new reference point r_j and the rest of database object is calculated. The distance and the label values for object o_i are updated if $d(r_{j+1}, o_i) < d(r_j, o_i)$. The algorithm continues until it meets the *'Stopping Criterion'*, which we discuss in section 3.4.3, along with the complexity.

3.4.2 Dense Locality Approach

The above approach accounts for the geometrical size of each OOVD cell, irrespective of the geometrical sampling made by the objects themselves. Hence, it assumes a uniform distribution of the objects. Since objects arise from the representation of real items (eg. image features) potentially arranged by classes, it is likely that objects o_i will rather come in clusters.

[1] This is the base for the incremental method for the construction of the Voronoi diagram.

[2] The figure is copied from `http://www.csie.ntnu.edu.tw/~u91029/`

Algorithm 3.1 Global Locality

IN: All the database objects o_i, $R = \phi$
OUT: List of references R
1. Select an initial reference point r_1, $R \leftarrow r_1$.
2. $dist[i] = d(o_i, r_1),\ \forall i < N$.
3. $label[i] = 1\ ,\ \forall i < N$
4. $j = 1$
5. while(*Stopping Criterion*)
6. $j++$
7. $i*$= max_distance_id($dist$)
8. $r_j = o_{i*}$
9. $R \leftarrow R \cup r_j$
10. For $i = 1$ to N:
11. newDist= d(r_j,o_i)
12. If newDist $\leq dist[i]$:
13. $label[i] = j$
14. $dist[i]$ = newDist

Hence, following the idea of adaptive partitions (such as Quad Tree), we measure the size of each cell by the number of objects it contains. In other words, we define η_j as the number of points in cell $V_{R^{(t)}}(r_j)$, and our global criterion

$$\eta_{max}^{(t)} = \max \eta_j^{(t)} \tag{3.9}$$

We use the same strategy as above when selecting as new reference point the farthest point from $V_{R^{(t)}}(r_{j*})$ where

$$j^* = \underset{j}{\operatorname{argmax}}\, \eta_j^{(t)}$$

Algorithm 3.2 shows the implementation of our procedure.

Similar to the previous approach, lines 1-4 select an initial reference point and the distance between this reference point and the rest of database objects is calculated. Then, all the objects are mapped to this reference point. At each iteration, the algorithm counts how many objects see the same reference points (lines 6-9). This set of objects are considered in the same Voronoi cell. Hence, to decrease the density of this Voronoi cell, the next reference object is chosen from it (line 10). As the farthest reference point from the center of the cell.

3.4.3 Practical Setup

The above two strategies differ from how we measure the size of the Voronoi cells and how we choose the cell to split at each iteration. A more general view of algorithm 3.1 and 3.2 is as follows:

Algorithm 3.2 Dense Locality

IN: All the database objects o_i, $R = \phi$
OUT: List of references R
1. Select a initial point r_1, $R \leftarrow r_1$.
2. $dist[i] = d(o_i, r_1),\ \forall i < N$.
3. $label[i] = 1\ ,\ \forall i < N$
4. $j = 1$
5. while(*Stopping Criterion*)
6. $j++$
7. $list$= get_dense_Voronoi_cell($label$)
8. $i*$= max_distance_id($dist$,$list$)
9. $r_j = o_{i*}$
10. $R \leftarrow R \cup \{r_j\}$
11. For $o_i \in list$:
12. newDist= d(r_j,o_i)
13. If newDist $\leq dist[i]$:
14. $label[i] = j$
15. $dist[i] =$ newDist

1. Choice of the initial reference point. The choice of the initial point can for example be selected randomly or as the center of gravity from the dataset.

2. Until Stopping Criterion

   (a) Estimate the representativity of $R^{(t)}$. The representativity of R is the size of the cells, which we keep uniform using ρ_{max} (Eq.3.8) or η_{max} (Eq.3.9).

   (b) Choose a new reference point r_{j^*} and

   $$R^{(t+1)} = R^{(t)} \cup \{r_{j*}^{(t)}\}$$

   Here we propose to chose r_{j^*} as the farthest point in the cell of r_j. Other choice to better split the cell exist. For example, r_{j^*} may be replaced by the pairs of objects of largest distances in $V_R(r_{j^*})$.

   (c) Updating pointers $o_i \rightarrow r_j (o_i \in V_{R^{(t)}}(r_j))$.

The complexity of the algorithms are $O(N^2)$ and $O(N \times \eta_{max})$ for the global and dense approach respectively. Instead of visiting points linearly, the performance step (c) may be optimized by exploring the Delaunay Graph and re-allocate pointers at neighbouring cells only. We would then converge to the complexity of the incremental method for building Voronoi diagrams $O(N)$ [94].

Stopping Criterion

Our procedure may run until $R^{(t)} = \mathcal{D}$. However, our updated procedure is based on labelling the objects with respect to their nearest reference point. That allows to insert various stopping criteria:

- $|R^{(t)}| = n$. It may be that, due to memory or time limitations $|R|$ is limited to a given n. In that case, we will show that, given n, our procedures produce an optimal arrangement of these n reference points.
- $\rho_{max}^{(t)} = \rho$. In the settings of our problem, the granularity of measurements may have some lower bound (eg. invariance in the measurements of image similarity). In that case, it will be possible to define a certain value of $\rho_{max}^{(t)}$ as stopping criterion.
- $\eta_{max}^{(t)} = \eta$ to directly measure the quality of $R^{(t)}$ using the idea that such cell $V_{R^{(t)}}(r_j)$ will be repartitioned in the $n-$OOVD and lead to a set of cells containing one object only, while creating a minimum number of empty cells. This constraint is directly considered in our density local approach.

3.5 Evaluation

To evaluate our selection policies, we applied our algorithms on synthetic and real datasets and compared them to other reference points selection methodologies. These methodologies are:

1. *Random:* The selection of R is done randomly
2. *Distributed:* Close reference points are neglected based on a certain threshold value. Hence, if one of the new selected points is close to an already selected point, the selection is ignored.
3. *Clustered:* The database objects are clustered into a number of clusters to support the selection of the reference points. Then a group of reference points are randomly selected from each cluster. We thus ensure that the objects located in the same cluster have the same reference points as the primary items in their order lists.
4. *Global Locality Approach:* Discussed in section 3.4.1.
5. *Dense Locality Approach:* Discussed in section 3.4.2.

The evaluation is performed by choosing n reference points with the already mentioned methodologies. We then measure three values for the five methodologies. The first value is $\eta_{max}^{(n)}$ (Eq. 3.9) which is the maximum number of objects per Voronoi cell. The second value is $\rho_{max}^{(n)}$ (Eq. 3.8) which is maximum radius of a Voronoi cell. The third value is the average radius of all Voronoi cells in the Voronoi diagram $\underset{j}{\text{avg}}(\rho_j^{(n)})$).

3.5.1 Synthetic Dataset

We generated two synthetic datasets of 2-dimensional vectors. The first dataset is uniformly distributed and the second dataset is clustered. The number of objects in each dataset is $N = 3979$. We selected a set $n = \{10, 50, 100, 1000\}$ of reference points from each dataset. The stopping criterion for the *global* and the *dense* locality approaches is $|R^{(t)}| = n$. We use small 2-dimensional datasets for visualisation purposes. For the *clustered* approach the data was clustered into 5 clusters.

Uniform

Figures 3.8(a), 3.8(b), 3.8(c) and 3.8(d) measure the percentage of Voronoi cells that has $\eta_{max}^{(n)} \leq 10$, $10 < \eta_{max}^{(n)} \leq 100$ and $\eta_{max}^{(n)} > 100$ with respect to 10, 50, 100 and 1000 reference points for the five reference selection methodologies.

It is clear from the figures that, with low number of reference points that does not cover the space (10 references), whatever the selection technique used, the majority of the Voronoi cells has more than 100 objects per cell. When the number of reference points increases, the number of objects per Voronoi cell decreases. This can be seen as incremental Voronoi diagram construction. Adding more reference points splits the current Voronoi cells, which leads to decrease the number of objects per Voronoi cell and leads to a better identification of the database objects using the surrounding reference points. For example, for 10 reference points most of the Voronoi cells have more than 100 objects. For 1000 references, the majority of the cells has less than 10 objects per Voronoi cell.

Figures 3.8(e) and 3.8(f) show $\rho_{max}^{(n)}$ and $avg\rho_{max}^{(n)}$ respectively. For the *global* and the *dense* locality approaches whatever the number of reference points $\rho_{max}^{(t)} \simeq avg\rho_{max}^{(t)}$. That means that the Voronoi cells have almost the same size. That implies that the reference points are well covering the database and the database objects are well categorised. For the other approaches with small number of reference points $\rho_{max}^{(t)} \not\simeq avg\rho_{max}^{(t)}$. That means that the reference points do not cover the space properly. There are some Voronoi cells that are bigger than the others. When the number of reference points increases $\rho_{max}^{(n)}$ and $avg\rho_{max}^{(n)}$ become equal and similar to the *global* and the *dense* locality approaches, because the reference points cover the space.

Figure 3.9 shows the distribution of the reference points in the dataset for the five selection strategies. For the *Random* approach, there is no pattern for the distribution of the reference points. There are some areas which have more reference points than the others with no justification. For the *Distributed* approach the reference points cover well the space. However, if the threshold value is small, the reference points would not give the same distribution. They would be more close to each other as we will discuss in the next section. For the *Clustered* approach, the reference points cover the space even with small number of objects. However, the reference points which are located in the same cluster are more close to each other. Hence, if the data is not clustered to appropriate number of clusters there is a high probability

that some clusters would have more objects than the others. For the *Global Locality* approach, whatever the number of reference points, the methodology assures well distributed reference points in the space, which leads to an almost equal size for Voronoi cells. For the *Dense Locality* approach, as the data is uniform, it gives similar distribution to the *Global Locality* approach. However, there are some regions that have more reference points that others. That means that this area where the close reference points are located is more dense. This effect increases when the data is clustered which is discussed in the next section.

Clustered Data

Figures 3.10(a), 3.10(b), 3.10(c) and 3.10(d) measure the percentage of Voronoi cells that has $\eta_{max}^{(n)} \leq 10$, $10 < \eta_{max}^{(n)} \leq 100$ and $\eta_{max}^{(n)} > 100$ with respect to 10, 50, 100 and 1000 reference points for the five reference selection methodologies.

Using the *Dense cluster* approach, with small number of reference points (10R), more than 20% of the Voronoi cells have less than 10 objects. The reason is that the dense locality approach target the Voronoi cells with high density and split them. That helps to distribute the references based on the density of the data. That is also verified through Figures 3.10(e) and 3.10(f). The $\rho_{max}^{(n)}$ and $avg\rho_{max}^{(n)}$ for the *Dense* approach are equal compared to the other approaches. Hence, this technique maintains a well-balanced representation of the reference points whatever the distribution of the data. This is proved in figure 3.11. It is clear from the figure that the *dense locality* approach adds more points in the dense regions, compared to the other approaches. For the *Distributed* approach, most of the reference points are located in the same region. The main reason for that is the threshold value. It was not large enough to spread the reference points in the data domain.

3.5.2 Real Dataset

As real dataset, we chose to work with the Color histogram dataset (112-dimensional vector) (DS_2 in section 2.6.3). We generated 3 datasets from it. Hence, we have 4 datasets of different dimensions as follows:

- Dim_5 represents the first 5 dimensions from 112-dimensional vector
- Dim_{10} represents the first 10 dimensions from 112-dimensional vector
- Dim_{60} represents the first 60 dimensions from 112-dimensional vector
- Dim_{112} is the full dataset.

We selected a set $n = \{100, 500, 1000, 2000, 5000, 10000\}$ of reference points from each dataset. For the cluster selection technique, the data was clustered into 10 clusters.

Similar to the synthetic dataset, Figures 3.12, 3.13, 3.14 and 3.15, shows that for a low number of reference points (100 and 500) the *global locality* and *dense locality* approaches give the best data distribution compared to the other approaches. When the number of reference points increases, the performance of all the approaches becomes similar.

For the $\rho_{max}^{(n)}$ and $avg\rho_{max}^{(n)}$ values for the *global locality* and *dense locality* approaches are almost equal compared to the other approaches. However, these values increase and the difference between them increases when the dimensions increases. The main reason for that is the curse of dimensionality. When the number of dimensions increases, the sparsity of the data increases. With more reference points, whatever the approach is, the sparsity decreases. In addition, the difference between $\rho_{max}^{(n)}$ and $avg\rho_{max}^{(n)}$ for the *global locality* and *dense locality* approaches is smaller compared to the other approaches. This is due to the appropriate distribution of the reference points.

3.6 Summary

The idea of encoding the database objects based on the surroundings becomes more used in many domain. For indexing, this technique is known as permutation-based indexing.

We gave in this chapter a geometric justifications of the permutation-based indexing by analysing its encoding model. We consider that the distribution of the reference points gives a Voronoi diagram representation. Where each Voronoi cell is represented by a reference points and all the objects that are close to this reference point are located in the same Voronoi cell. When the object are represented by n reference points, we obtain Ordered k-Order Voronoi Diagram (OOVD). We showed that there is a direct relationship between the geometrical organisation of the points and the organisation of these ordered lists.

We also proposed two different algorithms for selecting the reference points. The two algorithms are inspired from the construction algorithm of the Voronoi diagram. The first algorithm aims to cover the space based on the size of each Voronoi cell. The second algorithm follows the same approach considering the dense areas in the database. We also evaluated our proposed algorithms on synthetic and real datasets compared to other selection approaches.

For the number of reference points n, our algorithms identify it based on the distribution of the data. However, it can be considered as a parameter in our algorithm. For the nearest number of reference points $\tilde{n}$, it is based on the memory limitation and the performance that is needed as shown imperially in the next chapter.

In general, whatever the reference selection approach is, with large number of reference points, the resulting distribution becomes the same. For low number of reference points, our proposed algorithms (*global locality* and *dense locality*) achieve a better distribution for the reference points compared to the other techniques. That leads to improving the indexing time, searching time, recall as we will discuss in

the next chapter.

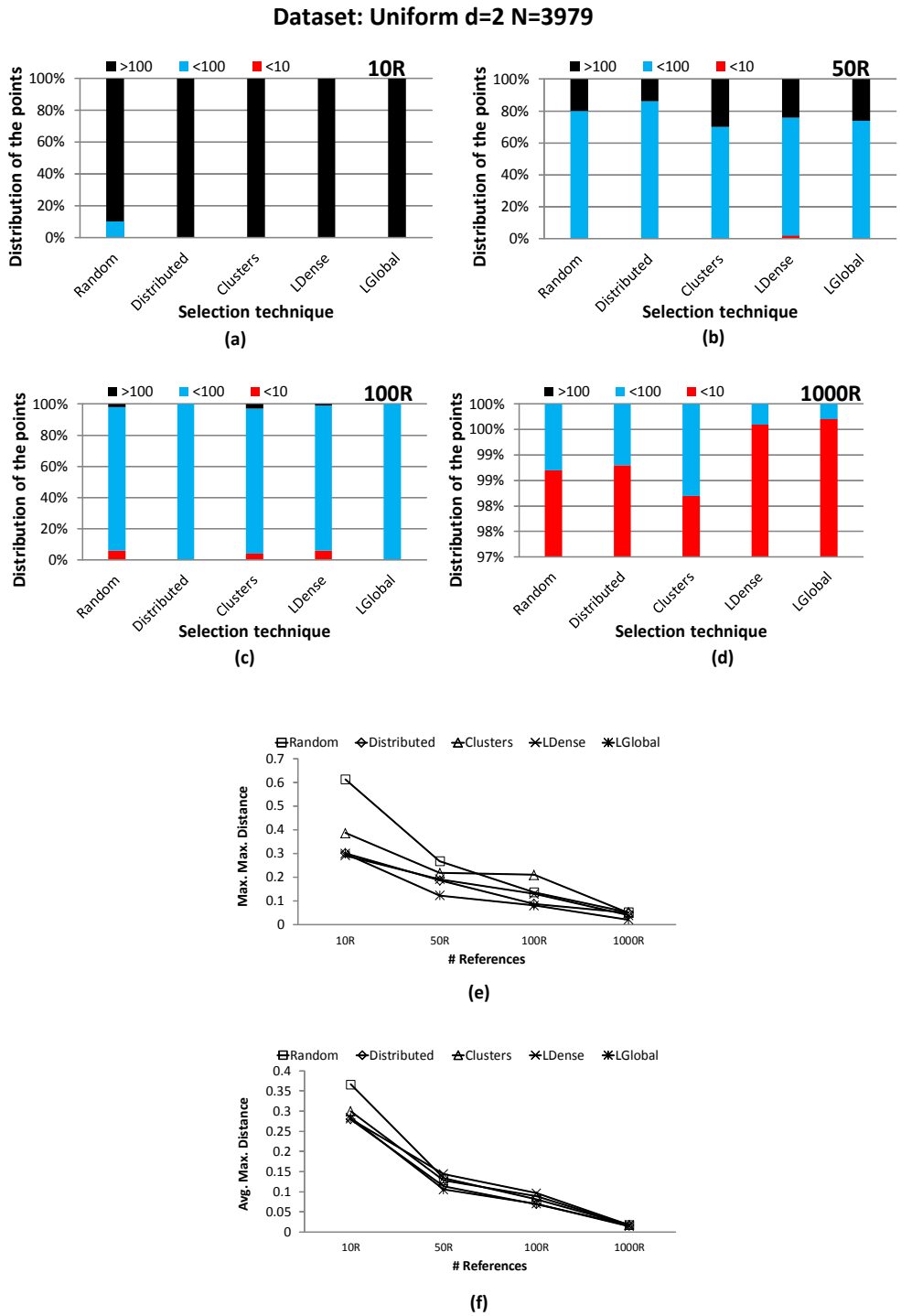


Figure 3.8: Uniform Dataset: a) Voronoi cells distribution using 10 reference points b) Voronoi cells distribution using 50 reference points c) Voronoi cells distribution using 100 reference points d) Voronoi cells distribution using 1000 reference points. e) $\rho_{max}^{(n)}$ of a Voronoi cell f) Average size of Voronoi cells $\underset{j}{\text{avg}}(\rho_j^{(n)})$).

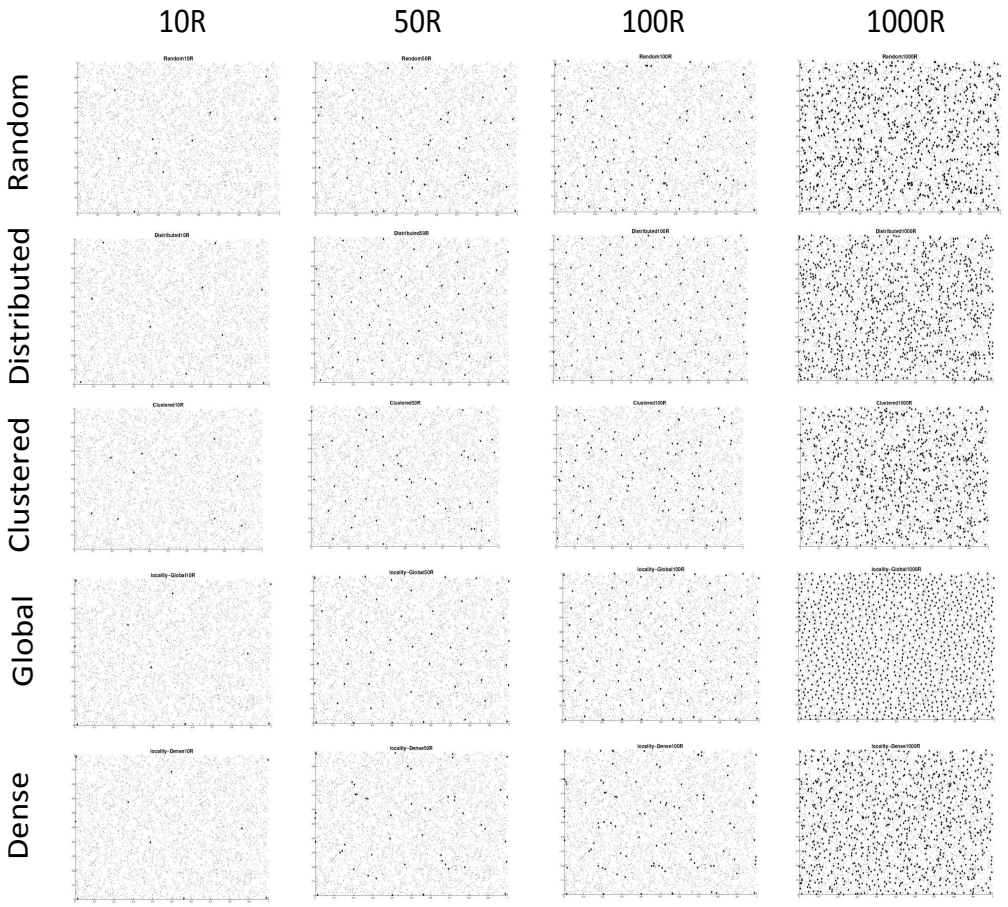


Figure 3.9: 10, 50, 100 and 1000 Reference points distribution with respect to the dataset for the random, distributed, clustered, global locality and dense locality approaches.

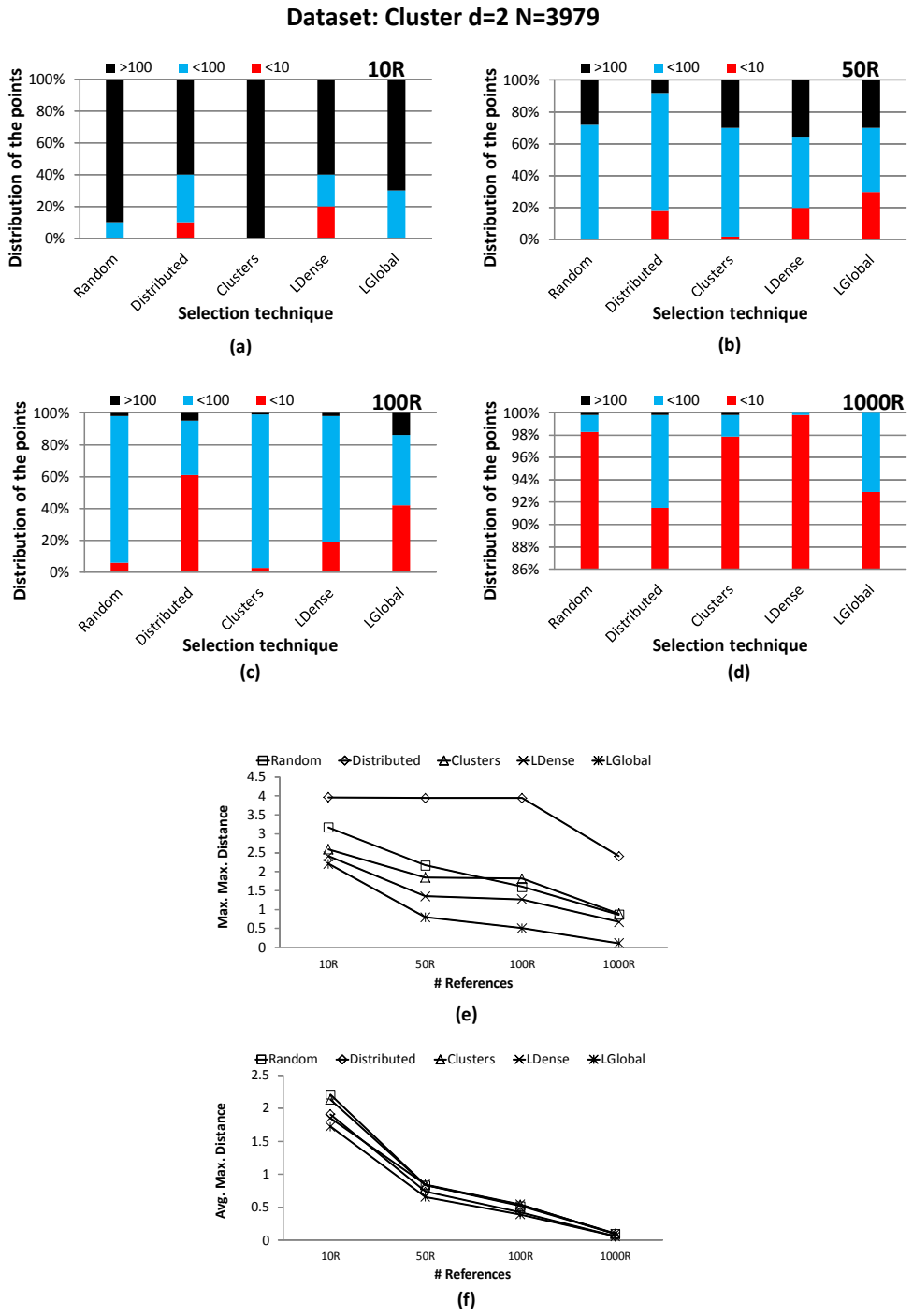


Figure 3.10: Clustered Dataset: a) Voronoi cells distribution using 10 reference points b) Voronoi cells distribution using 50 reference points c) Voronoi cells distribution using 100 reference points d) Voronoi cells distribution using 1000 reference points. e) Maximum size $\rho_{max}^{(t)}$ of a Voronoi cell f) Average size of Voronoi cells $\underset{j}{\text{avg}}(\rho_j^{(n)})$).

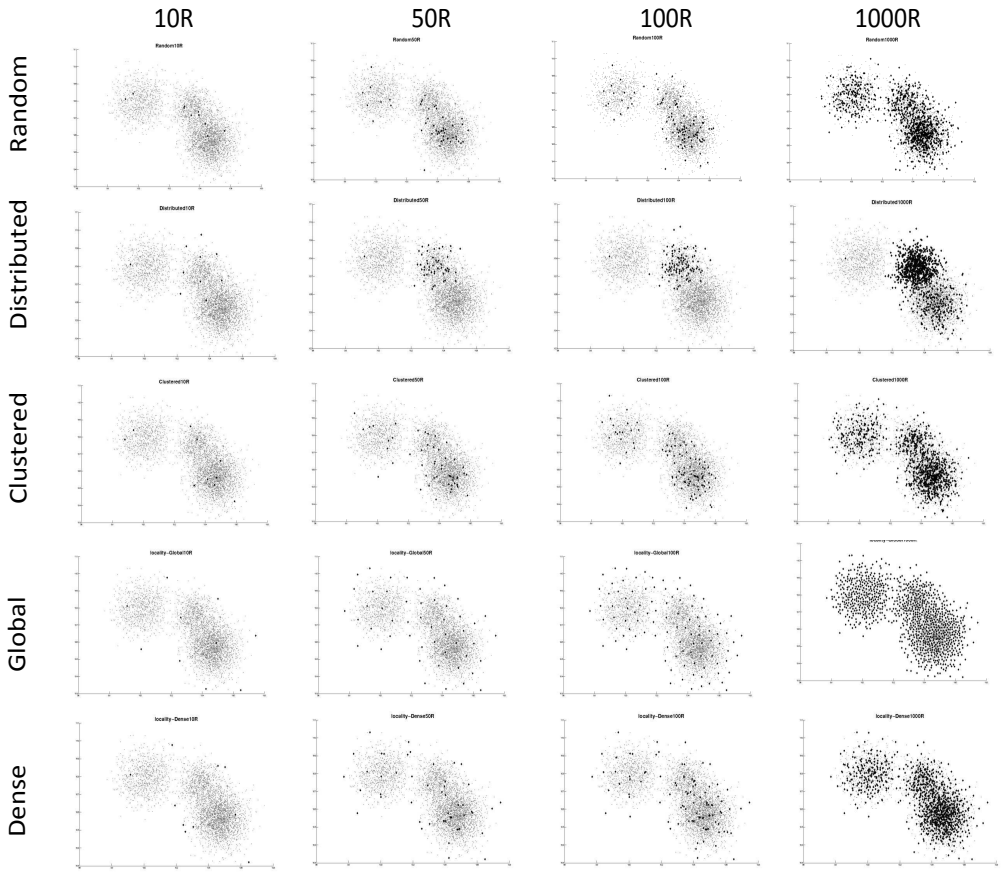


Figure 3.11: 10, 50, 100 and 1000 Reference points distribution with respect to the dataset for the random, distributed, clustered, global locality and dense locality approaches.

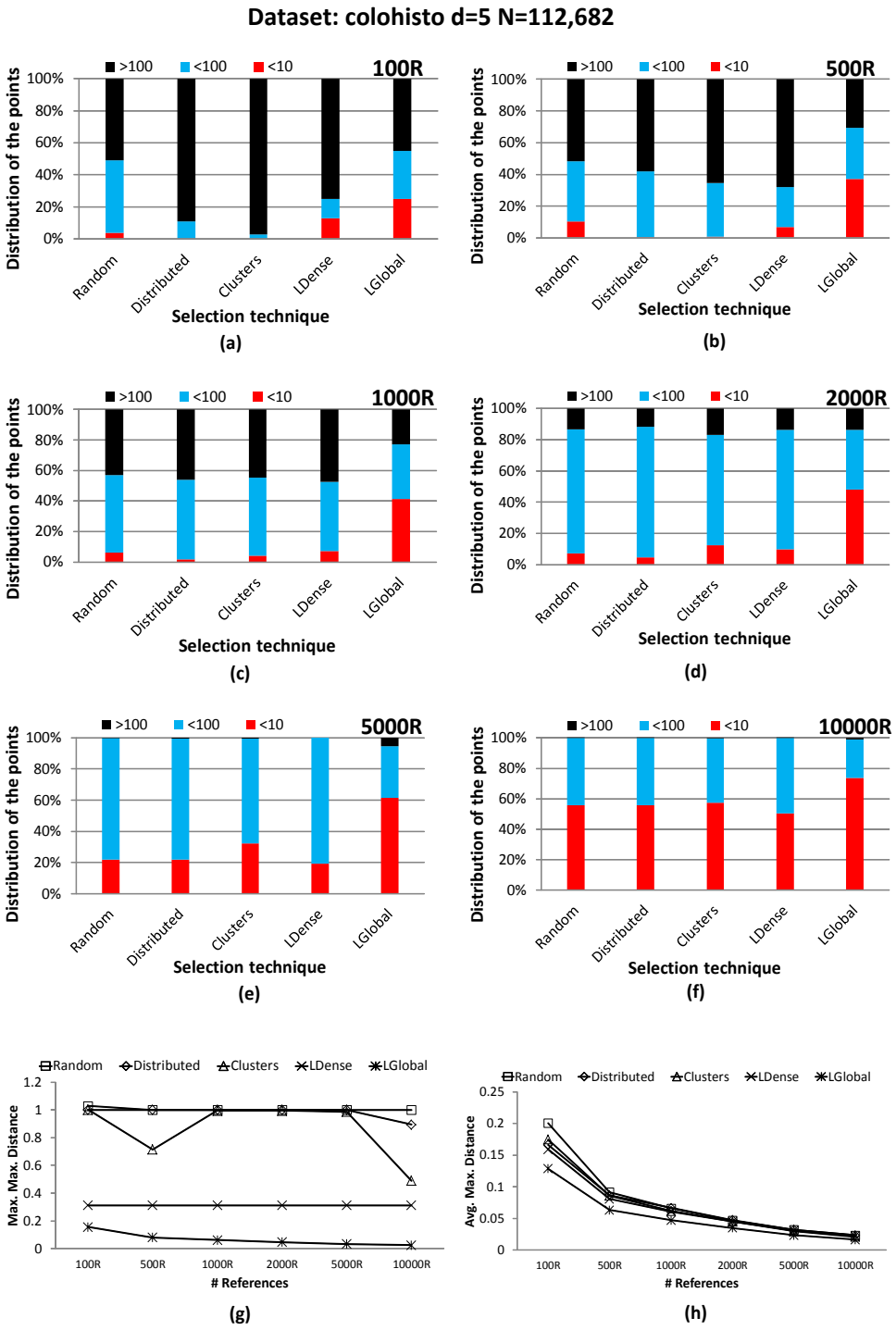


Figure 3.12: Dim_5 dataset: a) Voronoi cells distribution using 100 reference points b) Voronoi cells distribution using 500 reference points c) Voronoi cells distribution using 1000 reference points d) Voronoi cells distribution using 2000 reference points. e) Voronoi cells distribution using 5000 reference points f) Voronoi cells distribution using 10000 reference points g) Maximum size $\rho_{max}^{(t)}$ of a Voronoi cell h) Average size of Voronoi cells $\underset{j}{\text{avg}}(\rho_j^{(n)})$).

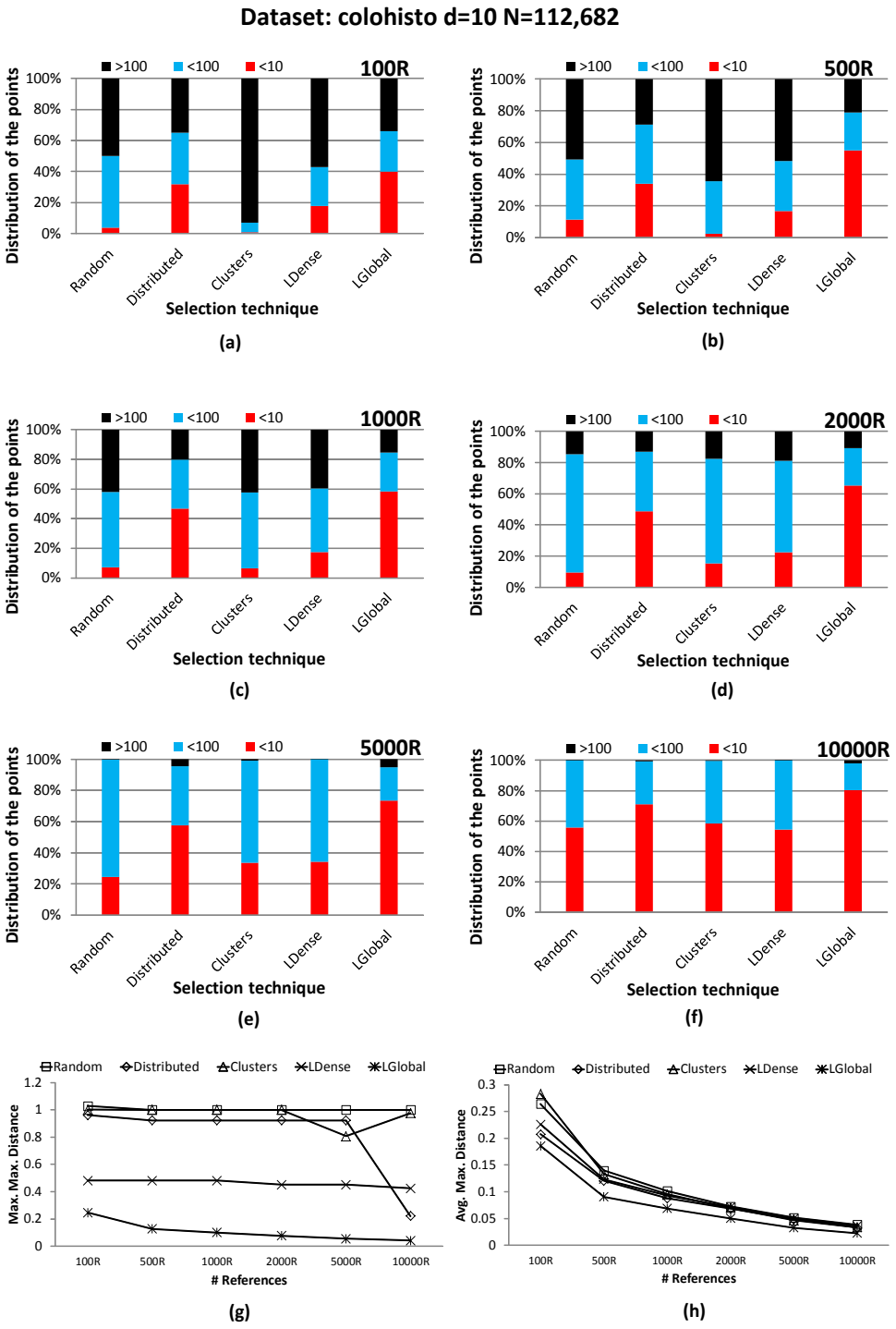


Figure 3.13: Dim_{10} dataset: a) Voronoi cells distribution using 100 reference points b) Voronoi cells distribution using 500 reference points c) Voronoi cells distribution using 1000 reference points d) Voronoi cells distribution using 2000 reference points. e) Voronoi cells distribution using 5000 reference points f) Voronoi cells distribution using 10000 reference points g) Maximum size $\rho_{max}^{(t)}$ of a Voronoi cell h) Average size of Voronoi cells $\underset{j}{\text{avg}}(\rho_j^{(n)})$).

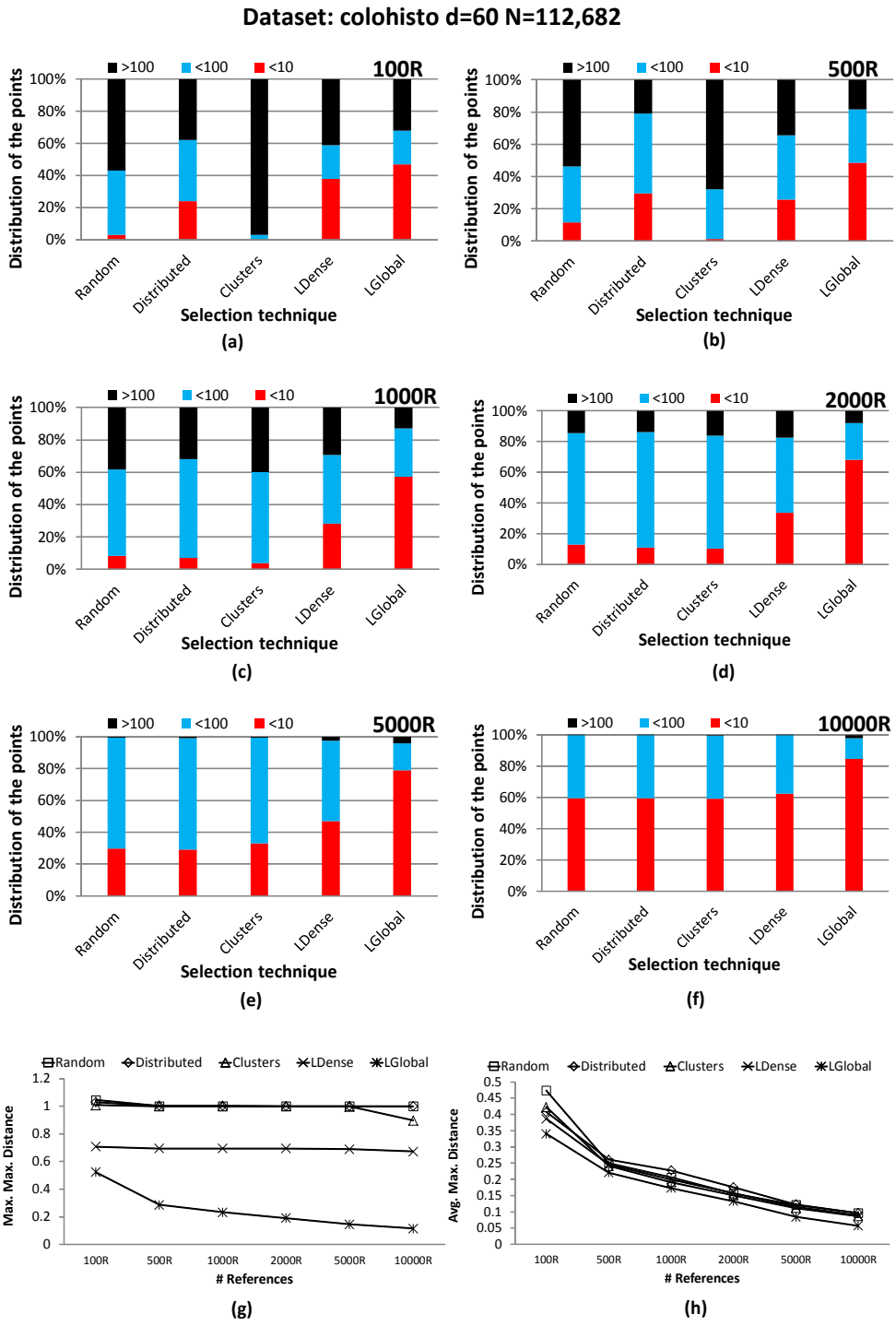


Figure 3.14: Dim_{60} dataset: a) Voronoi cells distribution using 100 reference points b) Voronoi cells distribution using 500 reference points c) Voronoi cells distribution using 1000 reference points d) Voronoi cells distribution using 2000 reference points. e) Voronoi cells distribution using 5000 reference points f) Voronoi cells distribution using 10000 reference points g) Maximum size $\rho_{max}^{(t)}$ of a Voronoi cell h) Average size of Voronoi cells $\underset{j}{\text{avg}}(\rho_j^{(n)})$).

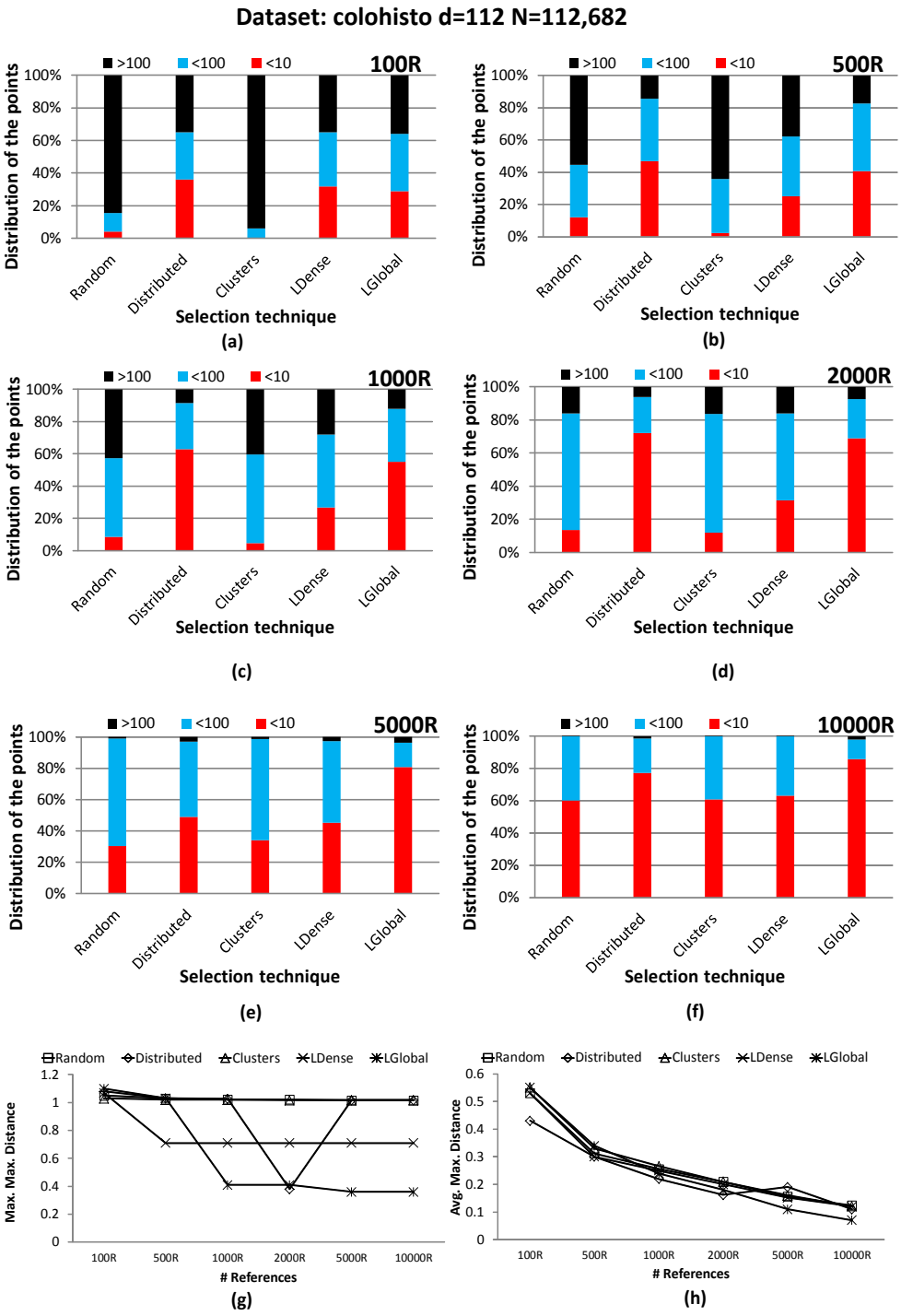


Figure 3.15: Dim_{112} dataset: a) Voronoi cells distribution using 100 reference points b) Voronoi cells distribution using 500 reference points c) Voronoi cells distribution using 1000 reference points d) Voronoi cells distribution using 2000 reference points. e) Voronoi cells distribution using 5000 reference points f) Voronoi cells distribution using 10000 reference points g) Maximum size $\rho_{max}^{(t)}$ of a Voronoi cell h) Average size of Voronoi cells $\underset{j}{\text{avg}}(\rho_j^{(n)})$).

Chapter 4

Permutation-Based Indexing Data Structures

Chapter 3 proposes a formal model for efficient reference-based approximate similarity search. The model is built around the approximation of distance-based ranking, extending the permutation-based indexing strategy. In this chapter, we propose two different data structures to store and use the permutation lists efficiently.

The chapter is organized as follows. In section 4.1, we propose the Metric Suffix Array (MSA) to store the permutations. Due to memory limitations of the MSA data structure, we propose the *Metric Permutation Table* (MPT) in section 4.2 for large scale indexing. In section 4.3, we perform a comparative study between the proposed techniques and the existing techniques, that were discussed in chapter 2. We conclude in section 4.4.

4.1 Metric Suffix Array (MSA)

We propose the Metric Suffix Array (MSA) as an efficient way to index permutation lists. The MSA is based on the suffix array data structure [96]. In section 4.1.1, we recall the principles of indexing with suffix arrays. We then extend these principles for a different usage in section 4.1.2.

4.1.1 Suffix Array

Let S be a string of length $m = |S|$ over a finite ordered *alphabet* Σ. Assume that the special symbol \$ is an element of Σ and appears once at the end; $S[m] = \$$. $S[i]$ indicates the character at position i in S, for $0 \leq i < m$. $S[i..j]$ represent the sub-string of S starting with the character at position i and ending with the character at position j. The sub-string $S[i..m]$ is the i-th suffix of S, and is denoted by $S(i)$.

The suffix array suff of the string S is an array of integers in the range 0 to m specifying the lexi-

i	suff	$S(\mathsf{suff}[i])$
1	10	\$
2	2	$aaacatat\$$
3	3	$aacatat\$$
4	0	$acaaacatat\$$
5	4	$acatat\$$
6	8	$at\$$
7	6	$atat\$$
8	1	$caaacatat\$$
9	5	$catat\$$
10	9	$t\$$
11	7	$tat\$$

Figure 4.1: The suffix array of the string $S = acaaacatat\$$. It is easy to find the substring '*at*' at positions 6 and 8.

cographic order of the m suffixes of the string S. That is, $S(\mathsf{suff}[0]), S(\mathsf{suff}[1]), \ldots, S(\mathsf{suff}[m])$ is the sequence of suffixes of S in ascending lexicographic order. Suffix arrays are used to locate every occurrence of sub-string B within S in an effective way for many applications such as Bioinformatics [96, 97, 98, 99]. Figure 4.1, illustrate the construction of a suffix array.

4.1.2 MSA Indexing Model

As it was discussed in section 3.1, we do not need the full permutation list. The pruned ordered list is sufficient to differentiate between the objects. Given all partial ordered lists $\tilde{L}(o_i, R)$, we construct $S = \bigcup_{i=1}^{N} \tilde{L}(o_i, R) = \{r_{i_1}, \ldots, r_{i_M}\}$, where $M = \tilde{n} \times N$. The set S can then be seen as a string of length M on the alphabet R. A Metric Suffix Array Ψ acts like a Suffix Array suff over S. More specifically, Ψ is a set of M integers corresponding to the permutation induced by the lexical ordering of all M suffixes in S ($\{r_{i_k}, \ldots, r_{i_M}\}\ \forall k \leq M$). The MSA is sorted into buckets.

Definition 15. *A* bucket *for reference point r_j is a subset of the* MSA Ψ *from position b_j. The bucket is identified to its position in Ψ so that $b_j\ \alpha\ \Psi_{[b_j, b_{j+1}-1]}$.*

Bucket b_j contains the positions of all the suffixes of S of the form $\{r_j, \ldots, r_{i_M}\}$, i.e. where reference point r_j appears first.

For example, in Figure 4.2(c), the string S corresponding to the objects and reference points shown in Figure 4.2(a) is given. The MSA Ψ is also shown. The bucket b_2 for reference point r_2 ($\mathsf{buk}_{\mathsf{r}_2}$) contains the positions in S of suffixes starting by r_2.

At query time, $\tilde{L}(q, R)$ is computed. From Eq.(3.2), we therefore need to characterise the objects

$$\{o_i \text{ s.t } r_j \in \tilde{L}(o_i, R), \forall r_j \in \tilde{L}(q, R) \text{ and } \tilde{L}(o_i, R)_{|r_j} = \tilde{L}(q, R)_{|r_j}\}$$

The MSA along with buckets encodes enough information to recover the relationships between an object

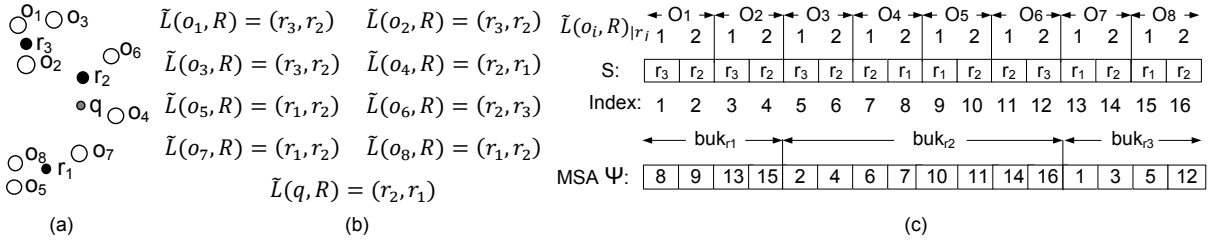


Figure 4.2: a) White circles are data objects o_i; black circles are reference objects r_j; the gray circle is the query object q b) Ordered lists for all the objects L_{o_i}. c) Example of Metric Suffix Array MSA

o_i and a given reference point r_j. Given $r_j \in \tilde{L}(q,R)$, we scan Ψ at positions $k \in [b_j, b_{j+1}-1]$ and determine i, and $\tilde{L}(o_i,R)_{|r_j}$ from Ψ_k as follows:

$$i = \left\lfloor \frac{\Psi_k}{\tilde{n}} \right\rfloor + 1 \qquad \tilde{L}(o_i,R)_{|r_j} = (\Psi_k \bmod \tilde{n}) + 1 \tag{4.1}$$

within each bucket b_j, $\tilde{L}(o_i,R)_{|r_j} \leq n$.

Using Eq.(4.1) and Eq.(3.3), all the MSA cells should be scanned in order to get the closest objects to a given query. To avoid scanning the complete MSA, the cells that represent objects that have a high difference in their permutations ordering should be pruned. If we are able to identify these cells, only a small part of the array is scanned, which improves the running time. In order to speedup further the scanning of buckets, we sort the buckets according to the value of $\tilde{L}(o_i,R)_{|r_j}$. Each bucket is divided into sub-buckets.

Definition 16. *A sub-bucket $b_j^{(l)}$ for* bucket b_j *points to objects o_i such that $\tilde{L}(o_i,R)_{|r_j} = l$, $1 < l < \tilde{n}$.*

For example, in Figure 4.3(b), the sub-bucket $b_1^{(1)}$ ($subuck_{r_1}$) points to all the suffixes that represent the objects that have the reference point r_1 as the first reference point in their pruned ordered list.

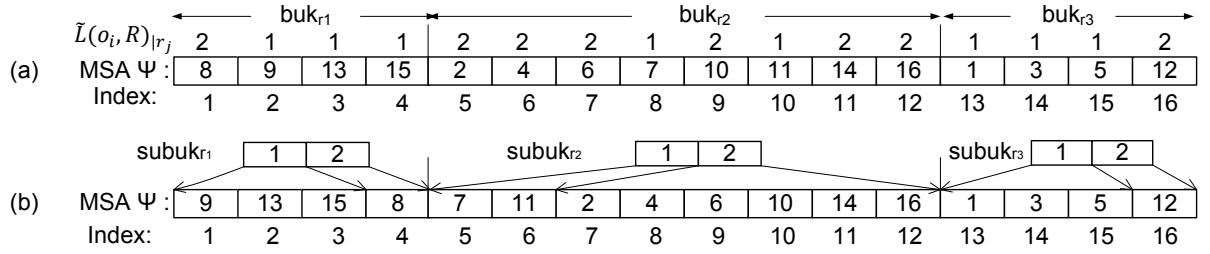


Figure 4.3: a) MSA sorted by bucket b) MSA sorted by sub-buckets.

4.1.3 MSA Practical setup

Indexing

For optimal memory usage, the MSA Ψ is built on the fly without actually constructing the string S. *CountSort* [22] is used for constructing the MSA Ψ using the pruned ordered list $\tilde{L}(o_i, R)$. The basic idea is to determine, for each suffix, the number of suffixes that are located before it. This helps placing the suffix value directly into its position in Ψ. Algorithm 4.1 details the indexing process.

In algorithm 4.1, lines 1-2 create an array buk_count of size n and initialize it with zero. This array is used to count the number of occurrences of each reference points in all the partial ordered lists. Lines 3-6 create the pruned ordered list $\tilde{L}(o_i, R) \forall o_i \in D$ and scan them reference by reference.

Once a reference point r_j is found, the corresponding counter $buk\_count[j]$ is incremented by one. Accordingly, the size of each bucket is defined. Lines 7-9 define the start and the end of each bucket. Since buckets are contiguous, the start of bucket b_j equals the end of the previous bucket $b_{j-1} + buk\_count[j-1]$.

Lines 10-14 fill Ψ. Counter C_j is created and initialized with zero for each reference point r_j. The counter is used to define the location of the references in the corresponding bucket. Ordered lists are scanned element by element and placed directly in their appropriate location in Ψ. After constructing the array, in lines 15-16, buckets are sorted based on $\tilde{L}(o_i, R)_{|r_j}$ (Eq.(4.1)) using the *Quicksort* algorithm [22]. All the suffixes representing the objects that have the same $\tilde{L}(o_i, R)_{|r_j}$ are located next to each other, which makes it easy to divide each bucket into $\tilde{n}$ sub-buckets $b_j^{(l)}$ (Line 17).

The complexity of creating the pruned ordered list is $O(Nnlogn)$. The complexity of creating the sub-buckets is $O(n\eta log\eta)$, where η is the average size of the buckets ($\eta \leq N$). Theoretically, the indexing complexity is $O(2N\tilde{n}+(Nnlogn)+(\eta log\eta)))$. The average memory usage is $O(M+(\tilde{n}\times n))$.

Searching

Equation (3.3) measures the discrepancy in ranking from common reference objects between each object and the query. In practice, it can be simplified by counting the co-occurrences of each object with the query in the same (or adjacent) sub-buckets. That is, each object o_i scores

$$s_i = \left| \left\{ r_j \in \tilde{L}(q, R) \text{ such that } (r_j \in \tilde{L}(o_i, R) \text{ and } |\tilde{L}(o_i, R)_{|r_j} - \tilde{L}(q, R)_{|r_j})| \leq 1 \right\} \right| \qquad (4.2)$$

Objects o_i are then sorted according to their decreasing s_i scores. This sorted candidate list provides an approximate ranking of the database objects relative to the submitted query.

The search procedure is described in Algorithm 4.2. First $\tilde{L}(q, R)$ is computed (line 1). For each $r_j \in \tilde{L}_q$, active sub-buckets are identified (line 3). For each object o_i pointed, a count $s[i]$ of co-occurrences is computed using (Eq.4.1) (lines 4-5). Line 6 sorts the candidate list in decreasing order of

Algorithm 4.1 Metric Suffix Array Indexing

IN: Domain D of N objects,
References list R of n references, $\tilde{n}$
OUT: MSA: Ψ
1. For each $j \longleftarrow 1\ to\ n$
2. $buk\_count[j] = 0$
3. For each $o \in D$
4. Generate the pruned ordered list $\tilde{L}(o_i, R)$
5. For each $r_j \in \tilde{L}(o_i, R)$
6. $buk\_count[j] + +$
7. $b_1.l = 1$
8. For each $j \longleftarrow 2\ to\ n$
9. $b_j = b_{j-1} + buk\_count[j-1]$
10. For each $o \in D$
11. For each $r_j \in \tilde{L}(o_i, R)$
12. $k = b_j + C_j$
13. $\Psi_k = i \times \tilde{n} + \tilde{L}(o_i, R)_{|r_j}$
14. $C_j + +$
15. For each $r_j \in R$
16. $\mathrm{QuickSort}(\Psi, b_j, b_{j+1} - 1)$
17. Define the sub-buckets $b_j^{(l)}$ within each bucket b_j

their score (counters s). Theoretically, the computational complexity to retrieve the k-NN is $O(2\tilde{n}\eta)$.

Evaluation

We conducted large-scale experiments using 5-million shape features extracted from dataset DS_4 (mentioned in chapter 2), to evaluate the indexing and searching time of our proposed algorithm. Table 4.1 shows the average indexing and searching time (in seconds) for the sub-bucket implementation compared to using the full permutation list (FPL) and the partial permutation list (PPL) based on the metric inverted files (MIF). The number of nearest references used is half of the reference points $\tilde{n} = \frac{n}{2}$.

Table 4.1: Indexing and searching time (in seconds)

n	Index FPL	Search FPL	Index PPL	Search PPL	Index Sub-bucket	Search Sub-bucket
100	45	4	86	1.5	113	**0.35**
1000	517	46	735	12	1622	**0.45**
2000	1081	94	1651	25	3523	**0.76**

The MSA sub-bucket technique is efficient in searching in terms of recall and searching time. For

Algorithm 4.2 Sub-buckets Searching

IN: Query: q, R of n, MSA, $b_j^{(l)}$, $s[0 \dots N]$
OUT: Sorted Objects list: out
1. Create the query ordered list $\tilde{L}_q$
2. For $r_j \in \tilde{L}_q, l = P(\tilde{L}_q, r_j)$
3. For $k = b_j^{(l)}$ to $b_j^{(l+1)} - 1$
4. $i = \left\lfloor \frac{\Psi_k}{\tilde{n}} \right\rfloor + 1$
5. $s[i] = s[i] + 1$
6. sort(s)
7. $sort(out)$

different n, it appears clearly from Table 4.1 that the sub-bucket technique is faster than the other techniques, as the algorithm does not need to scan all the MSA cells. On the other hand for indexing, the time for the sub-buckets algorithm is longer than that of the other algorithms. This is due to the sorting and defining the sub-buckets after building the MSA. Another limitation is the memory usage. We claimed that we need only 4 bytes for each reference point. This claim is true only if the size of the string $M < 2^{32}$. When $M > 2^{32}$, 8 bytes are needed for each suffix array value.

Hence, the MSA is a fast data structure for k-NN similarity search in permutation-based indexing. On the other hand, it is not fully efficient in terms of indexing time and memory usage. That leads us to propose another data structure based on the same principles of searching, while improving the indexing time and the memory usage.

4.2 Metric Permutation Table (MPT)

The metric permutation table is based on quantizing the ordered lists. This helps to decrease the size of the ordered lists, improves the memory usage and improves the indexing time.

4.2.1 Rank Quantization

It is clear from the previous section that the number of reference points is an important factor for permutation-based indexing. The number of reference points needs to increase as the data size increases for a better discrimination. Accordingly, aggregating and summarizing the ordered lists per object is required, especially for large-scale data. The strategy of quantizing the data into a given number of buckets in order to concisely summarize and access it is still an open problem, although it is used in many domains. We will apply it here and study its effect.

Eq.(3.3) is the base for the construction of our indexing scheme. The main idea is to simplify further the distance approximation by quantizing the ranks within the ordered lists into $B \leq \tilde{n}$ intervals

(buckets). We define

$$b_{ij} = \left\lceil \frac{B}{\tilde{n}}.\tilde{L}(o_i, R)_{|r_j} \right\rceil \quad \text{and} \quad b_{qj} = \left\lceil \frac{B}{\tilde{n}}.\tilde{L}(q, R)_{|r_j} \right\rceil \tag{4.3}$$

as regular quantization scheme. The similarity between a query q and object o is defined as

$$d(q, o_i) \overset{\text{rank}}{\simeq} \sum_{\substack{r_j \in \tilde{L}(q,R) \\ r_j \in \tilde{L}(o_i,R)}} |b_{qj} - b_{ij}| \tag{4.4}$$

In the next section, we detail the corresponding data structure and the corresponding indexing and search procedures.

4.2.2 MPT Practical setup

We show here how we construct an efficient data structure to support our indexing strategy and explicit the search procedure, based on this data structure.

Indexing

Eq.(4.3) above defines a bucket index b_{ij} as the quantized rank of reference object r_j with respect to object o_i. Considering how these factors will be used, it is useful to centre the construction of our data structure on the fact of efficiently answering the selection

$$r_j \in \tilde{L}(q, R) \quad \text{and} \quad r_j \in \tilde{L}(o_i, R)$$

used in Eq.(4.4). Note that, if $\tilde{L}(o_i, R) \cap \tilde{L}(q, R) = \phi$, point o_i should simply not be considered since it clearly does not belong to the neighbourhood of q.

At query time, $\tilde{L}(q, R)$ is computed. We therefore need to characterise the set

$$\{o_i \text{ s.t } r_j \in \tilde{L}(o_i, R), \forall r_j \in \tilde{L}(q, R)\}$$

which is the typical use case of an inverted file. Since we quantize $\tilde{L}(o_i, R)_{|r_j}$ index into B buckets, we construct the inverted file at the time of quantization. More formally the indexing procedure is given in Algorithm 4.3, using the MPT data structure presented in Figure 4.4(a).

The average complexity for sorting the ordered lists is $O(nlogn)$ as we are using the QuickSort algorithm. Hence, the average complexity for building the MPT is $O(N(\tilde{n} + nlogn))$.

Algorithm 4.3 Metric Permutation Table: Indexing

IN: D of size N, R of n and $\tilde{n} \leq n$ and $B \leq \tilde{n}$
OUT: MPT
1. For $o_i \in D$
2. Build $\tilde{L}(o_i, R)$
3. For $r_j \in \tilde{L}(o_i, R)$
4. $b_{ij} = \lceil \frac{B}{\tilde{n}}.\tilde{L}(o_i, R)_{|r_j} \rceil$ (Eq.(4.3))
5. Store the ID i of o_i in the list l of bucket number b_{ij}

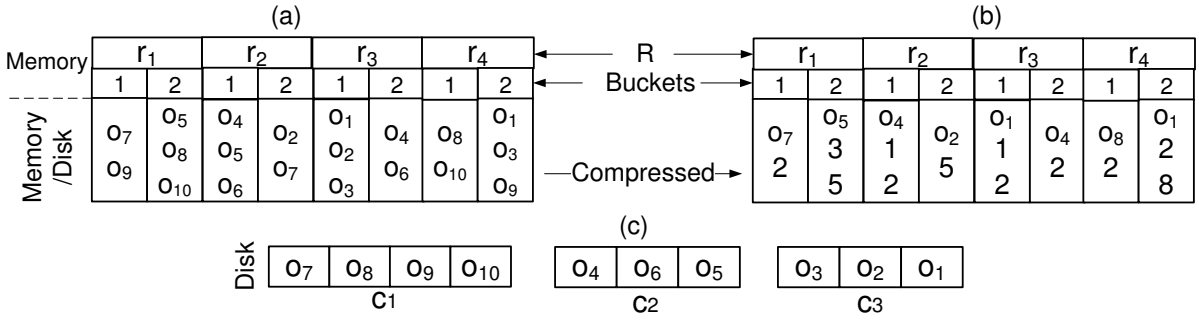


Figure 4.4: The MPT data structure

Searching

Equation (4.4) is the base for our search procedure. Essentially, it counts the discrepancy in quantized ranking from reference objects common between each object and the query. In practice, we further reduce the access to buckets. Instead of a *soft* (sum) counting of the difference in bucket index, we count the co-occurrences of each object with the query in the same bucket or neighboring buckets. That is, each object o_i scores

$$s_i = \left| \left\{ r_j \in \tilde{L}(q, R) \text{ such that } (r_j \in \tilde{L}(o_i, R) \text{ and } |b_{ij} - b_{qj}| \leq 1) \right\} \right| \tag{4.5}$$

This computation is efficiently supported by the above data structure. Objects o_i are then sorted according to their decreasing s_i scores. This sorted candidate list provides an approximate ranking of the database objects relative to the submitted query. This approximate ranking can be improved by direct distance calculation (DDC). Similar to the proposals in [8] (DDC factor) and [62] (amplification factor), for a k-NN query, we apply DDC on the $K_c = \Delta.K$ first objects in our sorted candidate list and call $\Delta > 1$ the *DDC factor*. The use of this parameter will be explored in our performance evaluation (section 4.3). The search procedure uses the above data structure as described in Algorithm 4.4.

First the pruned ordered list $\tilde{L}(q, R)$ is constructed (Line 2). Equation (4.3) is used to characterise

Algorithm 4.4 Metric Permutation Table: Search

IN: q, R of size n, Δ
OUT: Sorted Objects list: out
1. Create a list of counters $s[0 \dots N]$ and set them to 0
2. Build $\tilde{L}(q, R)$
3. For $r_j \in \tilde{L}(q, R)$
4. $b_{ij} = \lceil \frac{B}{\tilde{n}} . \tilde{L}(o_i, R)_{|r_j} \rceil$ (Eq.(4.3))
5. For $k = b_{ij} - 1$ *to* $b_{ij} + 1$
6. $obj_{id} = l[k][0]$
7. For $i = 1$ *to* $l[k].size()$
8. $s[obj_{id} + l[k][i]] + +$
9. $obj_{id} = obj_{id} + l[k][i]$
10. sort(s)
11. $K_c = K_{NN} \times \Delta$
12. $A \leftarrow s(0, K_c)$
13. out=calc_distance(A, K_c, q)
14. $sort(out)$

the active buckets (Line 4-5), which are decompressed (Line 6-9) to get the stored object IDs. For each such object, a count of co-occurrence is computed using Eq.(4.5). Line 10 sorts the candidate list in decreasing order of their score(counters s). Final DDC filtering is performed in lines 11-13.

The average complexity for accessing the reference lists and the buckets is $O(1)$. While it is $O(z)$ for the lists of each bucket, where z is the average size of a list. Hence, the total average searching complexity is $O(\tilde{n}z)$.

Optimizing the indexing

We propose two further optimization steps that help reducing costs in memory and disk access.

Compression: Since any given object o_i may appear in several buckets. The direct storage of objects IDs within the buckets is inefficient. Buckets store lists of IDs (large integers) and may thus be efficiently compressed using delta encoding [100]. This strategy consists in replacing a series of large integers (given values) by a series of smaller integers (their successive difference). For example, the sequence $\{562895, 562896, 562900, 562910\}$ requires 16 bytes (4 bytes per value) and may be replaced by $\{562895, 1, 4, 10\}$ that may be stored using 4 bytes for the first value and 2 bytes for each subsequent value (hence 10 bytes in total). This simple strategy is applied individually to buckets, leading to significant memory savings. Our compressed data structure is presented in Figure 4.4(b). These compressed lists may be saved on the hard-disk if they can not be fitted in the main memory.

Clustering: We cluster the original objects into a given number of clusters (using any cluster algorithm). Our motivation is to optimize the disk access overhead. This overhead depends on the size of the database and the hard disk. We cluster the database into C clusters $(C_1, \ldots, C_c)$ with the idea that objects sharing the same clusters will be accessed together. At query time, a reduced number of clusters should be accessed, containing the necessary points for further filtering (direct distance calculation, DDC). Figure 4.4(c) shows the clustered data.

Updating

MPT can easily handle the addition and deletion of objects. For new objects, it is similar to the insertion process. The objects ordered lists are calculated. Then, the objects are located in the suitable bucket efficiently. In terms of deletion, the objects can easily be removed from their corresponding buckets. However, the compression of the corresponding buckets should be recomputed as the difference between the objects in the lists changed.

Adding a new reference is not an easy process. The new reference point would change the order lists of some objects. That means, the location of these objects in the MPT have to be changed. More clearly, adding a new reference point is a combination of insertion, deletion and recompression. The most naive solution is the complete reconstruction of the MPT. Hence, adding a new reference point is considered as an expensive process due to the characteristics of the permutation-based indexing problem.

4.3 Experimental Results

Our data structures were implemented in C++. The experiments were run on the *Squid* machine (detailed in appendix A.1). We measure recall (RE), position error (PE) [15], indexing time, searching time and memory usage. In 4.3.1, we measure the general performance of the MPT on four datasets:

- DS_1 Color histogram (112,544 vector, 112-dimensional vector)
- DS_3 Color features (9-millions, 84-dimensional)
- DS_6 CoPhIR (106-millions, 208-dimensional)
- DS_5 CoPhIR (106-millions, 280-dimensional)

The datasets are detailed in section 2.6.3.

In section 4.3.2, we compare the performance of our proposed structure to that proposed in [61, 90, 3, 70, 62], based on the datasets reported in these publications.

For clustering the data, we used the *K-means* algorithm and the implementation available on [84] to cluster the ImageNet dataset. ImageNet was clustered into 20 clusters. For the CoPhIR dataset, we use

our disk-based *K-means* implementation, as the data can not be fitted in the memory. The dataset was clustered into 40 clusters. All the indexing times reported do not include the clustering time. We do not consider it as an important factor as it is only done once, whatever the number of reference points or buckets. However, the clustering time for the ImageNet dataset is around 7 hours and around 6 days for the CoPhIR dataset.

4.3.1 General Performance

Effect of B and Δ on RE

The selection of the number of buckets B and the parameter for direct distance calculation Δ affects the precision of the search result. Using the Color histogram dataset, we investigate their effect on the RE. Two sets of references objects of size $n = 256$ and $n = 512$ were selected from the dataset. For each set of reference points, the experiments are performed for pruned ordered lists of size $\tilde{n} = \{10, 20, 60\}$. For each pruned ordered list, a set buckets of sizes $B = \{1, \frac{\tilde{n}}{5}, \frac{\tilde{n}}{2}, \tilde{n}\}$ are selected. For example, if $\tilde{n} = 60$, the corresponding set of buckets sizes is $B = \{1, 12, 30, 60\}$.

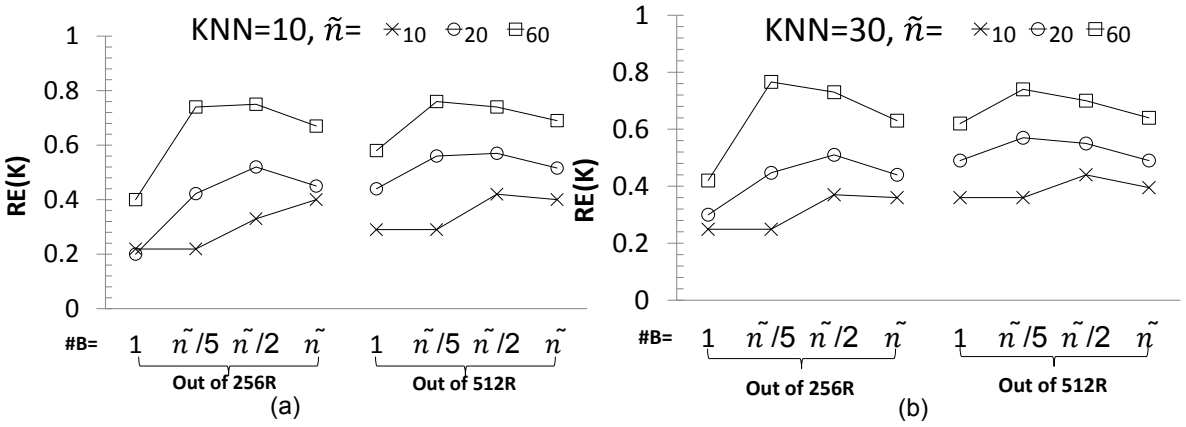


Figure 4.5: Effect of varying the value of B: using the Color histogram dataset, $B = \{1, \frac{\tilde{n}}{5}, \frac{\tilde{n}}{2}, \tilde{n}\}$, for $\tilde{n} = \{10, 20, 60\}$, for $n = 256$ and $n = 512$ a) RE for 10-NN b) RE for 30-NN.

Figures 4.5(a) and 4.5(b) show the average RE based on 200 randomly selected queries from the dataset for the 10-NN and 30-NN search scenarios for different number of references, pruned ordered lists and buckets.

We note that whatever the numbers of n and $\tilde{n}$ are, the RE has the lowest value when B is equal to 1. This is reasonable as it is similar to representing each object by 1 reference object. When B increases, the RE increases until a certain value and starts to drop back again. We believe that this increase in RE is due to concentrated references, which are at the same distance from an object into one position, as permutation-based indexing is only based on the ranking of the reference points. Hence, if there are l

reference points located at equal distances from an object o_i, there is a high probability that $\tilde{L}(o_i, R)$ has $l!$ different representations. All these different representations give the same information. When we apply the quantization, these references become represented by one position, which improves the results. Then, when we increase the number of buckets, the probability of having the reference points at the same distance from an object increases. Accordingly, it diminishes the RE. Hence, the effective number of buckets should be $B \simeq \frac{\tilde{n}}{2}$

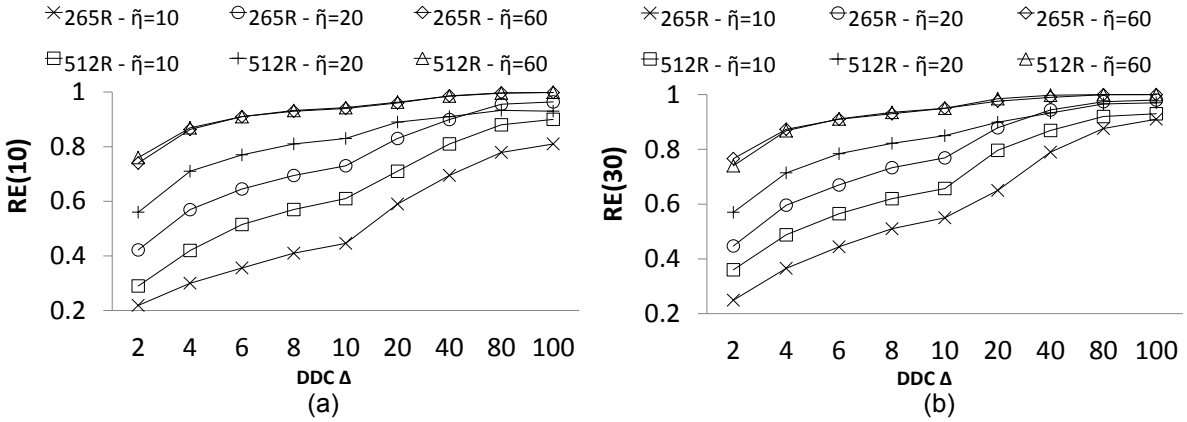


Figure 4.6: Effect of varying the value of Δ: using the Color histogram dataset, $\tilde{n} = \{10, 20, 60\}$ with buckets of sizes $B = \{5, 10, 12\}$ respectively for $n = 256$ and $n = 512$ a) RE for 10-NN b) RE for 30-NN.

Figures 4.6(a) and 4.6(b) show the average RE based on the same set of queries used in the previous experiments for the 10-NN and 30-NN search scenarios for different number of references, pruned ordered lists and Δ.

Based on the previous discussion, we fixed the number of buckets as follows. For pruned ordered lists of sizes $\tilde{n} = \{10, 20, 60\}$, the effective numbers of buckets are $B = \{5, 10, 12\}$, respectively. We note that when Δ increases, the RE increases. Reaching to an optimal RE can be achieved faster when the number of reference objects increases. For example, for $\tilde{n} = 60$, the algorithm is able to achieve RE $= 0.9$ with $\Delta = 6$.

Effect of n and $\tilde{n}$ with fixed B and Δ on RE and PE

Based on the discussion in the previous section, we ran exhaustive experiments on large datasets. The ImageNet dataset (9-million object, 84-dimensional) and the two CoPhIR datasets (106-million object, 208-dimensional) and (106-million object, 280-dimensional).

For each dataset, we observe empirically the value of B and Δ for good trade off values between performance and efficiency. For the ImageNet dataset, the experiments were performed fixing the number of buckets $B = 5$ and $\Delta = 40$ for pruned ordered lists of sizes $\tilde{n} = \{10, 20, 50\}$ for $n = 1000$ and

$n = 2000$ reference objects. For the CoPhIR datasets, the experiments were performed fixing $\Delta = 40$ for pruned ordered lists of sizes $\tilde{n} = \{10, 50, 100\}$ with $B = \{5, 10, 20\}$, respectively for $n = 1000$, $n = 2000$ and $n = 6000$ reference objects. For the two datasets, the average RE and PE were measured using 100 queries selected randomly from the dataset for the 1-NN, 10-NN and 100-NN search scenarios. Figures 4.7(a) and 4.7(b) show the average RE and PE for the ImageNet dataset. Figures 4.8(a) and 4.8(b) show the average RE and PE for the first CoPhIR dataset (208-dimensional). Figures 4.9(a) and 4.9(b) show the average RE and PE for the second CoPhIR dataset (280-dimensional).

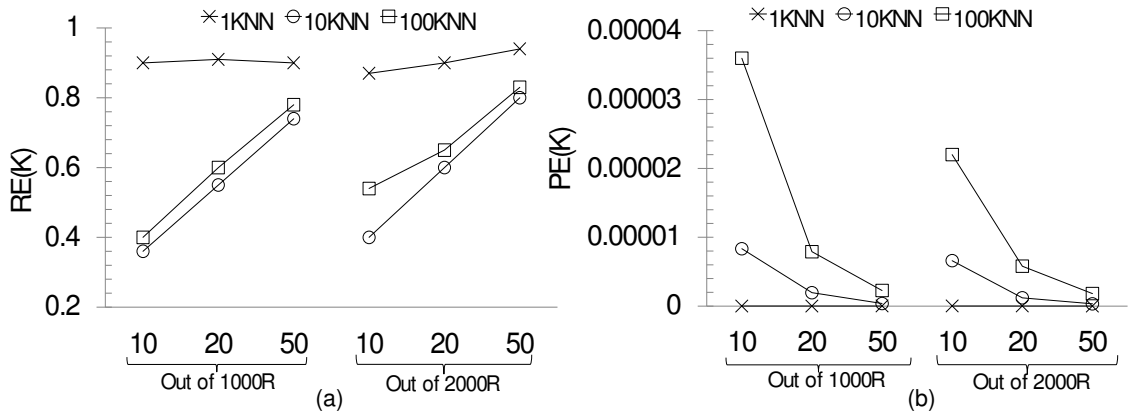


Figure 4.7: ImageNet: Using $\Delta = 40$, $B = 5$ and $\tilde{n} = \{10, 20, 50\}$ out of $n = 1000$ and $n = 2000$ a) RE for 1-NN, 10-NN and 100-NN queries. b) PE for 1-NN, 10-NN and 100-NN queries.

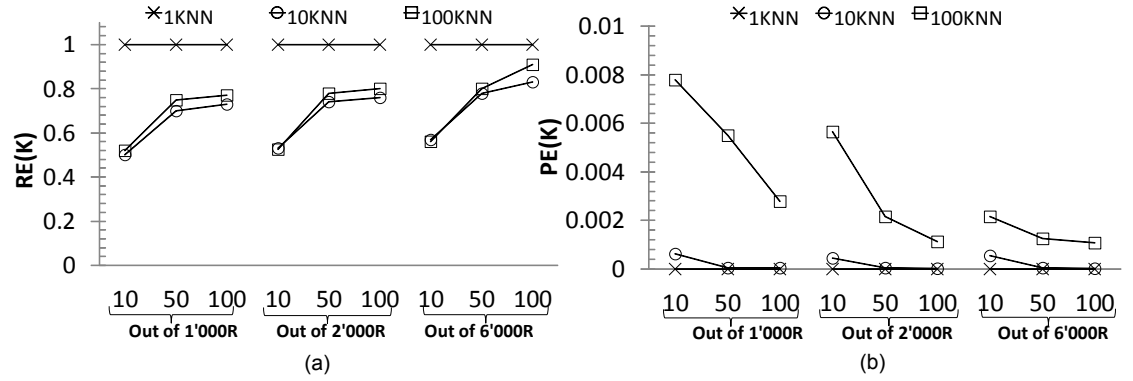


Figure 4.8: CoPhIR(106M, 208-dimensions): Using $\Delta = 40$, $\tilde{n} = \{10, 50, 100\}$ with $B = \{5, 10, 20\}$ respectively for $n = 1000$, $n = 2000$ and $n = 6000$ a) RE for 1-NN, 10-NN and 100-NN queries. b) PE for 1-NN, 10-NN and 100-NN queries.

We note that RE increases while PE decreases when $\tilde{n}$ increases. This shows that $\tilde{n}$ directly impacts the quality of approximation (as suggested in Eq.(3.3)). When comparing the RE and PE for the same number of nearest reference objects $\tilde{n}$ out of an increasing total number of reference objects n, we see

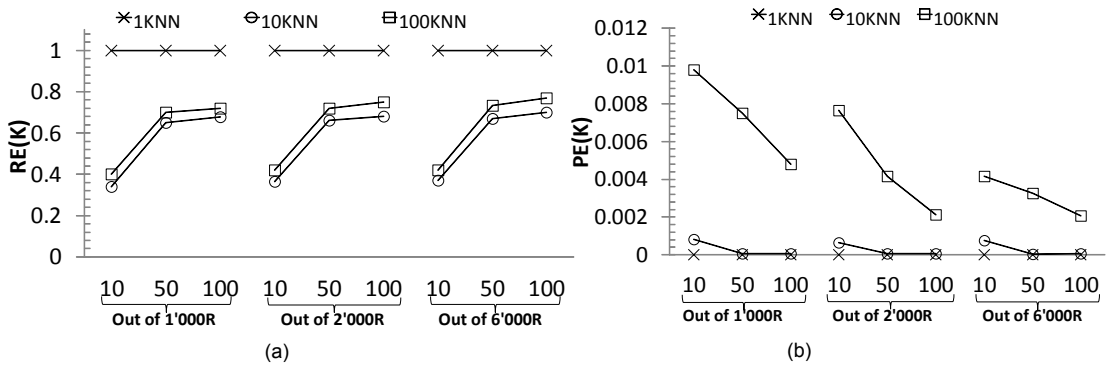


Figure 4.9: CoPhIR(106M, 280-dimensions): Using $\Delta = 40$, $\tilde{n} = \{10, 50, 100\}$ with $B = \{5, 10, 20\}$ respectively for $n = 1000$, $n = 2000$ and $n = 6000$ a) RE for 1-NN, 10-NN and 100-NN queries. b) PE for 1-NN, 10-NN and 100-NN queries.

that the RE increases and PE decreases. This is reasonable, since a higher n indicates a more dense set of reference objects and, therefore, a lower number of objects located in the same bucket for each reference object, thus decreasing the number of false positives. For example, for 10-NN for the ImageNet dataset (Figure 4.7(a)), if $\tilde{n} = 50$ out of $n = 1000$ and $n = 2000$ reference objects, RE$= 0.74$ and RE$= 0.8$, respectively.

The number of direct distance calculations K_c (line 11 in algorithm 4.4) for 1-NN, 10-NN and 100-NN queries is 40, 400 and 4000, respectively. This is about $4 \times 10^{-4}\%$, 0.004% and 0.04% of the total number of objects of the ImageNet database and it is about 0.00003%, 0.0003% and 0.003% of the number of objects of the CoPhIR dataset. Hence, using our technique, we significantly decrease the number of direct distance calculations between the query and the database objects even with a small set of reference objects.

From figures 4.8 and 4.9, we note that fixing the number of objects N, the reference objects n, the nearest reference objects $\tilde{n}$ and the number of buckets B for different dimensions, the RE decreases and PE increases when the dimension m increases. For example, for 100-NN, 106-million objects, $\tilde{n} = 100$ out of $n = 6000$ reference objects with 208-dimensions (Figure 4.8a) and with 280-dimensions (Figure 4.9a), RE$= 0.91$ and RE$= 0.77$ respectively. The reason for that is when the dimension increases more reference objects are needed to cover the space.

Indexing time, Searching time and Memory usage

Table 4.2 shows the average indexing time, searching time, uncompressed and compressed memory based on 100 queries for the ImageNet and the two CoPhIR datasets.

When the number of reference objects n increases, the indexing time consistently increases. On the other hand, the value of $\tilde{n}$ does not affect the indexing time, since the costly operations essentially depend

Table 4.2: Average indexing time, searching time, uncompressed and compressed memory for the MPT based on 100 queries

Data Set	n	$\tilde{n}$	B	Indexing Time	Searching time	Uncompressed memory	Compressed memory
ImageNet(9M)	1000R	10	5	52min	0.37s	367MB	252MB
		20	5		0.48s	720MB	427 MB
		50	5		0.75s	1773MB	955 MB
	2000R	10	5	102min	0.3s	370MB	267MB
		20	5		0.4s	719MB	442 MB
		50	5		0.62s	1773MB	973 MB
CoPhIR (106M - $m = 208$)	1000R	10	5	9h	4s	4GB	2.5GB
		50	10		5s	19GB	10GB
		100	20		6.9s	30GB	20GB
	2000R	10	5	19h	3s	4.2GB	3GB
		50	10		4s	20GB	11GB
		100	20		5.5s	40GB	22GB
	6000R	10	5	50h	3s	4.5GB	3.4GB
		50	10		3.5s	21GB	12GB
		100	20		4s	45GB	24GB
CoPhIR (106M - $m = 280$)	1000R	10	5	17.5h	4s	4GB	2.6GB
		50	10		5.5s	19GB	10.2GB
		100	20		7.2s	30GB	20GB
	2000R	10	5	26h	3.4s	4.2GB	3GB
		50	10		5s	20GB	12GB
		100	20		6.1s	40GB	21.8GB
	6000R	10	5	84h	2.9s	4.5GB	3.5GB
		50	10		4.1s	21GB	12.5GB
		100	20		5s	45GB	24.3GB

on n.

The size of the lists assigned to each bucket decreases as n increases, which leads to a decreasing running time (at the cost of a more rough approximation). The searching time, however, increases along with the number of nearest reference objects $\tilde{n}$.

The last two columns of Table 4.2 show the uncompressed and compressed memory usage. Using the delta compression technique [100], we are able to decrease around 50% of the memory usage. This percentage decreases when the number of reference objects increases. The main reason is that the distribution of the objects changes. When the number of reference objects increases, consecutive objects are not anymore close to each other (they are located in different lists), so that the compression technique is less effective.

For the two CoPhIR datasets, the sizes of the uncompressed memory are the same, because they have the same number of objects. On the other hand, after compressing, the compressed sizes are not similar. This happen because the reference objects have different distribution and the data have different

dimensions. This difference in distribution also affects the searching time.

In summary, increasing the number n of reference objects increases the memory usage, but decreases the searching time and improves the RE and PE. For the same number of objects N and references n and for different dimensions, the uncompressed memory size is the same, but the compressed memory size might change.

4.3.2 Comparative experiments

It is difficult to run exhaustive comparative tests with previous works since each proposal generally tests its implementation on a specific dataset, which is not always made available. And even if so, often not all parameters of the experiments are reported. In section 4.3.2, we perform a comparison study between MPT and the available data structures, based on permutation-based indexing. In section 4.3.2, we propose a comparison between MPT and other approximate similarity search techniques.

Comparative experiments with earlier permutation-based proposals

We compare MPT to earlier proposals (NAPP [70], Perms [66], PP-Index [69], MIF [61], MSA [3]).

In [70], the authors made a comparison based on the recall between their implementation and previous implementations using the first 10-million objects in the CoPhIR dataset using the first three descriptors of the MPEG-7 features (208-dimensional) and the Color histogram dataset, which we have used in section 4.3.1. We use the same datasets and compare our results with that summarized in [70]. For the 10-million CoPhIR dataset, we followed the same procedure. We used the first 200 objects as queries. For the Color histogram dataset, the 200 query objects were selected randomly. Figures 4.10(a) and 4.10(b) show the RE for the 30-KNN scenario using different numbers n of reference objects for the two datasets respectively. In [70], for the two datasets, the number of nearest reference objects is $\tilde{n} = 7$ for NAPP, PP-Index, Perms and MIF with at maximum 60'000 direct distance calculations. For the PP-Index, query expansion is not considered.

For the CoPhIR (10-Million) dataset (Figure 4.10(a)), we compare the results obtained in [70] to MPT using the same values for n and $\tilde{n}$ with $B = 3$ and $\Delta = 400$, so that K_c is 12'000 for the 30-NN scenario. MPT using $K_c = 12'000$ ($\Delta = 400$) was able to achieve a better performance than MIF [61] and PP-Index [69] and close to NAPP [70] when using $60'000$ DDC. Our MPT is then more efficient, thanks to the bucketing technique we propose. NAPP uses the full range of nearest reference objects and compares it to the full range of nearest reference objects of the query, leading to a high number of false positive objects. Since, in MPT, we care only about the shared buckets, we achieve a higher recall while saving calculations. The best searching time for NAPP is 0.3s, as opposed to 0.18s for MPT using 2048 reference objects.

Also, we conjecture that our technique is faster in terms of indexing. The indexing time is an offline process, but it is an important step. In [70], authors did not report the indexing time. For MPT, it takes

120 minutes to sequentially index the 10-million objects using 2048 references. For Perms [66], it gives a higher RE, but it is a slow technique. It was unachievable to perform experiments using more than 256 reference points.

For the Color histogram dataset (Figure 4.10(b)), we compare the results obtained in [70] to MPT using the same values for n and $\tilde{n}$ with $B = 7$ and $\Delta = \{40, 100\}$, so that K_c is between 1200 and 3000 for the 30-NN scenario. MPT using only $K_c = 1200$ ($\Delta = 40$) was able to achieve the same performance as NAPP [70] using 3000 DDC. The best searching time for NAPP is 0.0054s, opposed to 0.0031s for MPT using 2048 reference objects. In addition to what has already been mentioned, MPT is faster and more effective due to the bucketing technique.

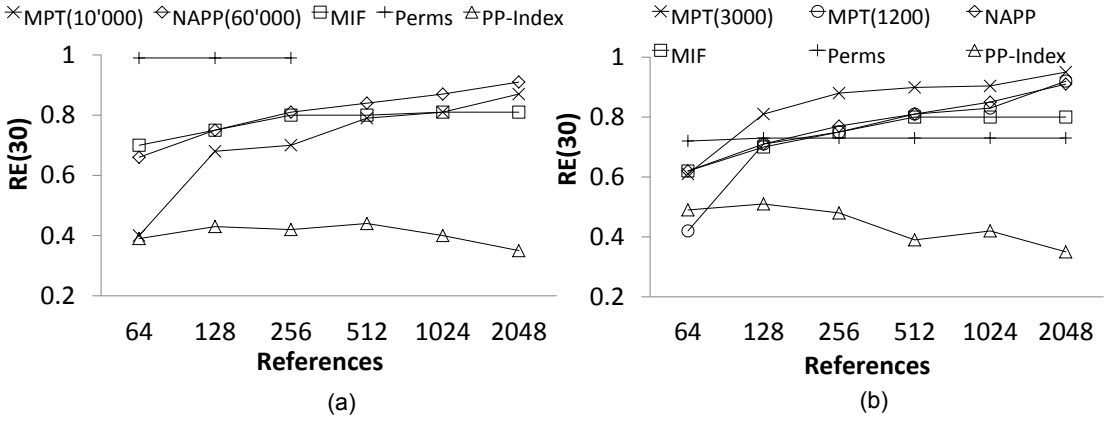


Figure 4.10: Recall comparison between MPT, MIF, NAPP, Perms and PP-Index for top 30-NN. The average recall values for these structures are taken from [70]. a) CoPhIR (our experiments are based on first 200 objects as queries) b) Color Histogram (our experiments are based on averaging 200 queries).

In [69], authors proposed a comparison between PP-Index and the MIF [61] using a smaller dataset available from the UCI knowledge Discovery in Database Archive [1]. The dataset consists of 50,000 HSV color histograms (32-dimensional vector) from an image database. We use the same dataset and compare our results to the results published in [69] (Figure 4.11a). MIF indexing used 50 out of a set of 100 reference objects. PP-index indexing used 50 references with a prefix length of 6 (experiments were made using queries with an expansion factor = 4). Our MPT indexing used 50 out of 100 nearest reference objects selected randomly from the dataset. With no query expansion, our MPT data structure outperforms the PP-Index (PP-Index-1 and PP-Index-4) and the MIF for different numbers of K-NN. Our proposal is also more effective in terms of indexing and searching time than PP-Index and MIF. MPT requires 2 seconds for indexing and 0.009s for searching, while PP-Index requires 4.9s for indexing and 0.01s and 0.02s for searching using 1 and 4 queries respectively. For MIF, it needs around 1 min for indexing.

[1] http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html

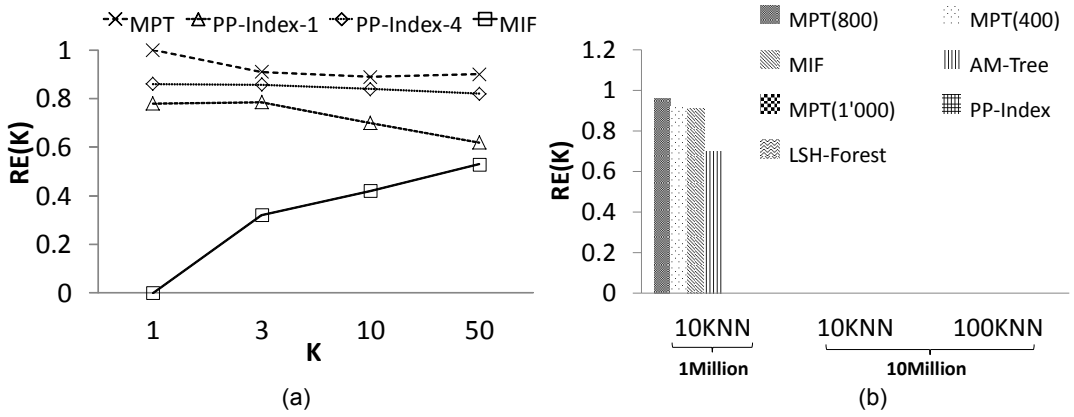


Figure 4.11: a) Recall comparison between MPT, MIF and PP-Index for 1, 3, 10 and 50 top K-NN. The Recall values for PP-Index and MIF are taken from [69] (our experiments are based on averaging 100 queries) b)(1-million) Recall comparison between MPT, MIF and AM-Tree using the first 1-million objects in the CoPhIR dataset. (10-million) Recall comparison between MPT, PP-Index and LSH-Forst using the first 10-million *Colour Structure* (64-dimensional) features from the CoPhIR dataset.

We finally compare our proposal to the MSA proposed in [3] using the same dataset, which is a part from dataset DS_4 detailed in chapter 2 (4'594'734 features (21-dimensional)). RE and PE were computed using the nearest $\tilde{n} = \{50, 250, 500, 750, 1000\}$ reference objects out of totals of $n = \{100, 500, 1000, 1500, 2000\}$, respectively. RE$= 0.7$ for 100-NN using 1000 nearest references and no DDC. Our proposed MPT achieves a higher recall value using less nearest reference objects. For only $\tilde{n} = 50$ out of $n = 1000$ reference objects, the recall value is 0.8 with $\Delta = 4$ ($K_c = 400$ objects have been compared to the query). Further, our data structure requires only 0.13s for the search while MSA needs 1.9s. The main saving is made in not using the SFD but an approximate version of its induced ranking Eq.(4.4) and (4.5). Memory saving is also significant with 490MB needed for MPT while MSA requires 17GB (due to the large number of reference objects needed to avoid DDC).

Comparative study against other approximate similarity search techniques

In [69], authors proposed another comparison between PP-Index and LSH-Forest [35] using the first 10-million *Colour Structure* (64-dimensional) features from the CoPhIR dataset. They proposed a distributed implementation of the PP-Index. Then, they propose a comparison based on the number of distributed indexes. Since, our data structure is not distributed yet, we compared our results to the "one index" [35] data structure. PP-index indexing used 100 references with a prefix length of 6 and $K_c = 1000$, where K_c is the number of candidate objects, where a direct distance calculation is calculated with them. The hash functions for the LSH-Forest were generated by random sampling from a family of hash function of the form $h(v) = \lfloor \frac{a.v+h}{e} \rfloor$, where v is the vector of the feature, a is a vector

of randomly sampled values from Cauchy distribution (which is 1-stable), $e = 5$ and h is randomly sampled with uniform distribution form $[0, e]$ [69, 35] (detailed in section 2.2). The maximum depth is $l = 6$. Hence, each function is composed by 6 hashing functions, where each function returns an integer number indexed by the nodes in the prefix tree. Our MPT indexing used $B = 3$ for $\tilde{n} = 6$ out of 100 references. The DDC factor is changing based on the K-NN to meet the same parameters in [69], for 10-NN, $\Delta = 100$ and for 100-NN, $\Delta = 10$.

Figure 4.11(b) (10-million dataset) shows that with the same parameters MPT is able to achieve 10% and 13% higher recall than the PP-Index and the LSH-Forest respectively for the 10-NN search scenario and 22% and 33% higher recall than PP-Index and the LSH-Forest respectively for the 100-NN search scenario. In addition, MPT is much faster in terms of indexing. For MPT, indexing the top 10-million object from the CoPhIR dataset with 100 references takes 10 minutes, while for PP-Index it takes around 20mins.

In terms of searching time and memory usage, other algorithm have slightly better performance, but MPT shows a better RE. The searching time for MPT is similar to the searching time for PP-Index. There is no significant difference between them. The compressed memory usage for MPT is 289MB, while, for PP-Index, it is 848KB. The difference in memory is due to saving the objects ids in the main memory for the MPT.

In [62], authors propose a comparison between the MIF and the AM-Tree[101] using the first 1 million objects from the CoPhIR dataset using the full MPEG-7 features (280-dimensional). In [62], the number of nearest references is 200 out of 2000 references, with $\Delta = 90$ for MIF. The AM-Tree proximity thresholds range is from 3×10^{-4} to 7.8×10^{-4}, with $\Delta = 90$. The experiments were done for the top 10-NN search scenario. For MPT, we used the same parameters, $n = 2'000$, $\tilde{n} = 200$, $\Delta = \{40, 80\}$ and $B = 10$. Figure 4.11(b) (1-million dataset) shows that with $\Delta = 40$, we are able to achieve the same performance as MIF and a better performance than AM-Tree at much lower cost (400 DDC compared to 900 DDC). With almost the same cost for the MPT (800 DDC), we are able to achieve 5% and 27% higher recall than MIF (900 DDC) and AM-Tree(900 DDC). The average RE was calculated using the first 500 objects in the dataset as queries.

4.4 Summary

In this chapter, we proposed two different data structures for permutation based indexing. The aim is to decrease the memory usage, improve the indexing and searching time and improve the efficiency of the search.

The Metric Suffix Array (MSA) stores the reference information, the object id, the location of the reference point in the ordered list of the object in 4 bytes. The MSA follows the same principles as the suffix arrays. The main idea is to concatenate the ordered lists for all the database objects to construct

one string. Afterwards, the MSA can be built for this string. In order to improve the search performance, the MSA is divided into buckets and each bucket into number of sub-buckets. The MSA proves to be efficient in searching, recall and position error. However, the indexing time is long compared to other data structures. In addition, MSA does not scale well due to the limitation in memory. The 4 bytes can only be used if the string size is $< 2^{32}$. When the string size increases, 8 bytes would be needed.

Then, we proposed the Metric Permutation Table (MPT) data structure. That aims to improve the indexing time and the memory size, while preserving good precision and good searching time. The MPT is based on the quantization the ordered lists. The helps to decrease the size of the ordered list, improves the memory usage and improves the indexing time. Its structure is similar to the inverted files with some compression. The MPT proved to be efficient in terms of indexing, searching and precision. We show empirically the effect of varying the main parameters of our data structure and discussed their inter-relationships. In summary, increasing the number of reference points decreases the searching time and improves the recall and the position error. However, it increases the memory usage. For the same number of objects and reference points and for different number of dimensions, the memory usage is the same, but the Recall and the Position Error change.

The evaluation is performed using standard and large-scale datasets of millions of objects. We also demonstrate empirically the validity of our proposals by comparing their performance with that of state-of-the-art methods.

Chapter 5

Parallel Computing

Nowadays, the volume of available data is increasing exponentially and has reached to the petabyte-scale in many domains such as information retrieval, astronomy and bioinformatics. This big data needs to be indexed, stored, analysed and visualized to allow easy access and extraction of information to benefit to a large audience. Current sequential algorithms cannot handle such volumes. Hence, the role of parallel and distributed computing in all of its forms becomes more important than ever before by offering large-scale processing capabilities.

In this chapter, we first give a brief introduction about parallel computing and its different classes. In sections 5.2 and 5.3, we show the different parallel computing memory architectures and the related programming models for each architecture. In section 5.4, we propose the MRO-MPI as an enhanced programming model for distributed computing based on MPI and MapReduce. In section 5.5, we evaluate our model compared to the other distributed programming models.

5.1 Introduction

Parallel computing aims to perform many operations simultaneously by dividing the problem or the data on multiple processors in order to obtain results faster. There are different classifications for parallel computing [102, 103, 104]. The most widely used classification is the Flynn's taxonomy [104]. What makes the Flynn's taxonomy popular is that it is not based on the machine architecture, it is based on *instruction* and *data* streams.

Flynn's taxonomy consists of four classes based on the multiplicity of instruction streams and data streams (Figure 5.1). The first class is *Single Instruction Single Data (SISD)*. This represents the sequential computers (no parallelism). A program is transferred to a stream of instructions and these instructions are executed over a stream of data sequentially. The second class is *Single Instruction and Multiple Data (SIMD)*. There is only one instruction and this instruction is executed in parallel by different processors over different data streams. There is no interaction between the instructions. Hence, SIMD provides the

simplest way of parallelization, which is known as *Data Parallelization* or *embarrassing parallelization* [105]. The third class is *Multiple Instruction and Single Data (MISD)*. This is similar to the pipe-lining architecture. There is a single data stream and different operations are performed on this data stream. Accordingly, some instructions might have to wait for other processors to finish their work. The fourth class is *Multiple Instruction and Multiple Data stream (MIMD)*. In this class, multiple instructions are performed on multiple different data streams in parallel. The *MIMD* is more scalable than the other representations. Recently, a new term appeared; *Single Instruction Multiple Thread (SIMT)*. SIMT is similar to SIMD and is used for Graphics processing units (GPUs) and stream processors [106] (more details in section 5.3.2).

Instruction\Data	Single	Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

Figure 5.1: Flynn's taxonomy

5.2 Parallel Computing memory architecture

In terms of memory architecture, parallel computing is classified in three models:

- Shared Memory Model,
- Distributed Memory Model,
- Hybrid Memory Model,

which we review in details in the next sections.

5.2.1 Shared Memory Model

In the shared memory model, a node (computing machine) is composed of a number of processors, which have access to the same memory (Figure 5.2). Any change that happens due to any processor at a memory location affects the other processors.

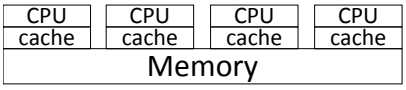


Figure 5.2: Shared Memory Model: four processors located in the same node, each processor has a cache and accesses the same memory.

There are two advantages to this model. First, this creates a fast sharing of the data between the processors, because they share the same address space. The second advantage is its physical availability.

Most of current systems are multicore including mobile phones. However, this model is not scalable. Processors cannot easily be added to the system.

The most common programming model for this architecture is the threading and multi-processing, which are supported through Pthreads [107], OpenMP [108] APIs, respectively.

5.2.2 Distributed Memory Model

In this model, there are multiple nodes. Each node is composed of at least one processor, some memory, and a network card (Figure 5.3).

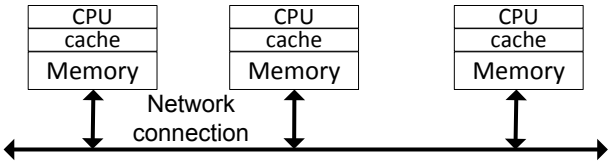


Figure 5.3: Distributed Memory Model: three nodes, each node has its own memory and the data exchange is done through the network connection.

The nodes are connected together through a network backbone. A processor within a node has only access to its local memory. Hence, unlike in the shared memory model, the changes that are made in a local memory location have no effect on the memory of other nodes.

The main advantages of this model are the scalability and cost effectiveness. It is scalable because more nodes can be added easily to the system. It is cost effective because a powerful system can be built using commodity, off-the-shelf processors and networking. The disadvantages of this model are the relative difficulty in programming and debugging. Moreover, network communication between the nodes affects the performance. Hence, adding more nodes does not always give the best solution.

The main architecture for this model is the *computer cluster*. A computer cluster is a group of independent computers, inter-connected and that work as one unified system. These computing nodes work together throughout a software (i.e: parallel libraries and management tools) and network connection. The most common programming models are the Message Passing Interface (MPI) [109] and MapReduce [110] (see section 5.3).

5.2.3 Hybrid Memory Model

This model is a combination of the previous two models (Figure 5.4).

It includes multiple nodes and each node has multiple processors. The nodes are connected through a network connection. Changes that happen in local memories do not affect the other nodes, but affect the processors within the node. The main advantage of this model is the full usage of all the available resources. However, the programming and debugging complexity are increased.

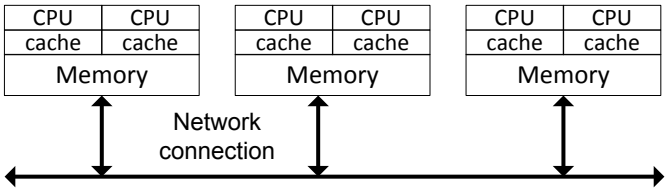


Figure 5.4: Hybrid Memory Model: three nodes, where each one has its own memory and two processors, exchanging data between the nodes is done through network connection.

5.3 Parallel programming models

5.3.1 Open Multi-Processing (OpenMP)

OpenMP [108] is an API that supports shared memory architectures in C, C++ and Fortran on different platforms (*Linux*, *Unix* and *Windows*). OpenMP is based on the *Fork-Join* model. All applications that are written with openMP start with a single thread which is called the master (main) thread. When the master thread enters a parallel region, it starts to create a team of parallel threads that run in parallel (*Fork*). At the end of the parallel region, the threads terminate, leaving only the master thread (*Join*). Figure 5.5 shows the *Fork-Join* model. OpenMP is suitable for SIMD, MISD and MIMD models.

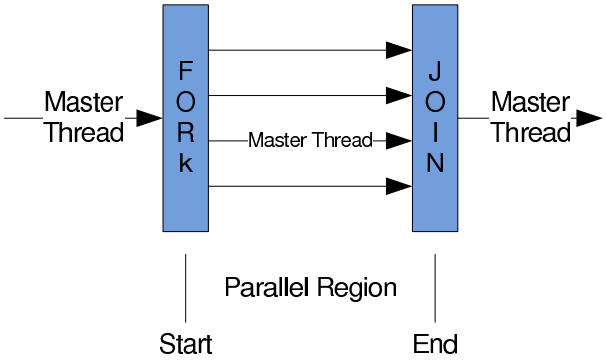


Figure 5.5: In Fork-Join model, the main thread create a team of parallel threads that run simultaneously.

5.3.2 Compute Unified Device Architecture (CUDA)

CUDA programming model is a heterogeneous model in which the CPU and the Graphics Processing Units (GPUs) can work together [111]. GPUs were initially dedicated to handle graphics primitives. With the rapid evolution of the NVIDIA CUDA API, people from different communities got the access to the GPU parallel architecture for powerful computation.

A *device* is the GPU card and a *host* is the computer that hosts the GPU. The GPU architecture contains a large number computing cores with limited specification, which can handle large numbers

of operations at the same time. A kernel (function) launches thousands of threads for parallel execution organized in two levels: grids and blocks. The grids are two or three-dimensional arrangements of blocks, where every block consists of an upper limit of threads (512 or 1024). Each group of threads is called a wrap [111]. Threads have access to some built-in variables, which are used for thread indexing. GPU also has an efficient memory hierarchy divided in global and local memory. A main difference between them is that accessing the local memory is much faster than accessing the global memory. On the other hand, the local memory size (kilo bytes) is smaller than the global memory (Giga bytes). In order to get a good performance from the GPU, the data that needs to be computed should be small enough to fit in the registers, otherwise all the time is consumed in fetching the data from the global memory. Accordingly, algorithms like sorting are not effective on the GPU, as the data that needs to be sorted cannot be fitted at once in the local memory. The GPU architecture and the CUDA programming model follow the *Single Instruction Multiple Thread (SIMT)* model.

5.3.3 Message Passing Interface (MPI)

The Message-Passing Interface (MPI) [109] is a standardized and portable message-passing system that is available on *Windows*, *Linux* and *Unix*. A program written with MPI runs on all machines concurrently as a separate processes, where each process has a unique *rank*. Variables within a process are private. Within the program, the programmer has to define the work done by each process and how the processes communicate with each other. MPI supports point-to-point, one-to-all, all-to-one and all-to-all communications. Table 5.1 gives examples of commonly used MPI functions. Figure 5.6 illustrates the message passing paradigm. MPI was targeted for distributed memory systems, but MPI supports virtually other programming models; shared, and hybrid models. Accordingly, MPI is suitable for SIMD, MISD and MIMD models.

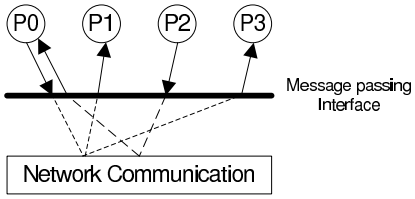


Figure 5.6: Message passing paradigm: MPI is responsible for transferring messages between the machines, these machines are connected through a network.

5.3.4 Map-Reduce

MapReduce is a programming model for data intensive applications. MapReduce was proposed in [110]. It is mainly composed of two functions: a *map* function and a *reduce* function. In general, the input data is divided into small chunks and randomly passed to a group of *map* functions. The *map* functions

Table 5.1: Example of the most common MPI functions

MPI Method	Description
MPI_Send()	Send data directly to a certain process
MPI_Recv()	Receive data from a certain process
MPI_Gather()	Gather data from different precesses
MPI_Scatter()	Scatter data on the processes
MPI_Bcast()	Broadcast data on all processes
MPI_Barrier()	Process synchronization

process the input data and generate intermediate *(key,value)* pairs. These pairs are grouped together with respect to their *key* to produce *(key,list(values))* tuples. The tuples are then passed to a group of *reduce* functions, which do some analysis.

Many applications can be adapted to follow this workflow, including inverted indexing [112], k-means [113], sorting and PageRanking [110]. Hence, MapReduce simplifies the parallel programming process through the two *map* and *reduce* functions. The most successful implementation of MapReduce is Hadoop [114].

The execution of the MapReduce model is done through four stages, as follows (see also Figure 5.7):

1. **Mapping:** A user-defined map function M starts simultaneously in parallel on different machines. Each Map function is responsible for a chunk of the input data $(K_r \times V_r)$. The map functions process this input data and emit intermediate *(key,value)* pairs : $M : (K_r \times V_r) \mapsto (K_m \times V_m)$. These intermediate pairs are saved locally. A master machine monitors all the mapping functions. When it receives a termination signal from all the mappers, it starts to assign reducers tasks by defining a key, or range of keys, for each reducer to process them.

2. **Shuffling:** Reducers communicate with the mappers through remote procedure calls to gather all the intermediate (K_m, V_m) pairs based on the key K_m, or within the range of keys assigned to them. This process requires all-to-all communication as the intermediate keys K_m are distributed on different machines.

3. **Merging:** Each reducer then starts to merge the pairs, based on identical intermediate keys to produce $(K_m, list(V_m))$ tuples.

4. **Reducing:** These tuples are passed to a user-defined reducing function R one by one to emit the output O, which is distributed on different machines: $R : (K_m \times list(V_m)) \mapsto O$.

Thus, the MapReduce framework brings a simple and powerful interface for data parallelization by keeping the user away from the communications and the exchange of data, as this is directly handled by the framework. The user only needs to write the map M and the reduce R functions. Hence, the MapReduce programming model is only suitable for SIMD.

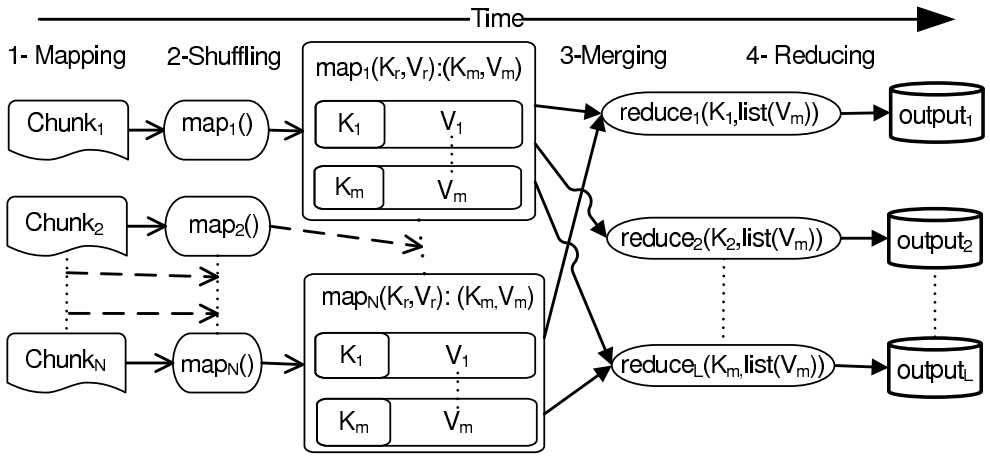


Figure 5.7: MapReduce framework is composed of 4 phases: 1) Mapping, 2) Shuffling, 3) Merging, and 4) Reducing.

MapReduce Complexity

The theoretical memory and computational complexities of MapReduce applications are hard to evaluate, because of the structure of the MapReduce framework. The complexity for the algorithms based on MapReduce depends on:

- the number of input pairs to output by a Mapper/Reducer,
- the maximum running time for a Mapper/Reducer,
- the maximum memory used by a Mapper/Reducer to process the input pairs.

Hence, in all of our experiments, the complexity will be discussed empirically with the results.

5.4 MapReduce overlapping using MPI (MRO-MPI)

Currently, most parallel applications depend on two types of parallelization: data parallelization and algorithmic parallelization [115, 105]. It seems sensible to build a library supporting the two types of parallelization (MPI-MapReduce). In the same library, we can combine the advantages of simplicity of the MapReduce programming model with the efficient communication capabilities offered by MPI to build a framework for fast data and algorithms processing. We propose the MRO-MPI model (MapReduce Overlapping using MPI) as a strategy to speed up the MapReduce model by avoiding its bottlenecks and using MPI functions.

The idea of implementing MapReduce using MPI has already been studied. Hoefler et al. [116] made the first attempt to write MapReduce using MPI. They provided strategies for implementing MapReduce

using blocking, non blocking and collective operations based on the original MapReduce model. They also built it based on MPI-2.2 and MPI-3 features to improve MapReduce. Steven et al. [117] released the first public library for MapReduce using MPI. Their architecture is exactly similar to the original MapReduce. Xiaoyi et al. [118] gave a good analysis on the Hadoop communication and implemented the original MapReduce model using MPI by providing some ideas on the overlapping between the mapping and the communication phases. The results based on the word count application showed that a good speed improvement comparing to Hadoop can be obtained based on MPI communications. Our work is orthogonal to the previous work. The main difference between our model and the earlier MPI-MapReduce model [117] lies in the idea of overlapping using MPI. In our proposals, the *map* and the *reduce* functions work concurrently, while for [117, 116] the *reduce* function cannot start before the termination of the *map* functions.

5.4.1 MapReduce and MPI bottlenecks

The MapReduce model has at least three bottlenecks:

- **Dependence:** reducers cannot start before the mappers are done, which affects the total performance of the system. If one of the mapping functions is slower than the others, all the reducers have to wait.

- **Disk access (multiple read/write):** writing the intermediate (K_m, V_m) pairs during mapping, and reading these pairs again during reducing, influences the performance, especially if the emitting is done at a high rate.

- **All-to-All communication:** excessive communication between all the machines needs to be done during the shuffling phase to create the $(K_m, list(V_m))$ tuples. The running time of this phase depends on the number of intermediate pairs.

At the same time, MPI is missing the simplicity; it is not easy to handle. The mutation of a sequential algorithm using MPI requires reconstruction of the algorithm. Also, sending many small chunks affects the performance, which sometimes becomes slower than a sequential implementation, because of the network latency and bandwidth limitations.

The current implementations of MapReduce using MPI follow the original four steps; mapping, shuffling, merging and reducing [117, 116, 118]. As they have the same structure, the three above bottlenecks still exist. In our MRO-MPI model, we overcome these main bottlenecks to get better performance and, at the same time, we maintain the usability and the simplicity of MapReduce by keeping the user away from MPI complex functions.

5.4.2 MRO-MPI Model

The main idea behind our model is to overlap the mapping and the reducing phases by sending partial intermediate data. Simply, when available, each mapper sends partial intermediate (K_m, V_m) pairs to the responsible reducers. The reducers then work on this partial data and wait for more data, until all the mappers are done. With this model, we rule out the multiple read/write. As the mapper continuously sends the partial data directly to the reducers, there is no need to save the intermediate data locally. Hence, the shuffling phase is merged with the mapping phase instead of being a separate step. The main gain is in the simultaneous execution of the *map* and the *reduce* functions. Unlike in the original model [110, 117], the reducers do not wait for the mappers to finish their work. This diminishes the running time and gives a good speedup, as demonstrated in section 5.5.1. Figure 5.8 illustrates our idea of overlapping.

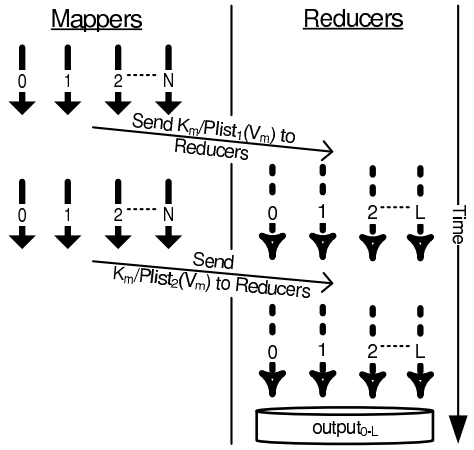


Figure 5.8: MRO-MPI: The mappers and reducers work in parallel and partial data is sent in a pipeline fashion.

5.4.3 Technical implementation

In MPI, the sender has to define the *rank* of the receiver process and the type and the size of the sent data. At the same time, the receiver has to be ready and informed about the received data. Thus, this decreases the usability of MapReduce using MPI if we leave this complexity to the user. In our prototype, we keep the user away from the MPI complexities. Figure 5.9 shows the architecture of our prototype. The Map and Reduce sides are divided into two parts; user and system. The user side is the part where the programmer writes the mapping function. The system side is the part responsible to handle the communication and data merging. Our model is based on three steps as follows:

- **Mapping and Shuffling:** In our model, mapping and shuffling are merged into one phase. The mapping is exactly similar to the map function in the original MapReduce model [110]. It emits

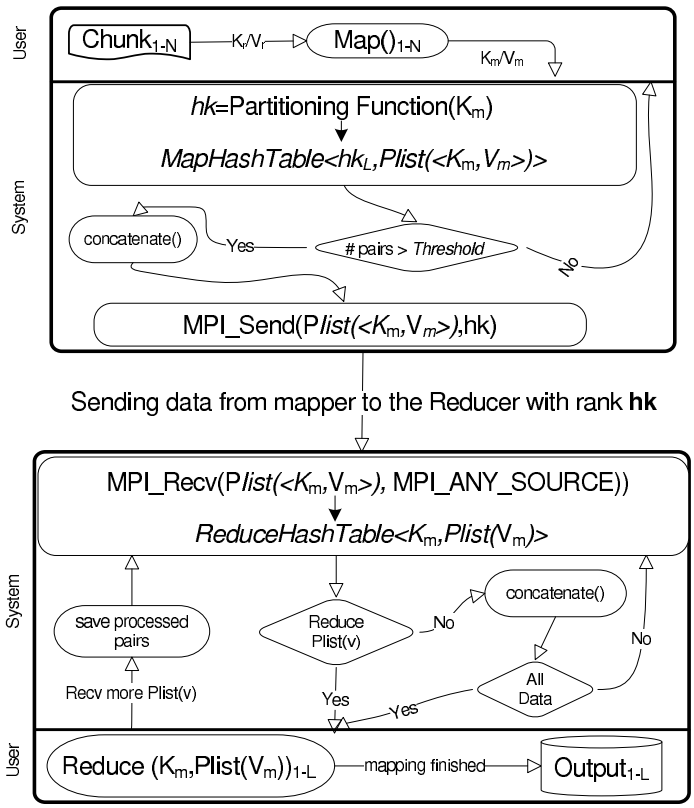


Figure 5.9: MRO-MPI: Technical details

(K_m, V_m) pairs. The K_m, of each pair is passed to a partitioning function :

$$Partitioning : (K_m) \mapsto (hk_l)$$

The output range of the function is based on the number of the reducers; for L reducers the range is [1,L]. The default function is a *"Hash"* function, which is similar to *hashPartitioner* in Hadoop, but also can be replaced by any user-defined hash function. For the default hash function, hash collisions occur as the number of keys is larger than the number of the reducers, but it happens with equal distribution to create a rough equal load balancing of the keys on the reducers. The $(hk_l, (K_m, V_m))$ pairs are saved in a local hash table for each process. Each hk_l is associated with a list of various (K_m, V_m) pairs:

$$MapHashtable : (hk_l) \rightarrow ((K_i, V_1), (K_i, V_2), (K_{i+1}, V_1), \ldots (K_m, V_m)).$$

There are l counters $C_1 \ldots C_l$, each counter is assigned to a hash key hk_l. They are used to count the number of pairs associated to each hk_l. Every time a new pair is emitted, counter C_{hk_l} is checked if it is greater than a *threshold* value T, which is a user-defined buffer size. If so, the

partial data is concatenated as one chunk and sent directly to the responsible reducer, which has a rank hk_l, otherwise the data is added to the list.

Sending data in MPI is based on the MPI data types like MPI_INT, MPI_CHAR,... etc. Additionally, MPI gives the user the ability to construct specific data types based on the original ones, which is called "derived data types" [119]. In our prototype, we have tested concatenating the pairs into one *CHAR* array against defining our MPI derived data type of a simple structure consists of a *CHAR* array as a word and an *INT* as a value. We found that good performance is achieved when the data is sent as a *CHAR* array (see section section 5.5.1 for details). Hence, we combine all the data as one *CHAR* array and send it to the responsible reducer.

- **Receiving and Merging:** All reducers are actors, they are ready to receive data from any mapper. The received data contains multiple intermediate (K_m, V_m) pairs. A hash table is responsible for organizing this data. The key of this hash table is the intermediate key and the value is a list of intermediate values received in this partial list and related to this key:

$$ReduceHashTable : K_m \rightarrow Plist(V_m).$$

- **Reducing:** After grouping the data, the system checks if the user asked for reducing partial data or not, as in some applications, the reduce function needs the full $(K_m, list(V_m))$ (e.g. average functions). In this case, the reduce function receives the data and merges it with the already saved data without any processing until the complete mapping is done. If the partial reduce can be processed (e.g. sum functions), all the keys within its partial list are passed to the reduce function one by one. After reducing, the partial data is saved in local memory. For the two scenarios, after saving the data, the system calls the receiving function again and this process continues until all the data is received from the mappers. When mapping is done and the last partial data is reduced, output data are saved on the local hard disk of each reduce process.

In our model, we have reorganized the original four phases into three phases. The three phases run in parallel on different machines and continue until the mapping is done. The user has to define the number of mappers, reducers, and the threshold value T. The ratio between the mappers and the reducers affects the performance of the model. A good ratio between the mappers and the reducers with analysis of the effect of changing the T value are given in section 5.5.

5.4.4 Scheduling

In Hadoop, the number of map functions depends on the number of input chunks. Each slot is responsible to handle a number of map functions. The default number of map functions per slot depends on the user

configuration. To achieve maximum performance, Hadoop uses a greedy task scheduling by assigning more mappers to the slots which do not run reducer tasks. There is therefore an overhead by the system for map functions scheduling. In MRO-MPI, we schedule the chunks in a different and a simple way to override the scheduling overhead. In our model, the number of mappers does not depend on the number of chunks. The number of mappers is fixed and only depends on the number of the mapping processors, which is defined by the user before starting the MPI job. Hence, the input chunks are divided over these mapping functions. Thus, one map function handles more than one chunk and there is no more need for the scheduling overhead. For instance, in our model, if we have 50 chunks and 10 processes for mapping, we will have 10 map functions and each map function will handle 5 chunks. Using Hadoop, there would be 50 map functions handled by 10 processes.

5.5 Evaluation

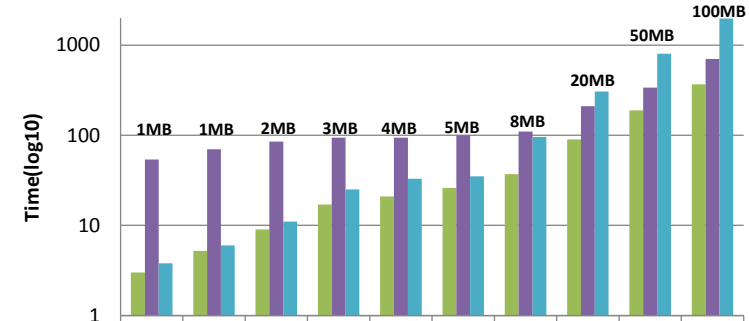
We have conducted large-scale experiments to test the validity of our model based on two different applications. In section 5.5.1, the *WordCount* application [110] (classically used as a benchmark for MapReduce) is tested using our model MRO-MPI and compared to MR-MPI [117] and Hadoop. In section 5.5.2, an inverted index was implemented using MRO-MPI for a 9.3-million text (XML) collection of size 36GB, related to 9.3-million images from the 12-million ImageNet corpus [91]. Our performance is compared to Hadoop. The experiments were conducted on our *Coral Cluster*. The cluster specification is detailed in appendix A.1.

5.5.1 Word Count

WordCount [110] simply counts the occurrence of words in different documents. The map function emits $(word, 1)$ pairs, where $word$ is the key and 1 represents the value. The input data size varies from 0.2GB to 53GB from project Gutenberg [120]. For this example, we use 48 cores, 24 as mappers and 24 as reducers for our model. The threshold value T is empirically set to 10000. For Steven et al.'s implementation [117], the 48 cores are used as mappers then as reducers, as there is no overlapping like in our model. For Hadoop, we set the number of reducers to 48 and the number of mappers varies from 200 to 650 according to the number of partial input files. The data replication factor is 10.

Figures 5.10 and 5.11 show the $\log_{10}$ of the running time and the speedup for the three implementations respectively. Hence, Speedup is defined as:

$$Speedup = \frac{T_{MR}}{T_{MRO}}, \tag{5.1}$$



Data size(GB)	0.2	0.5	1	2	2.5	3	5	12	27	53
MRO-MPI	3	5.2	9	17	21	26	37	90	189	369
Hadoop	54	70	85	94	94	100	110	210	339	704
MR-MPI	3.8	6	11	25	33	35	96	306	805	1980

Figure 5.10: WordCount example: The x-axis shows the data size in gigabytes. The y-axis shows the log_{10} of the running time. The value in the table under the figure shows the running time in seconds. The values above the columns shows the size of each partial file in the data set. As we can see, MRO-MPI outperforms MR-MPI [117] and Hadoop

where T_{MR} is the Hadoop or MR-MPI execution time and T_{MRO} is our MRO-MPI execution time. As we can see from the figures, our MRO-MPI model and MR-MPI model [117] achieve high speedup compared to Hadoop when the data size is less than 5GB. For data with size more than 5GB, Hadoop becomes faster than MR-MPI, but not faster than our model. The reason is that both of Hadoop and MR-MPI follow the same model, but Hadoop has its own file system, which is based on moving the computation power instead of moving the data in contrast to MR-MPI. For our model, we achieve high speedup compared to Hadoop and MR-MPI because of the full overlapping between the mappers and the reducers. For example, for 27GB, our model is 1.7 times faster than Hadoop and 4.2 faster than MR-MPI.

Figure 5.12 shows the effect of changing the threshold value T. The T value is the sent chunk size in bytes. Normally, using MPI, sending small chunks of data between processes increases the running time, because of the network latency. The same effect happens when we send large chunks through the network, because of the bandwidth limitation. So, for different data sizes, we have tested the effect of changing the T value. Based on our network connection (1GB/s), we found that the best processing time is achieved when the chunk type is *CHAR* array and with size in the range of 0.0095MB to 4MB. Figure 5.13 shows the effect of choosing MPI_CHAR against a simple structure that consists of a *key* of type *CHAR* array of size 100 and a *value* of type *INT*. As we can see, the structure degrades the performance of the model for the same data sets with the same configurations. The reason for that is the overhead of the MPI Derived Datatype [121, 122, 123].

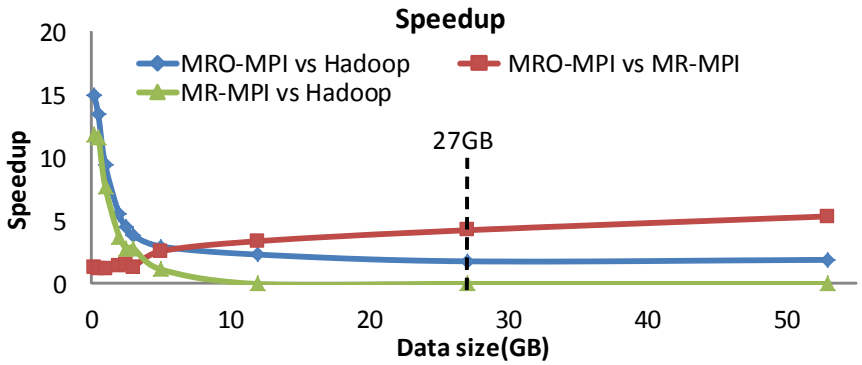


Figure 5.11: Speedup based on different data sizes. The x-axis shows the data size in GB. The y-axis shows the speedup

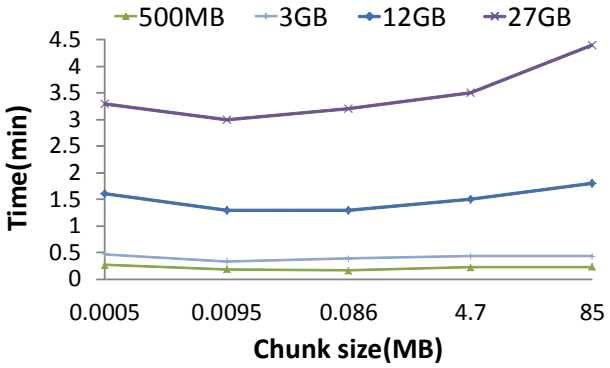


Figure 5.12: Effect of changing T. The x-axis shows the chunk size in megabytes. The y-axis shows the running time in minutes.

5.5.2 Distributed Inverted Indexing

Building inverted files (discussed in section 2.4.1) is a computationally intensive problem and needs to be done from time to time to update the databases. In order to preserve acceptable processing times, indexing should be done in parallel. For validating our model, we used MRO-MPI to construct distributed inverted files for the 9.3-million ImageNet XML collection. Figure 5.14 shows an example of the XML data used.

MapReduce for Distributed inverted files

In our implementation, the mappers are responsible to read and to tokenize the terms from every document. Mapper nodes emit *(key,value)* pairs. The *key* is the term extracted from the file and the *value* is the document name and the corresponding tf for the term:

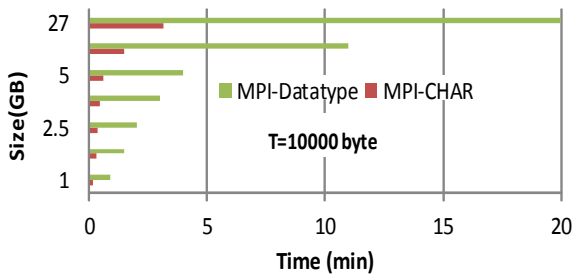


Figure 5.13: Effect of changing of choosing MPI_CHAR against MPI derived data types. The x-axis shows the running time in minutes and the y-axis shows the data size in gigabytes.

```
<?xml version="1.0" encoding="utf-8"?>
<image name="n00015388_30042.JPEG" synsetid="n00015388" synsetnum="100015388">
        <url>http://farm4.static.flickr.com/3176/2433586904_180fae9f14.jpg</url>
        <description>animal % animate being % beast % brute % creature % fauna @ a living organism
characterized by voluntary movement</description>
        <keywords>animal % animate being % beast % brute % creature % fauna</keywords>
        <definition>a living organism characterized by voluntary movement</definition>
        <fulldescription>animal % animate being % beast % brute % creature % fauna @ a living organism
characterized by voluntary movement</fulldescription>
        <fullkeywords>animal % animate being % beast % brute % creature % fauna</fullkeywords>
        <fulldefinition>a living organism characterized by voluntary movement</fulldefinition>
</image>
```

Figure 5.14: XML for one of the images used in the indexing

$$(K_m, V_m) = (term, (documentname, tf)).$$

The *(key,value)* pairs are sent to the reducing nodes. The *partitioning function* distributes the data based on their lexicographic order. Each reducer is responsible for a certain range of terms. For example, if we have 13 reducers, reducer 1 is responsible for all the terms start with character ”a” and ”b” , reducer 2 for all terms starting with ”c” and ”d”, and so on. Of course, this distribution could be adapted more finely to guarantee load balancing. Each reducing node only receives the terms located within its lexicographic range. As the same terms from all documents are saved into the same database, reducer nodes can calculate the correct value of the IDF and then assign a weight to every term according to the TF-IDF scheme.

In this application, the reducers do not work on the partial data as before. Instead, they concatenate this partial data until the mapper is done and then calculate the TF-IDF, which is not time consuming as the data is grouped and sorted.

Experiments

Table 5.2 shows the running time in minutes. The head row shows the number of the reducers: 4, 13 and 26. The first column shows the number of mappers: 4, 10, 13, 20, 22, and 44. The value 0 for mappers and reducers corresponds to the sequential time, which was obtained on a machine holding 2.7 GHZ

processor with 32GB of memory. The running time using Hadoop was 40 minutes using 93961 mappers, 26 reducers and a replication factor of 20 on the same cluster. Compared to our results, which were obtained using 22 mappers and 26 reducers, our implementation was faster due to the partial transfer of data, done during the mapping process.

Table 5.2: Running time for indexing in minutes with respect to different number of mappers and reducers. For example, the value 14.2 is the running time for 22 mappers and 26 reducers. Sequential time is represented for 0 mappers and reducers.

Map/Reduce	**0**	**4**	**13**	**26**
0	81.6	-	-	-
4	-	39.2	46.6	51.7
10	-	41.5	39.8	39.2
13	-	66	28.8	24
20	-	111.9	19.6	18.5
22	-	113.9	16.45	**14.2**
44	-	118.5	125.2	45.8

MapReduce ratio From the reducer side, for 13 and 26 reducers, and 4, 10, 13, 20 and 22 mappers, the running time decreases when the number of mappers increases. This is expected as the mappers do most of the work, and more mappers means that the workload is divided, which helps to improve the performance. But when the number of mappers is high compared to the number of reducers, the running time increases. The reason is that the reducers are not able to handle all the received data in an efficient way. For example, that happens for 4 reducers and 13, 20, 22, and 44 mappers. The rate of emitting the (key,value) pairs is much higher than the rate of receiving, due to lower number of reducing nodes. From the mapper side, for 10, 13, 20 and 22 mappers with respect to 4, 13 and 26 reducers, the running time decreases when the number of reducers increases. More reducers help decreasing the communication load, which helps to improve the performance. This is not the case for 4 mappers, the running time increases when the number of reducers is larger than the double of mappers. The reason is the large number of reducers compared to the number of mappers. Mappers need to communicate with a high range of reducers with high load of data, which increases the communication time and affects the total performance.

For 44 mappers and 13 and 26 reducers, there is not such speedup, although the ratio between the mappers and the reducers is reasonable. The reason is cluster overloading. Our cluster is composed of 48 cores and the number of processes are 57 and 70, which means that the numbers of processes running is much higher than the number of actual cores and this affects the performance of the model.

Based on our experiments, we found that when the number of mappers and reducers increases, the running time systematically decreases, with a non-linear decay, but we should care about the ratio be-

tween them and we should not overload the cores with more than one process. The best ratio between the mappers and reducers is found to be:

$$2M \geq R \geq M,$$

where M is the number of mappers and R is the number of reducers. This ratio does not depend on the application. In the next application, we preserve this ratio.

5.6 Summary

Parallel computing is the process of dividing a problem into sub-problems by solving them separately in parallel to gain a speedup. Based on memory architecture, there are three architectures; shared memory (CPU and GPU), distributed memory (Computer Clusters) and hybrid memory model. Due to the limitation of the memory size, the distributed and the hybrid models provide more scalability. The Message Passing Interface (MPI) and the MapReduce programming models are the most common programming models for the distributed memory architecture.

We proposed a new way of handling the MapReduce programming model using MPI for fast large scale data processing. We have proposed the idea of the overlap between the map and reduce functions using MPI "MRO-MPI", which speeds up the process with two extra parameters. Our first parameter is the ratio between the number of mappers and the number of reducers. The second parameter is the threshold value T for the sent data.

The main advantages of our model are:

1. Maintain the simplicity of the MapReduce model.
2. Speedup: with the same number of nodes and less number of mappers and reducers, we achieve high speedup comparing to other implementations of MapReduce.
3. No dependency: we remove the dependency between the two functions. The reducers do not have to wait until the mappers are done.

To evaluate our model, we have tested it on two different applications. The first one is the *WordCount* example. Using our model, we achieved a high speedup comparing to Hadoop and the earlier implementation of MapReduce-MPI. For example, for 53GB, our model is 2.8 times faster than Hadoop and 5.3 faster than MR-MPI. Also, we have discussed the threshold value T and the derivative data types against normal MPI data types and how these can affect the running time of our model.

In the second application, we have indexed text data using our model against Hadoop. From the results, our model is faster than Hadoop. Also, we have discussed the ratio of mappers to reducers and how this can affect the running time.

Chapter 6

Multi-Core (CPU and GPU) For Permutation-Based Indexing

As was mentioned in chapter 4, several data structures have been proposed to handle the permutation lists [1, 61, 90, 70, 66]. These data structures manage to answer users queries in fast and effective way. However, the indexing process consumes a lot of time, which makes the algorithm not effective for large-scale data (aka Big-Data). To handle this big data, parallel strategies and platforms are needed.

In this chapter, we propose multi-core implementations of the permutation-based indexing algorithm on CPU and GPU. Our parallel indexing proposals can be adapted easily to work for any of the available data structures for permutation-based indexing [1, 61, 90, 70, 66]. Also, we propose parallel searching algorithm on CPU and GPU for the MPT [1], which was discussed in details in chapter 4.

The chapter is organized as follows. In the next section, we recall the basics of permutation-based indexing. In section 6.2, we propose a multi-core implementation of the indexing algorithm on GPU. In section 6.3, we propose a multi-core implementation of the indexing algorithm on CPU. Section 6.4 evaluates the proposed parallel indexing algorithms. In section 6.5, we propose parallel searching algorithms on CPU and GPU for the MPT data structure.

6.1 Permutation-Based Indexing Outline

The main weakness of permutation-based indexing is the indexing time, especially when the number of references and objects increases. Algorithm 6.1 details the basic indexing process which is used by all permutation-based indexing data structures. For each object in $o_i \in D$, the distance $d(o_i, r_j)$ with all reference points in the reference set R is calculated (lines 1-4). After sorting the distance values in increasing order using the suitable sorting algorithm (*QuickSort* is used), the full ordered list $L(o_i, R)$ is created (Line 5). In line 6, partial lists $\tilde{L}(o_i, R)$ are generated by choosing the top $\tilde{n}$ references from $L(o_i, R)$. In line 7, $\tilde{L}(o_i, R)$ are stored in the appropriate data structure [1, 61, 90, 70, 66]. Theoretically,

the sorting complexity is $O(nlogn)$ which leads to $O(N(n + nlogn))$ indexing complexity.

Algorithm 6.1 Indexing Algorithm

IN: D of size N, R of n and $\tilde{n} \leq n$
OUT: $\tilde{L}(o_i, R) \forall i = 1, \ldots, N-1$
1. For $o_i \in D$
2. For $r_j \in R$
3. $b[j].dis = d(o_i, r_j)$
4. $b[j].indx = j$
5. $L(o_i, R) = quicksort(b, n)$
6. $\tilde{L}(o_i, R) = partiallist(L(o_i, R), \tilde{n})$
7. Store the ID i of o_i and its $\tilde{L}(o_i, R)$ for other processing.

6.2 Exploiting GPU Architecture

As presented in chapter 5, the GPU architecture is based on single instruction multiple thread (SIMT) model. All the threads perform the same operation, but for different data in parallel, which fits our algorithm. We propose three parallel strategies that work on different level for improving the running time and increasing the throughput.

It should be noted that since we are working with large-scale data, the data cannot be allocated neither on the GPU *global memory* nor on the CPU random-access memory (RAM). The indexing process is done by portion. Hence, the indexing process is organized as follows. We read a portion of the data of size N_l. The read portion is indexed and stored in the RAM. When the RAM is full, the data is stored on the hard-disk.

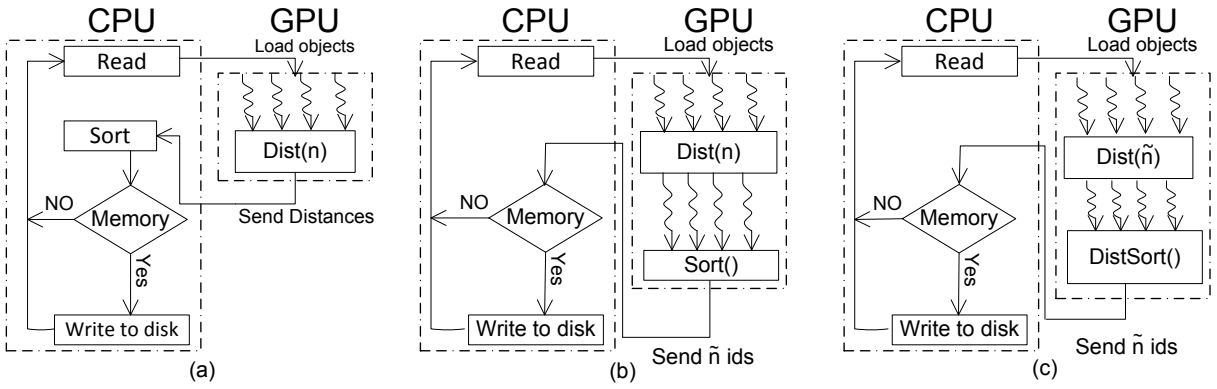


Figure 6.1: Our 3 GPU algorithms a) PDSS b) PDPS c) PIOF

6.2.1 Parallel Distance calculation and Sequential Sorting (PDSS)

As explained above, the indexing process is composed of two procedures. First, we calculate the distance between each object o_i and each reference point r_j in R. Then, we select the closest $\tilde{n}$ reference points to each object o_i by sorting the references based on their distances (building $\tilde{L}(o_i, R)$).

In this strategy, the work is shared between the CPU and the GPU. On the GPU, there is one kernel (*Dist()*), which calculates the distances between each N_l object and the reference set R. The distance information is then sent back to the CPU to be sorted and stored (Figure 6.1a). Using this technique, only the distances are calculated in parallel. Hence, the complexity of the algorithm is reduced to $O(\frac{Nn}{P} + N(nlogn)) + t_1$, where P is the number of the threads that can run in parallel at the same time and t_1 is the time needed to transfer the data between the CPU and GPU. t_1 increases when the partial number of objects N_l or references n increases due to the larger amount of distance information that needs to be transferred from the GPU to the CPU at each iteration. The data size that can be processed in parallel on the GPU has to satisfy this equation:

$$s \times ((N_l \times m) + (n \times m) + (N_l \times n)) < \text{GPU } \textit{Global Memory} \text{ Size}, \tag{6.1}$$

where $(N_l \times m)$, $(n \times m)$ and $(N_l \times n)$ represent the objects, the references and the distance information respectively. s represents a 4 byte data-type (*float* for distance information and *int* for object id). The distance calculation kernel on the GPU is build using 2 dimensional block with sizes equal to 16 and 2D grid with sizes equal to N_l divided by the block size. Hence, every thread calculates the Euclidean distance between a given object and a given reference point based on the thread coordinates [111].

6.2.2 Parallel Distance calculation and Sorting (PDPS)

Here, the complete process is done on the GPU. We parallelize the sorting step to minimize the size of the data transferred from the GPU to the CPU.

We have two kernels. The first kernel (*Dist()*) calculates the distances similarly to the previous technique, between the N_l read objects and the reference set R in parallel. In the second kernel (*Sort()*), each thread sorts a separate list using the QuickSort algorithm. The sorting process of one list on one GPU core is slower compared to using the CPU core, as the GPU cores are less powerful. On the other hand, on GPU, multiple lists are sorted in parallel, which should improve the running time (Figure 6.1b).

The complexity of the algorithm is therefore reduced to $O(\frac{N(n+(nlogn))}{P}) + t_2$, where t_2 is the time needed to send the ranks to the host. t_1 in *PDSS* is greater than t_2, because the number of elements that are sent to the CPU is $N_l \times n$ while, for this new technique, it is $N_l \times \tilde{n}$.

The main constraint of this technique is that all the data needs to reside on the GPU (the partial objects, the references, the ids of references and the distances values). Since, the GPU *global memory*

size is limited. Equation (6.1) becomes:

$$s \times ((N_l \times m) + (n \times m) + (N_l \times n) + (N_l \times \tilde{n})) < \text{GPU Global Memory Size}, \quad (6.2)$$

where $(N_l \times \tilde{n})$ represents the reference id (4 bytes). Accordingly, with a high number of reference points, the number of objects processed at the same time N_l has to be decreased, which decreases the throughput and affects the speedup.

6.2.3 Parallel Indexing On the Fly (PIOF)

As mentioned in section 5.3.2, a good performance can be achieved using the GPU, if the data that needs to be computed is small enough to fit in the registers of the GPU cores. Otherwise all the time is consumed in moving the data from and to the *global memory*. Hence, the sorting process on the GPU for the *PDPS* is a major bottleneck, since the distances are located in the GPU *global* memory.

We propose algorithm 6.2 to reduce the complexity of sorting to pick up the closest $\tilde{n}$ references. Instead of sorting the whole distances between an object and the list of references, to get the closest references, we sort only the first $\tilde{n}$ references of the list (line 1) by their distance to the object. Since, there might be other references in the list closer to the object than the objects in this pilot sorted partial list. The algorithm checks the distances starting from position $\tilde{n}+1$ to n (lines 2-5). If one of the distance is smaller than the distance at position $\tilde{n}$, the partial list is updated (lines 5-9). Lines 3-6 in algorithm 6.1 are simply replaced by algorithm 6.2.

Theoretically, the indexing sequential complexity is reduced from $O(N(n + nlogn))$ to $O(N(n + (\tilde{n}log\tilde{n}) + n))$, where $O(\tilde{n}log\tilde{n})$ represents the partial sorting and $O(n)$ represents the updating.

Algorithm 6.2 Picking $\tilde{n}$ nearest references

IN: Distance information b and $\tilde{n}$
OUT: $\tilde{L}(o_i, R)$
1. $quicksort(b, \tilde{n})$
2. For $j = \tilde{n} + 1 \rightarrow n$
3. If $(b[\tilde{n}].dis > b[j].dis)$
4. $i = BinarySearch(0, \tilde{n}, b[j].dis)$
5. $tmp = \tilde{n}$
6. while$(tmp \geq i)$
7. $b[tmp].dis = b[tmp - 1].dis$
8. $b[tmp].indx = b[tmp - 1].indx$
9. $tmp - -$

To map this strategy on the GPU, we have two kernels (Figure 6.1c). In the first kernel (*Dist()*), the

distance is calculated similarly to the previous techniques, but between the loaded objects and the first $\tilde{n}$ references only, to get a primary distance information (line 1 in algorithm 6.2). Accordingly, the size of the distance information array is reduced to $(N_l \times \tilde{n})$. Then, in the second kernel (*DistSort()*, lines 2-11 in algorithm 6.2), each thread calculates the distance between an object and the rest of references from position $\tilde{n}+1$ to position n. If a relative smaller distance information is found, the distance array is updated. Otherwise, the distance information is ignored. The complexity of the algorithm is reduced to $O(\frac{N(2n+(\tilde{n}log\tilde{n}))}{P}) + t_2$. For the memory constraint, equation (6.2) becomes:

$$s \times ((N_l \times m) + (n \times m) + 2(N_l \times \tilde{n})) < \text{GPU } \textit{global memory} \text{ Size}, \tag{6.3}$$

With this technique, we gain three privileges. The first gain is decreasing the size of the distance information array from $N_l \times n$ to $N_l \times \tilde{n}$. That allows to process more objects in parallel compared to *PDSS* and *PDPS*. The second gain is in decreasing the dependency between the threads by sharing the work between the kernels, which allows faster processing of the objects. The third gain is decreasing the latency between the cores and the *global memory*. In *PDPS*, for sorting the lists, the distance information is loaded from the *global memory*. With *PIOF*, the distance information is calculated instantly. Hence, it is located in the *local memory* and accessing the *local memory* is much faster than the *global memory* in the GPU architecture, which improves the performance.

In the second kernel (*DistSort()*), as the distance is calculated between an object and the rest of references (from $\tilde{n}+1\ to\ n$), we reduce the latency by taking a copy of the object under processing from the *global memory* to the *local memory*.

6.3 Multi-Core CPU

6.3.1 Single-disk access

In this procedure, there is only one thread which accesses the hard disk. The main thread reads N_l objects at each iteration. After that, P threads start to process the objects in parallel (creating the ordered list, sorting and storing it). The main issue with this technique is that, if one of the threads finishes its work before the others, it has to wait to synchronize with the master thread, since the master thread is the only thread that can access the hard disk per-time.

The complexity of the algorithm is therefore reduced to $O(\frac{N(n+(nlogn))}{P}) + t_s$, where t_s is the time needed for synchronization.

6.3.2 Multiple-disk access

Unlike in the previous procedure, all the threads can access the hard disk. Each thread reads multiple objects per-time and processes them separately without waiting for the master thread. To avoid random

disk access, we setup a pipeline. Hence, when the algorithm starts, the first thread reads a portion from the file, then the second thread reads a second portion and so on. Once a thread is done with its portion, it goes directly to the disk and reads the next portion. The total output size from all the threads should be less than the maximum size of the RAM:

$$s \times ((n \times m) + P((N_p \times m) + 2(N_p \times \tilde{n}))) < RAMSize, \tag{6.4}$$

where P is the number of active processes and N_P is the number of objects read by each processor (N_p=$\frac{N_l}{P}$).

The algorithm complexity is similar to that of the previous technique, but the pipeline strategy gives more flexibility for the threads, reduces the synchronization time t_s and avoids random disk access.

6.4 Indexing Evaluation

We use the CoPhIR dataset [92]. In our experiments, we extracted three data sets of sizes 1-million, 10-millions and 106-millions. The vectors in each dataset are constructed by combining the first three descriptors, which result in vectors with 208-dimension (more details about the CoPhIR dataset DB_6 in section 2.6.3). We applied our parallel indexing algorithms on the *metric permutation table* (MPT) [1]. The experiments are performed for pruned ordered lists of size $\tilde{n} = 10, B = 5$ and $\tilde{n} = 100, B = 20$ for $n = 1000$ and $n = 2000$.

The algorithms are implemented in C++. We used CUDA 5.1 and OpenMP version 3 for GPU and CPU parallelization respectively. The experiments were run on the *Squid* and the *GPU* machines. The specifications of the two machines are detailed in appendix A.1. On the two machines, the maximum RAM size which is used is 1.5 GB. Hence, writing to the hard disk is done when the size of the permutation lists produced exceeds 1.5 GB.

As the two machines do not have the same specifications, we run the sequential algorithm on both of them. On the first machine Seq_{C1}, to compare it with the GPU implementations. On the second machine Seq_{C2} to compare it with the CPU multicore implementation. For equity, we fixed the size of N_l, with respect to the dataset for all the algorithms. In general, whatever parallel algorithm is used, when the number of reference objects n or nearest reference objects $\tilde{n}$ increases, the indexing time consistently increases.

For a fair comparison, algorithm 6.2 is applied for all the implementations (sequential and parallel on CPU and GPU) to pick up the closest $\tilde{n}$ reference objects. Accordingly, the complexity of all the implementations (sequential, GPU and CPU) is reduced. Table 6.1 shows the complexity of the sequential and the parallel algorithm (CPU and GPU) using the normal QuickSort algorithm and the picking algorithm 6.2.

In section 6.4.1, we show a comparison between the different GPU implementations. In section 6.4.3,

Table 6.1: Multi-Core algorithms complexity.

Algorithm	QuickSort	Picking ($\tilde{n}$) (Alg. 6.2)
Sequential	$O(N(n + nlogn))$	$O(N(2n + (\tilde{n}log\tilde{n})))$
PDSS(GPU)	$O(\frac{Nn}{P} + N(nlogn)) + t_1$	$O(\frac{Nn}{P} + N(\tilde{n}log\tilde{n} + n)) + t_1$
PDPS(GPU)	$O(\frac{N(n+(nlogn))}{P}) + t_2$	$O(\frac{N(2n+(\tilde{n}log\tilde{n}))}{P}) + t_2$
PIOF(GPU)	-	$O(\frac{N(2n+(\tilde{n}log\tilde{n}))}{P}) + t_2$
Multiple-disk access (CPU)	$O(\frac{N(n+(nlogn))}{P}) + t_s$	$O(\frac{N(2n+(\tilde{n}log\tilde{n}))}{P}) + t_s$

we show the performance of the multi-core CPU implementations on different number of cores. Then, we analyse the GPU performance compared to the CPU performance.

6.4.1 Indexing using GPU

Table 6.2 shows the sequential indexing time Seq_{C1}, the parallel indexing time and the speedup ($S_p = \frac{Sequential\ time}{Parallel\ time}$) for the three GPU algorithms proposed in section 6.2.

Table 6.2: Average indexing time in minutes, average memory usage in megabytes and speedup on GPU.

N	n	$\tilde{n}$	Seq_{C_1}	GPU						
				N_l	PDSS	S_p	PDPS	S_p	PIOF	S_p
1M	1K	10	5	1000	3.9	1.3x	3.5	1.4x	**2.4**	**2x**
		100	6	1000	4.98	1.2x	6	1x	**5**	**1.2x**
	2K	10	10	1000	6.3	1.5x	6	1.7x	**3.54**	**2.8x**
		100	12	1000	8	1.5x	9	1.3x	**7.44**	**1.6x**
10M	1K	10	44	10000	33	1.3x	32	1.4x	**16**	**2.7x**
		100	50	10000	47	1.1x	62	0.8x	**42**	**1.2x**
	2K	10	82	10000	57	1.4x	56	1.5x	**24**	**3.4x**
		100	90	10000	72	1.3x	88	1x	**56**	**1.6x**
106M	1K	10	445	10600	345	1.3x	330	1.34x	**180**	**2.5x**
		100	513	10600	498	1x	498	1x	**473**	**1x**
	2K	10	888	10600	601	1.5x	570	1.6x	**261**	**3.4x**
		100	1206	10600	771	1.5x	996	1.2x	**620**	**1.9x**

From table 6.2, we note that the *PIOF* algorithm gives the best indexing time and speedup compared to the *PDSS* and *PDPS* algorithms. The reason is the partial sorting process that is achieved through the picking algorithm that we proposed (algorithm 6.2). That helps to calculate the distances instantly, which makes the distances reside in the *local memory*. Accordingly, the latency of sorting which is consumed in getting the distance values from the *global memory* is reduced.

For the *PDSS* algorithm, we note that the speedup decreases when the number of nearest references $\tilde{n}$ increases, as the sorting step is done sequentially. For the *PDPS*, with small number of nearest references $\tilde{n} = 10$, the algorithm performs better than the *PDSS* algorithm. On the other hand, when $\tilde{n} = 100$, *PDPS* becomes similar or slower than *PDSS* in performance. The main reason is that for the *PDPS*, the

distances are pre-calculated and saved in the *global memory*. As a result, the sorting process on GPU becomes not efficient due to increasing the latency for getting the distances from the *global memory*. Hence, when $\tilde{n}$ increases, the parallelization which is offered by the GPU for sorting becomes not as efficient as the sequential sorting on the CPU. This effect is reduced, in the *PIOF* algorithm as the distance information are calculated instantly, which means that it is located in the *local memory*, which improves the performance.

Table 6.3 shows the memory usage per iteration for the three proposed algorithms.

Table 6.3: Average memory usage in megabytes on GPU.

N	n	$\tilde{n}$	GPU			
			N_l	PDSS	PDPS	PIOF
1M	1K	10	1000	5.4MB	5.4MB	**1.6MB**
		100	1000	5.4MB	5.8MB	**2.3MB**
	2K	10	1000	10MB	10MB	**2.5MB**
		100	1000	10MB	11MB	**3.5MB**
10M	1K	10	10000	46MB	47MB	**5MB**
		100	10000	46MB	50MB	**16MB**
	2K	10	10000	85MB	86MB	**10MB**
		100	10000	85MB	89MB	**17MB**
106M	1K	10	10600	49MB	50MB	**10MB**
		100	10600	49MB	53MB	**10.5MB**
	2K	10	10600	90MB	91MB	**10MB**
		100	10600	90MB	94MB	**18MB**

The *PIOF* also gives the lowest memory usage at each iteration. The main reason is that we do not store all the distance information. We store only $\tilde{n}$ distance values and consider them as a guidance to calculate the rest of references, that allows to reduce the size of the memory that is needed to be allocated. That, in turn, helps to process more objects in parallel, which improves the total running time.

6.4.2 Effect of N_l

The number of objects N_l that are processed at the same time affects the indexing time. To study the effect of N_l, we have increased the size of $N_l = 106000$ (10 times) and run the experiments on the 106M dataset for $n = 1000$, $n = 2000$ and $\tilde{n} = 100$. Table 6.4 shows the indexing time using $N_l = 106000$ for the *PIOF* algorithm. Comparing to the results in table 6.2, it is clear that the running time is reduced and we gain more speedup when the number of objects that are processed in parallel increases. We also believe that with more powerful GPU (more cores, more *global memory*), a much better performance can be achieved.

Table 6.4: Average indexing time in minutes, average memory usage in megabytes and speedup on GPU for the *PIOF* for $N_l = 106000$.

N	n	$\tilde{n}$	Seq_{C_1}	GPU			
				N_l	M	PIOF	S_p
106M	1000	100	513	106000	170MB	466	1.1x
	2000	100	1206	106000	180MB	600	2.1x

6.4.3 Indexing using multi-core CPU

Tables 6.5 and 6.6 show the sequential indexing time, the parallel indexing time, the speedup (S_P) and the efficiency ($E = \frac{S_P}{P}$) for the three datasets on 4, 8 and 16 multicore CPU for the *single-disk access* and *multi-disk access* algorithms presented in section 6.3.2 on the second machine C_2.

Table 6.5: Average indexing time in minutes, speedup and efficiency on multi-core CPU for *single-disk access* algorithm

N	n	$\tilde{n}$	Seq_{C_2}	Multi-core CPU (*single-disk access*)								
				4	E_4	S_4	8	E_8	S_8	16	E_{16}	S_{16}
1M	1K	10	4	1.96	0.51	2.03x	1.5	0.33	2.64x	1.35	0.19	2.96x
		100	4.5	1.93	0.58	2.3x	2.1	0.3	2.14x	2	0.14	2.3x
	2K	10	9.3	2.9	0.79	3.2x	2.55	0.45	3.6x	2.2	0.3	4.3x
		100	9.4	4.4	0.53	2.12x	4.4	0.3	2.1x	6.3	0.094	1.5x
10M	1K	10	40.7	17.4	0.58	2.3x	14	0.36	2.9x	10.9	0.23	3.7x
		100	46	18.1	0.63	2.5x	15.3	0.4	3x	12.2	0.24	3.8x
	2K	10	73	29.4	0.6	2.5x	24.3	0.4	3x	17.4	0.3	4.2x
		100	82.4	31.7	0.65	2.6x	27.2	0.38	3x	22	0.23	3.8x
106M	1K	10	402	140	0.0.71	2.9x	109	0.5	3.7x	129	0.193	3.1x
		100	450	187	0.6	2.4x	144.9	0.38	3.1x	135	0.21	3.3x
	2K	10	804	279	0.72	2.88x	171.8	0.58	4.7x	153	0.33	5.2x
		100	912	298	0.76	3x	189.2	0.6	4.8x	174.8	0.33	5.2x

We note that the *multi-disk access* algorithm is faster than the *single-disk access* algorithm. In *multi-disk access* algorithm the threads do not need to wait for each other in order to access the hard disk compared to the *single-disk access* algorithm, as was discussed in section 6.3.2. Hence, its indexing time, speedup and efficiency are better compared to the *single-disk access* algorithm.

For the two algorithms, we note that when the number of cores increases, the efficiency decreases. The reason for that is the increasing time for reading from and writing to the hard-disk. For reading (*multi-disk access*), the threads access the hard-disk in a pipeline fashion. Hence, when we have more threads, the pipe-lining queue length increases, which increases the running time. For writing (*single-disk access* and *multi-disk access*), when the number of processors increases, more objects are processed at the same time. The available RAM size is then divided by the available number of processes, which increases the writing rate to the hard-disk as the RAM is filled in at a higher rate. For example, we

Table 6.6: Average indexing time in minutes, speedup and efficiency on multi-core CPU for *multi-disk access* algorithm.

N	n	$\tilde{n}$	Seq_{C_2}	Multi-core CPU (*multi-disk access*)								
				4	E_4	S_4	8	E_8	S_8	16	E_{16}	S_{16}
1M	1K	10	4	1.8	**0.55**	2.2x	1.2	**0.41**	3.3x	0.66	**0.375**	6x
		100	4.5	1.4	**0.8**	3.2x	1.2	**0.47**	3.8x	1.3	**0.22**	3.5x
	2K	10	9.3	3.9	**0.6**	2.4x	2.4	**0.48**	3.8x	1.2	**0.48**	7.7x
		100	9.4	3	**0.75**	3x	2.46	**0.5**	4x	2.94	**0.2**	3.2x
10M	1K	10	40.7	10.5	**0.98**	3.8x	5.4	**0.94**	7.5x	5.34	**0.48**	7.7x
		100	46	13.2	**0.86**	3.4x	7.8	**0.74**	6x	7.5	**0.38**	6x
	2K	10	73	19.8	**0.93**	3.7x	10.5	**0.88**	7x	7.8	**0.59**	9x
		100	82.4	22.8	**0.9**	3.6x	13.2	**0.77**	6x	10.68	**0.5**	8x
106M	1K	10	402	108	**0.93**	3.7x	85	**0.59**	4.7x	42.6	**0.59**	9.4x
		100	450	132	**0.85**	3.4x	75	**0.75**	6x	66	**0.43**	6.8x
	2K	10	804	204	**0.98**	3.9x	120	**0.83**	6.6x	84	**0.59**	9.4x
		100	912	240	**0.95**	3.8x	126	**0.9**	7x	120	**0.48**	7.6x

assigned 1.5 GB of memory for our application. With 16 cores, each thread has only around 96 MB of this memory, which increases the writing rate to the hard disk for each thread. There should be a good balance between the available memory and the number of cores that are used to avoid disk contention and to get a good performance.

Increasing Memory Size for Multi-Core CPU

To measure the effect of the memory size on the multi-core CPU algorithm, we indexed the 106M dataset again but with different memory size per thread for $n = 1000$, $n = 2000$ and $\tilde{n} = 100$. In these experiments, we assigned 1.8GB of memory for each thread. Hence, if we have 16 running processes, the total allocated memory would be 28.8GB. Table 6.7 shows the running time in minutes. Comparing to the results in table 6.6, it is clear from the table that increasing the memory per thread improves the indexing time. The main reason for that improvement is the decrease of the writing time to the disk.

Table 6.7: Average indexing time in minutes, speedup and efficiency on multi-core CPU for *multi-disk access* algorithm with 1.8GB per thread.

N	n	$\tilde{n}$	Seq_{C_2}	Multi-core CPU *multi-disk access*		
				16	E_{16}	S_{16}
106M	1000	100	450	50	**0.6**	9x
	2000	100	912	80	**0.7**	11x

6.4.4 Comparing GPU to Multi-core CPU

Figures 6.2a and 6.2b show a comparison between the speedup of the *PIOF* algorithm on GPU to the speedup of using a 4 core CPU, for the 106M dataset using $n = \{1000, 2000\}$, $\tilde{n} = \{10, 100\}$, $N_l = 10600$ and a total memory of 1.5 GB. We note that the CPU provides much better performance at a lower cost.

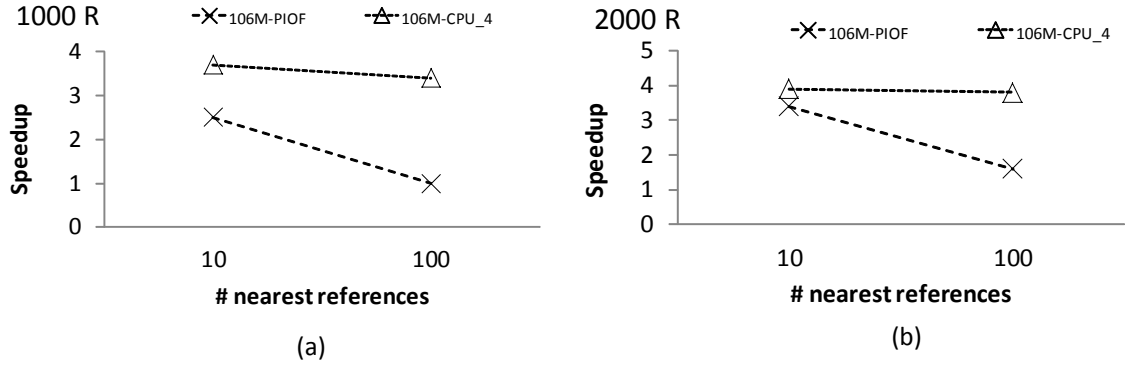


Figure 6.2: Indexing speedup for the 106 million data set using the *PIOF* and *multi-disk access* (4 cores) algorithms. a) $n = 1000$, $\tilde{n} = \{10, 100\}$ b) $n = 2000$, $\tilde{n} = \{10, 100\}$.

We believe that there are two reasons for that difference in performance. The first reason is that the data cannot be loaded on the GPU directly. CUDA does not support system calls. Hence, the reading should be done via the CPU. This results in a two-layer process, the data is read in the RAM, then transferred to the GPU. While for the CPU, it is a one layer process, the data is read to the RAM and processed directly.

The second reason is the algorithm itself. To get a good performance on the GPU, the GPU kernel should be as compact as possible to make all the data on the registers of the cores, which is not possible in the case of our algorithm due to the huge number of sorting operations that is required. The figures 6.2a and 6.2b prove that. For low number of nearest reference points ($\tilde{n} = 10$), the speedup of the GPU implementation is close to the CPU implementation. When $\tilde{n} = 100$, the difference between them increases due to increasing the number of sorting operations.

6.5 Multi-Core Searching

The naive permutation-based indexing search algorithm is a computationally expensive process. As mentioned before, there are several data structures that were proposed to handle the permutation lists. Each one of them is based on a different technique and handles the searching based on its own structure. In this section, we show how can we adapt our MPT searching algorithm on CPU and GPU.

6.5.1 Parallel Searching on GPU

As explained in section 4.2, the searching algorithm counts the discrepancy in quantized ranking from common reference objects between each object and the query. As the counting is not computationally expensive, when we do it list by list most of the time would be consumed on transferring the data from the host to the device and vice versa. Accordingly, we load the object IDs once from all the lists where $r_j \in \tilde{L}(q, R)$. After decompressing, the ids are sent to a GPU kernel to count how many time each object in $\tilde{L}(q, R)$ appeared in object lists. With this implementation, there is a high chance that more than one thread might access the same counter simultaneously, which affects the output results. To overcome this race condition problem, we set the increment process as an atomic operation. Atomic means that the process is inseparable, either happens completely or it does not happen at all. In addition, other processes cannot interfere during the execution. Thus, if more than one object appeared many times in the list, the majority of the processing time would be consumed in organizing the processing of the atomic process. As a result, the performance of the searching algorithm on the GPU would be affected.

6.5.2 Parallel Searching on CPU

On the CPU, as loading the data is faster than on the GPU, we perform the parallelization on the lists. We read the lists one by one and the counters are updated based on the objects in these lists in parallel. Hence, we do not need any atomic operation as for the GPU implementation.

Theoretically, the complexity for the two implementations becomes $O(\frac{\tilde{n}z}{P})$, where P is the number of active processors.

6.5.3 Comparing GPU and Multi-core CPU Searching Techniques

In MPT, the size of the lists assigned to each bucket decreases as n increases. The main reason is that the distribution of the objects across the references changes, which leads to a decreasing running time. The searching time however increases along with the number of nearest reference objects $\tilde{n}$ [1]. Table 6.8 shows the sequential searching time, the parallel searching time on GPU and CPU and the speedup for the multicore-CPU using on the GPU machine (appendix A.1).

For multicore-CPU, the speedup is not linear either, since the work that can be done in parallel does not consume the majority of the time. For the GPU, searching using the GPU becomes slower than the sequential version. The main reason is the atomic operation and the data transfer, as we have explained before. The only way to avoid these atomic operation is to load the data bucket by bucket similar to the multi-core algorithm. Nevertheless, this technique consumed more time than the atomic operation. This is due to transferring the data from the RAM to the GPU *global memory* for each bucket list. For example, for 1 million objects, $n = 1000$ and $\tilde{n} = 100$ the searching time is $0.4s$ when we process bucket by bucket compared to $0.1s$ for the atomic operation.

Table 6.8: Average searching time in seconds.

N	n	$\tilde{n}$	Seq	GPU	CPU_4	S_4
1M	1K	10	0.05	0.06	0.02	2.5x
		100	0.08	0.1	0.03	2.6x
	2K	10	0.04	0.05	0.02	2x
		100	0.07	0.09	0.03	2.3x
10M	1K	10	0.5	0.7	0.3	1.6x
		100	0.8	1	0.4	2x
	2K	10	0.4	0.6	0.2	2x
		100	0.6	0.9	0.36	1.6x
106M	1K	10	9	11	4	2.3x
		100	12	15	6	2x
	2K	10	7	10	3.7	1.8x
		100	10	12	5.2	1.9x

6.5.4 Recall and Position Error

We measure the average recall (RE) and the average position error (PE) (detailed in section 2.6). Figure 6.3(a) shows the average RE and Figure 6.3(b) shows the average PE for the third dataset which contains 106 million objects. We used the MPT data structure (chapter 4) to store the permutation lists. The experiments were performed fixing the direct distance calculation factor $\Delta = 40$ for pruned ordered lists of sizes $\tilde{n} = \{10, 100\}$ with a quantization factor $B = \{5, 20\}$, respectively for $n = 1000$ and $n = 2000$ reference objects (see chapter 4 for the definition of Δ and B). The average RE and PE were measured using 100 queries selected randomly from the dataset for the 1-NN, 10-NN and 100-NN search scenarios. We note that the RE and PE did not change with any of the parallel indexing algorithms.

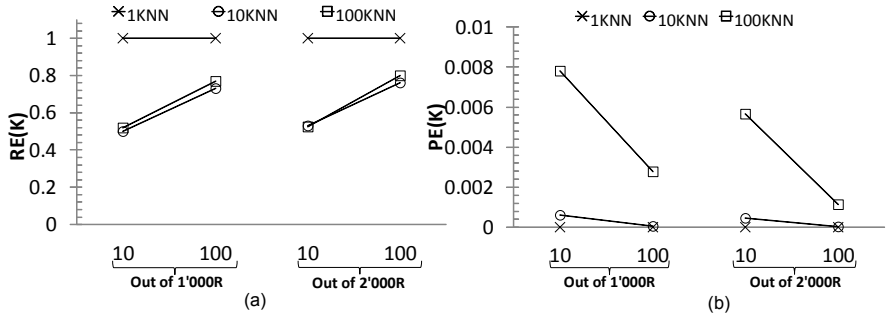


Figure 6.3: a) RE and b) PE for 106M CoPhIR objects, using $\Delta = 40$, $\tilde{n} = \{10, 100\}$, out of $n = 1000$ and $n = 2000$ for 1-NN, 10-NN and 100-NN queries

Similar to the sequential algorithm, the RE increases while PE decreases when $\tilde{n}$ increases. This shows that $\tilde{n}$ directly impacts the quality of approximation. When comparing the RE and PE for the same number of nearest reference objects $\tilde{n}$ out of an increasing total number of reference objects n, we see

that the RE increases and PE decreases due to decreasing the number of false positives. For example, for 10-NN (Figure 6.3(a)), if $\tilde{n} = 100$ out of $n = 1000$ and $n = 2000$ reference objects, RE$= 0.7$ and RE$= 0.75$, respectively.

6.6 Summary

The exponential increase in data and the availability of multi-core architecture (CPU and GPU) drives us to use parallel computation to obtain faster results at a lower cost. We presented different permutation-based indexing algorithms for multi-core GPU and CPU architectures.

For GPU, we proposed three parallel strategies that work on different level for improving the running time and increasing the throughput. These strategies are *PDSS, PDPS and PIOF*. The *PDSS* divided the indexing process between the CPU and the GPU, which makes the algorithm scalable but with limitation, as the distance calculation is the only process which is done in parallel. The *PDPS* does all the computation on the GPU but, with larger number of reference points, its performance becomes worse than the *PDSS*. The main reason for that is the sorting process on the GPU is slow, due to increasing the latency of fetching the data from the *global memory*. We tried to overcome this problem in the *PIOF* strategy by sorting only a small portion of the reference points and then perform an update process in case a close reference appeared. This algorithm gives much better speedup than the previous two techniques. However, when the number of nearest references increases, its performance becomes like the *PDPS* technique.

The main bottleneck for multi-core CPU indexing is the disk access as the data can not be located in the memory. Hence, we proposed a multiple-disk access algorithm, that allows multiple threads to access the hard disk at the same time, in a pipeline fashion. We show that, with using multi-core CPU, much better performance can be achieved compared to the GPU, at a lower cost due to the limitation of the GPU platform and the complexity of our algorithm.

We also proposed multicore searching algorithms on GPU and CPU for the MPT data structure. We found that the GPU searching is not efficient due to the searching strategy of the MPT implementation. We also show that the efficiency of the algorithm in terms of recall (RE) and position error (PE) is not affected by the type of the parallel algorithm that is used.

Chapter 7

Distributed Approaches For Permutation-Based Indexing

As discussed in chapter 6, the GPU and multicore architectures are not the best possible solutions due to the limitations of the memory size, memory access speed and the architecture. In this chapter, we propose different distributed approaches for permutation-based indexing, which can be adapted for the different permutation-based indexing data structures [1, 61, 90, 70, 66] that we discussed in chapters 2 and 4. The proposed distributed approaches are based on different programming models, including our enhanced MapReduce implementation (MRO-MPI), that was discussed in section 5.4.

The chapter is organized as follows. In the next section, we propose three distributed approaches for indexing and their related searching methodologies. In section 7.2, we show the technical implementations of the indexing for the three approaches using MapReduce. In section 7.4, we show a comparative study between the three proposed approaches in terms of indexing time, searching time, recall and position error.

7.1 Distributed Permutation-Based Indexing approaches

The permutation-based indexing model can be adapted on a distributed environment through three approaches:

1. Distributed Data.
2. Distributed References.
3. Distributed Data-References.

We discuss the three approaches in details in the next three subsections. Also, we show how they can be applied on two different data structures which are the metric inverted files (MIF) [8] and the metric

suffix array (MSA) [3, 2], which were discussed in detail in chapter 2.

7.1.1 Distributed Data

The main idea for this model is to divide the data structure between the processes based on the input data, while sharing the same reference information.

Indexing

Given a data domain D of N objects, a reference set R of n points and P processes. The reference set R is shared between all the processes. Correspondingly, the data domain D is randomly divided into sub-domains $D_1 \dots D_P$ of equal sizes $\frac{N}{P}$, among the processes. Each process p builds the partial ordered lists $\tilde{L}(o_i, R)$ for all $o_i \in D_p$. The ordered lists for these objects are saved locally in the appropriate data structure DS_p for each process. As a result, the workload of indexing is divided equally between the processes. Algorithm 7.1 shows the indexing process in detail by each process.

Algorithm 7.1 Distributed Data Approach: Indexing

IN: D_p of size $\frac{N}{P}$, R of n and $\tilde{n}$
OUT: Data structure DS_p
1. For $o_{i_p} \in D_p$
2. For $r_j \in R$
3. $b[j].dis = d(o_{i_p}, r_j)$
4. $b[j].indx = j$
5. $L(o_{i_p}, R) = quicksort(b, n)$
6. $\tilde{L}(o_{i_p}, R) = partiallist(L(o_{i_p}, R), \tilde{n})$
7. Store the global object ID of o_{i_p} and its $\tilde{L}(o_{i_p}, R)$ for other processing.

Using this procedure, the complexity of indexing is reduced from $O(N(n+nlogn))$ to $O(\frac{N(n+nlogn)}{P})$.

Searching

The data is equally divided between all the processes. Hence, whatever the data structure DS_p that is used, when a query q is submitted, all the processes have to participate to answer the submitted query. Algorithm 7.2 shows the searching process. There is a *broker* process (line 1). This broker process is responsible to handle users' requests. Once a query is received, the query is broadcasted to all the processes, which we call the *workers* (line 2). Each worker creates $\tilde{L}(q, R)$ and searches through its local partial data structure DS_p (lines 4-6). Once the search is finished, the resulting ranked list from every process RL_p are sent back to the broker (line 7). The broker organizes the received results lists $RL_1, \dots, RL_P$ and send them to the user as a one global ranked list RL (lines 8-11).

The organization of the results is simple. The similarity between the query the objects in any data domain is identified using the full reference list. Hence, the organization of the results lists consists in merging the lists and sorting them based on the similarity score that is related to each object. This similarity score is defined based on the searching algorithm that is used.

Algorithm 7.2 Distributed Data Approach: Searching

IN: Data structure DS_p, References R and Query: q
OUT: output ranked list RL
1. if (*broker*):
2. Broadcast(q)
3. else:
4. Recv. q from the *broker*
5. create the query partial ordered list $\tilde{L}(q, R)$
6. RL_p=Search in DS_p using $\tilde{L}(q, R)$.
7. Send RL_p to the *broker*
8. if(*broker*)
9. For $p \in P$
10. Recv(RL_p).
11. Merge $RL_1 \dots RL_P$ to create RL and send the results to the user.

Applicability

The *Distributed Data* approach is applicable for all the permutation-based indexing data structures [1, 61, 90, 70, 66]. We applied it on the MSA [3] and the MIF [8]. In [3], when the *Distributed Data* approach is applied, the MSA array is distributed over the running processes, which we call *Distributed MSA* (Figure 7.1(a)).

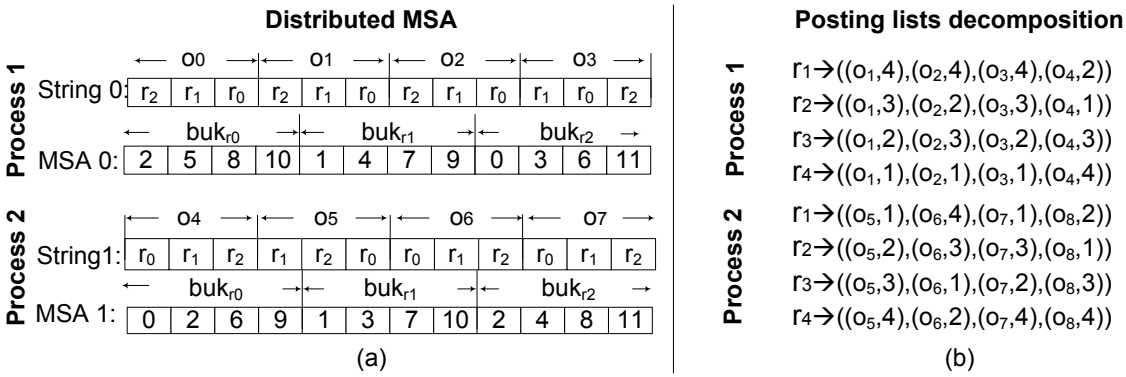


Figure 7.1: a) Distributed MSA b) MIF: Posting Lists Decomposition

Each process is responsible for a part of the MSA. In [8], when the *Distributed Data* approach is

applied, the posting list for each reference point is distributed on the running processes, which we call *Posting Lists Decomposition (PLD)* (Figure 7.1(b)). Each reference for each process has a portion from the full posting list. The search strategy for the two approaches is based on ranking the objects based on their SFD score (Def. 4). Hence, the *broker* applies the *Mergesort* algorithm on the received output lists in order to select the closest objects to the query.

The *Distributed Data* technique has two advantages. The first advantage is equal data division. The data is equally divided over the active processes, which leads to an effective load balancing. The second advantage is the low communication time while indexing. The processes do not need to communicate between each other while indexing, which makes the algorithm scalable.

The main disadvantage is the usage of all the processes for answering one query due to the random data division and indexing the objects within each domain using the same reference list. That means this approach does not support well inter-query parallelism. More clearly, multiple queries can not be answered efficiently at the same time.

7.1.2 Distributed References

The number of reference points is an important factor that affects the indexing and search time for permutation-based indexing. Hence, if the data is divided between the processes based on the references, all the processes would not need to participate to answer one given query.

Similar to the previous technique, the data domain is divided into P non-overlapping sub-domains $D_1 \dots D_P$. However, we will use the locality inducted by our list pruning to organize the objects with respect to their corresponding closest reference objects.

Indexing

Here, the references have a different distribution depending on the role of the processes. Each process has two roles. Each process is *indexer* and *coordinator*. When processes are in the indexing role, they share all the reference information R. Otherwise, when they are in the coordinating role, the reference set is equally divided between them. Accordingly, each process has an independent subset of the references $R_p \subset R$, where $|R_p| = \frac{n}{P}$. Algorithm 7.3 shows the indexing process.

All the processes start in the indexing role. For each object $o_i \in D_p$, the partial ordered list $\tilde{L}(o_i, R)$ is created using the full reference set R (lines 1-7). Then, the locations of the $\tilde{n}$ closest reference points in the partial ordered list of each object $\tilde{L}(o_i, R)_{|r_j}$, the reference id j and the related object global id are sent to the corresponding coordinating process (line 8-12). A corresponding coordinating process is a process p, where $r_j \in R_p$. Accordingly, each process p is responsible for a group of objects, which have the related set of references R_p as the closest references. Formally, a set of objects o_i belongs to process p is characterized as

$$\{o_i \; such \; that \; r_j \in \tilde{L}(o_i, R) \; and \; r_j \in R_p\}$$

Hence, after the indexing process the data would be divided between the processes based on the references, and not equally as the previous technique. The complexity of indexing is $O(\frac{N(n+nlogn)}{p}) + t_1$, where t_1 is the time needed to transfer the objects information from the indexing process to the coordinating process.

Algorithm 7.3 Distributed References approach: Indexing

IN: D_p of size $\frac{N}{P}$, R of n, R_p of $\frac{n}{P}$,$\tilde{n}$
OUT: Data structure DS_p
1. if(*indexer*)
2. For $o_{i_p} \in D_p$
3. For $r_j \in R$
4. $b[j].dis = d(o_i, r_j)$
5. $b[j].indx = j$
6. $L(o_{i_p}, R) = quicksort(b, n)$
7. $\tilde{L}(o_{i_p}, R) = partiallist(L(o_{i_p}, R), \tilde{n})$
8. For $r_c \in \tilde{L}(o_{i_p}, R)$
9. Send the global object ID, the $\tilde{L}(o_{i_p}, R)_{|r_c}$, and the reference id r_c to coordinating process.
10. if(*coordinator*)
11. Recv. data from any indexer.
12. Store the received data in DS_p.

Searching

Unlike in the *distributed data* search scenario, the processes that participate to answer the query are the processes which have the references that are located in the query partial ordered list. Algorithm 7.4 shows the search process.

Once a broker process receives a query, the partial ordered list $\tilde{L}(q, R)$ is created (line 2). An active process p is a process having at least one of its references R_p included in $\tilde{L}(q, R)$. More formally, a process p is defined as an active, if

$$\exists\, r_j \in R_p \;/\; r_j \in \tilde{L}(q, R)$$

Hence, the broker notifies the active processes and sends the position of the related references $\tilde{L}(q, R)_{|r_j}$ to them (lines 3-4). Each active process searches within its local partial data structure and sends the results back to the broker (lines 5-9). The broker organizes the received results and sends them to the user (lines 10-13). The organization of the results is not simple. In this approach, each process defines the similarity between the query and the related objects based on a partial list of references R_p. Hence, the same object may be seen by different processes, so the object can be found in each output list from each process with a partial similarity score. Hence, the broker has to combine (add) these partial

Algorithm 7.4 Distributed References approach: Searching

IN: Data structure DS, References R and Query: q
OUT: output list OL
1. if (*broker*):
2. Create the query partial ordered list $\tilde{L}(q, R)$
3. Notify the active processes: $r_j \in \tilde{L}(q, R)\ and\ r_j \in R_p$
4. Send $\tilde{L}(q, R)_{|r_j}$ to the active processes p
5. if (*active*):
6. Recv. $\tilde{L}(q, R)$
7. For $r_j \in \tilde{L}(q, R)\ and r_j \in R_p$
8. OL_p=Search in DS_p.
9. Send OL_p to the *broker*
10. if(*broker*):
11. For $p \in P$
12. Recv(OL_p).
13. Create OL and send the results to the user.

similarity scores related to each object and then sort the objects based on their total similarity score.

Applicability

This technique cannot be applied on all the permutation-based indexing data structures, such as [66]. The main reason is that, in [66], the search algorithm needs the references to be close to each other and not distributed on different processes. This approach is nevertheless applicable on MPT [1], MIF [61], PP-Index [90] and NAPP [70].

We applied this approach on the MSA [3] and MIF [8] data structures. We applied it on the MSA to propose the *Bucket Decomposition* technique [3]. The main idea is to divide the MSA array on the processes based on the references buckets. (Figure 7.2(a)). In [8], when the *Distributed References* approach is applied, the MIF is distributed based on the references, which we call *Reference List Decomposition (RLD)* (Figure 7.2(b)). Instead of dividing the MIF based on the objects, it is divided by the reference list. The search strategy for the two approaches is based on ranking the objects based on their SFD (Def. 4) value to the query. The broker cannot sort the objects based on their partial SFD similarity score directly. The scores for the same object have to be added first before sorting.

The *Distributed References* technique has one main advantage. All the processes do not need to participate to answer one query. Therefore, multiple queries can be answered in parallel efficiently.

The main disadvantage of this model is the indexing time. It consumes a lot of time, as the indexing process might not be the coordinating process. Hence, the time would be consumed in the transformation of the data between the processes. Another main disadvantage is the unbalanced structure. If the

references were not selected in an efficient way to cover the whole data space, there is a high probability that one reference point appears as the closest reference to many objects. In that case, the process responsible for this reference point would be responsible for a large number of objects, which would affect the searching time.

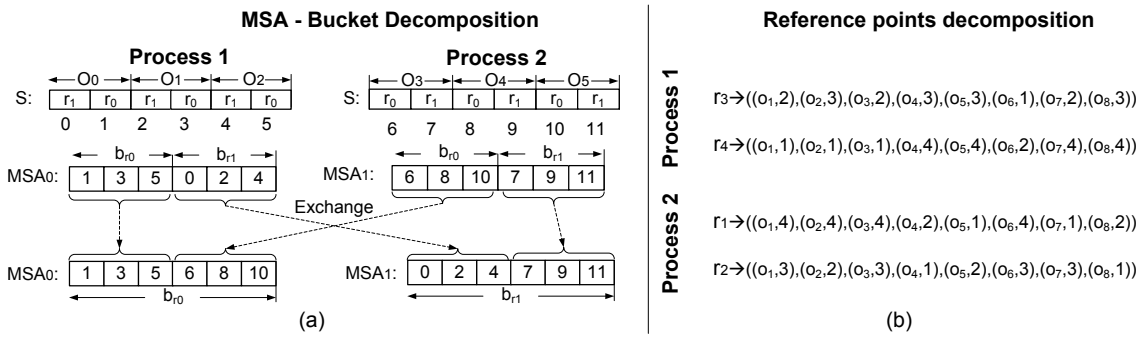


Figure 7.2: a) MSA: Bucket Decomposition b) MIF: Reference List Decomposition

7.1.3 Distributed Data-Reference

In the previous two approaches, the distributed indices are related to each other based on the data or the references. While searching for a given query, more than one process might participate in the search process. In this model, we aim to decrease the dependency between the data and the references.

Indexing

Similarly to what was done in the previous techniques, the data domain is divided into P sub-domains $D_1 \dots D_P$, based on a clustering algorithm. If the data is large, it will not be feasible to apply the clustering algorithm. Hence, the data can be divided randomly. However, this could affect the searching space and increase the number of active processes that would participate to answer the query (more details in chapter 8). The references are selected independently from each sub-domain and not shared with the other processes R_p. Once the references are defined for each portion, the processes create the partial ordered lists $\tilde{L}(o_i, R_p)$ for their own local objects D_p based on their local reference sets R_p. Hence, with this approach, the processes do not need to communicate to each other while indexing.

The indexing complexity is $O(\frac{N(n+nlog\frac{n}{P})}{P^2})$. The main reason for this significant reduction in the complexity is the division of the references between all the processes.

Searching

Algorithm 7.5 shows the search process. The broker process has information about all the references $R = \{R_1 \cup R_2 \cup \dots \cup R_P\}$. Hence, once a query q is submitted, the partial ordered list $\tilde{L}(q, R)$ is

created using the full reference list. If the query is located within a certain data domain D_p, the reference points R_p which were selected from that data domain are located as the closest references to the query. Hence, searching is only done by the process responsible for the corresponding data domain (lines 1-4). If the query is located between two or more data domains, the closest references are shared between these different domains. Formally, a process p is an active process if

$$\exists r_j \in R_p / r_j \in \tilde{L}(q, R)$$

After searching, the results from all the processes cannot be sent to the broker directly. The similarity between the query and each object for each process is defined by different sets of reference points R_p that are not related to each other. To combine and rank them, each process computes the distance between q and the $K_{c_p} = \Delta \times k$ first objects in its candidate list. $\Delta \geq 1$ is the *direct distance calculation (DDC)* factor (chapter 4). This distance information, along with the related objects ids, are sent to the broker process (lines 5-10). Finally, the broker ranks these objects based on their distance values and sends the results back to the user.

For example, if $\Delta = 2$, for *KNNquery(q,30)* each process searches within its local data structure and gets the top approximate candidate list $K_{c_p} = 60$. Then, a direct distance calculation is performed between the query and this top $K_{c_p} = 60$ set. If the references of the query ordered list are located within the sets of 3 processes, the broker receives $3 \times 60 = 180$ distances. These 180 distances are sorted and the top 30 objects are sent back to the user.

Algorithm 7.5 Distributed Data-Reference approach: Searching

IN: Data structure DS, References R and Query: q
OUT: output list OL
1. if (*broker*):
2. Create the query partial ordered list $\tilde{L}(q, R)$
3. Notify the active processes
4. Send $\tilde{L}(q, R)$ to the active processes
5. if (*active*):
6. Recv. $\tilde{L}(q, R)$
7. For $r_j \in \tilde{L}(q, R)\ and r_j \in R_p$
8. OL_p=Search in DS_p.
9. d_p=Calculate_distance(q,$\Delta \times k$,OL_p)
10. Send the distances d_p to the broker.

Applicability

This algorithm is applicable for all the permutation based indexing data structures. We applied these indexing and searching approaches on MIF [8] and the metric suffix array MSA[3]. For a better locality and organization of the data, instead of dividing the data randomly, we cluster the data into P clusters. For each cluster p, a center c_p and a radius ρ_p are determined. The center c_p is an average imaginary object for all the other objects in the cluster and ρ_p is the distance from the center of the cluster to the farthest object in the cluster. Every process nominates a number of reference points R_p from its own cluster based on the number of objects in the cluster. Then, the indexing is done independently.

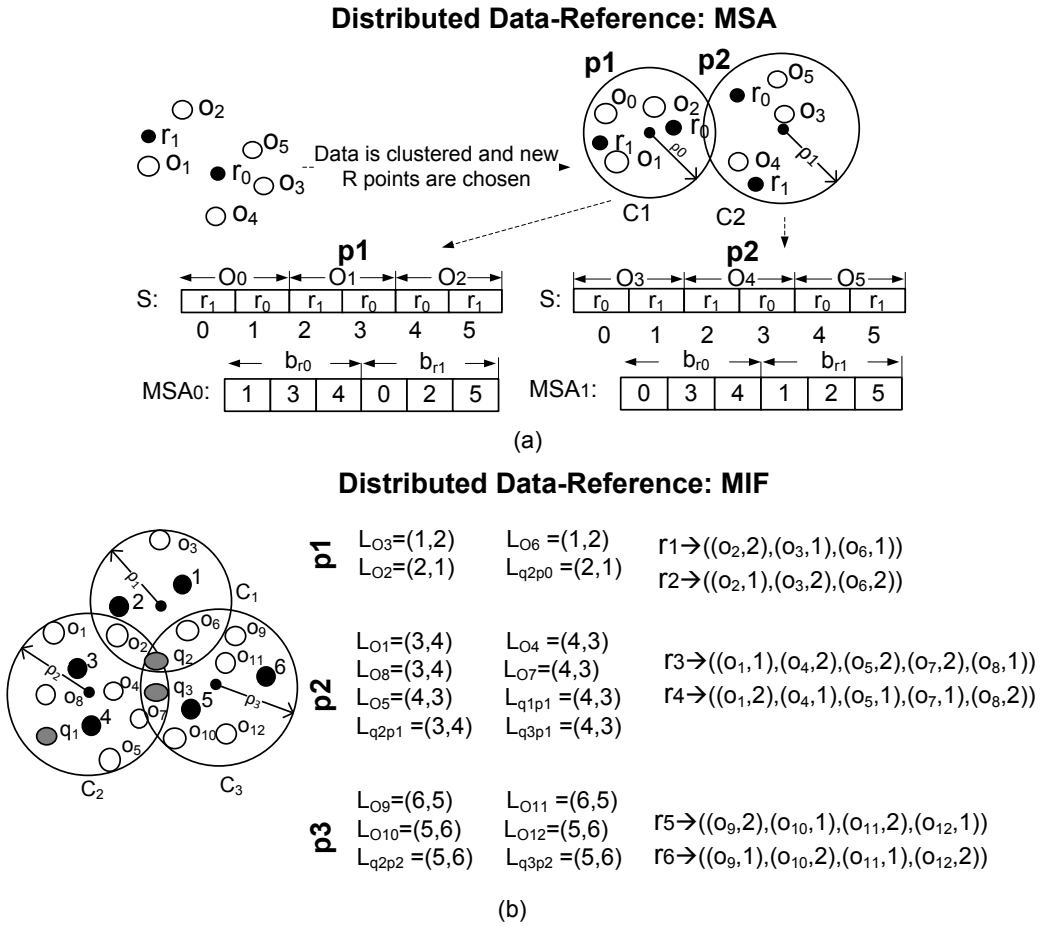


Figure 7.3: a) Independent Distributed MSA b) Independent Distributed MIF

In terms of searching, instead of defining the active processes based on the shared number of nearest references, the active process is defined based on the distance value between the query and the central

imaginary point. Hence, a process p is active if:

$$d(q, c_p) < \rho_p$$

The search procedure is then followed as discussed before. Note that this split is efficient if the clusters are compact and spherical as often k-means finds them. Figures 7.3a and 7.3b show the *Distributed Data-Reference* approach using data clustering for the MSA and MIF data structures, respectively.

The *Distributed Data-Reference* technique has two main advantages. The first advantage is no communication while indexing, which makes the indexing scalable. The second advantage is decreasing the dependency between the processes, which also increases the scalability of the search step.

7.2 Permutation-Based Indexing using MapReduce

MapReduce is a programming model for data intensive applications (discussed in detail in section 5.3.4). The input data is divided into small chunks and processed by *map* functions to generate intermediate *(key, value)* pairs. These pairs are grouped together with respect to their *key* to produce *(key, list(values))* tuples. The tuples are then passed to a group of *reduce* functions, which do some analysis. Each reduce function is responsible for a key or a group of keys.

The three above approaches can be implemented based on MapReduce. MapReduce is not efficient for online processing, due to the long time of initializing the mappers and the reducers. Hence, we use MapReduce only for the indexing process. In the next sections, we show how to implement the three above approaches using MapReduce.

7.2.1 MapReduce for Distributed Data Approach

The *Distributed Data* approach can be easily be implemented using the MapReduce model. Each *map* function handles a chunk of the data D_p. The *map* functions read its data chunk and emit a sequence of *(key, value)* pairs. The *key* is the object-id i as the approach is based on data division. The *value* depends by the work shared between the mappers and the reducers. We propose two techniques for the implementation of the *Distributed Data* approach using MapReduce, namely, with *Unshared workload* and *Shared workload*.

Unshared workload

In this model, the *value* is composed of the reference-id j and the position of the reference point r_j in the partial ordered list of this object $\tilde{L}(o_i, R)_{|r_j}$:

$$(K_m, V_m) = (i, (j, \tilde{L}(o_i, R)_{|r_j}))$$

Hence, the mappers are responsible to create the ordered list for each object and to segment it. The reducers receive the pairs to save them. The partitioning function (section 5.4.3) distributes the pairs based on the *key* (the object-id i). The output of the reducers is the partial data structure DS_p. Algorithm 7.6 and Algorithm 7.7 show the pseudo-code of the mapping and the reducing functions of the *Unshared workload* respectively.

Algorithm 7.6 Unshared workload: Mapping

IN: Key: D_p
Value: objects o_i
OUT: Key: Object-Id i
Value: $(j, \tilde{L}(o_i, R)_{|r_j})$
1. For each $o_i \in D_p$
2. Generate the partial ordered list $\tilde{L}(o_i, R)$
3. For each $r_j \in \tilde{L}(o_i, R)$
4. emit(i,$(j, \tilde{L}(o_i, R)_{|r_j})$)

Algorithm 7.7 Unshared workload: Reducing

IN: Key: Range of object ids i
Value: List of $< j, \tilde{L}(o_i, R)_{|r_j} >$
OUT: DS_p
1. For each $p \in < j, \tilde{L}(o_i, R)_{|r_j} >$
2. Store $(i, j, \tilde{L}(o_i, R)_{|r_j})$ in DS_p
3. emit(DS_p)

Shared workload

In the *unshared workload* algorithm, all the work is done by the mappers, and the reducers just organize the data. Hence, if we balance the work between the mappers and the reducers, we can achieve a better performance. Similar to *unshared workload*, the map functions emit a sequence of *(key,value)* pairs. The *key* of the map function is the object-id i and the *value* is the partial ordered list $\tilde{L}(o_i, R)$ related to this object:

$$(K_m, V_m) = (i, \tilde{L}(o_i, R))$$

Algorithm 7.8 shows the pseudo-code of the mapping function. The partitioning function is similar to the one used in the *unshared workload* algorithm. It divides the objects based on their ids. When the

reducers receive the data, they save the partial ordered lists $\tilde{L}(o_i, R)$ based on the used data structure. Algorithm 7.9 shows the pseudo-code of the reducing function.

Algorithm 7.8 Shared workload: Mapping

IN: Key: D_p
Value: objects o_i
OUT: Key: Object-id i
Value: ordered list $\tilde{L}(o_i, R)$
2. For each $o_i \in D_p$
3. Generate partial ordered list $\tilde{L}(o_i, R)$
4. emit(i,$\tilde{L}(o_i, R)$)

Algorithm 7.9 Shared workload: Reducing

IN: Key: object id i
Value: Partial ordered list $\tilde{L}(o_i, R)$
OUT: DS_p
1. For each $r_j \in \tilde{L}(o_i, R)$
3. Store $(i, j, \tilde{L}(o_i, R)_{|r_j})$ in DS_p
4. emit(DS_p)

Using the *shared workload* algorithm, the workload is balanced between the mappers and the reducers. The mappers are responsible to create the ordered lists and the reducers are responsible to organize and save them.

7.2.2 MapReduce For Distributed Reference Approach

In *Distributed Reference* approach, each process has two roles; *indexer* and *coordinator*. The analogy within the MapReduce model is that, the *indexer* are the *mappers*, while the *reducers* are the *coordinators*.

The input of a map function is the sub-data domain D_p. There, the data is distributed based on the references. Each map function emits $(key, value)$ pairs. The *key* is the reference id j, while the *value* is a combination of the object id i and the location of the r_j in the partial ordered list of the object $\tilde{L}(o_i, R)_{|r_j}$.

The partitioning function distributes the data based on the distribution of the references on the reducers. A pair $(j, (i, \tilde{L}(o_i, R)_{|r_j}))$ is sent to reducer j if $r_j \in R_p$ (since $r_j \in \tilde{L}(o_i, R)$).

On the reducer side, the pairs are organized and saved for further processing, based on the choice of the data structure used (eg. MSA and MIF).

7.2.3 MapReduce For Distributed Data-References Approach

The model is similar to the *Distributed Data* approach (section 7.2.1), but instead of indexing the objects based on the same reference list, the objects which belongs to the same data domain D_p are indexed using different reference sets R_p. Accordingly, each map function indexes its data domain using a unique set of references.

For the mappers, the *key* is the object id and the *value* can be $\tilde{L}(o_i, R_p)$ or $(r_{j_p}, \tilde{L}(o_i, R_p)_{|r_{j_p}})$. Hence, similar to the *MapReduce Distributed Data Approach*, the *Unshared workload* (algorithm 7.2.1) and *Shared workload* (algorithm 7.2.1) can be applied easily.

7.3 Distribution Taxonomy

Figure 7.4 shows the taxonomy of distributed permutation based indexing. As we are working on a distributed environment, the distribution is mainly based on the data division. The data division can be based on common references or independent references.

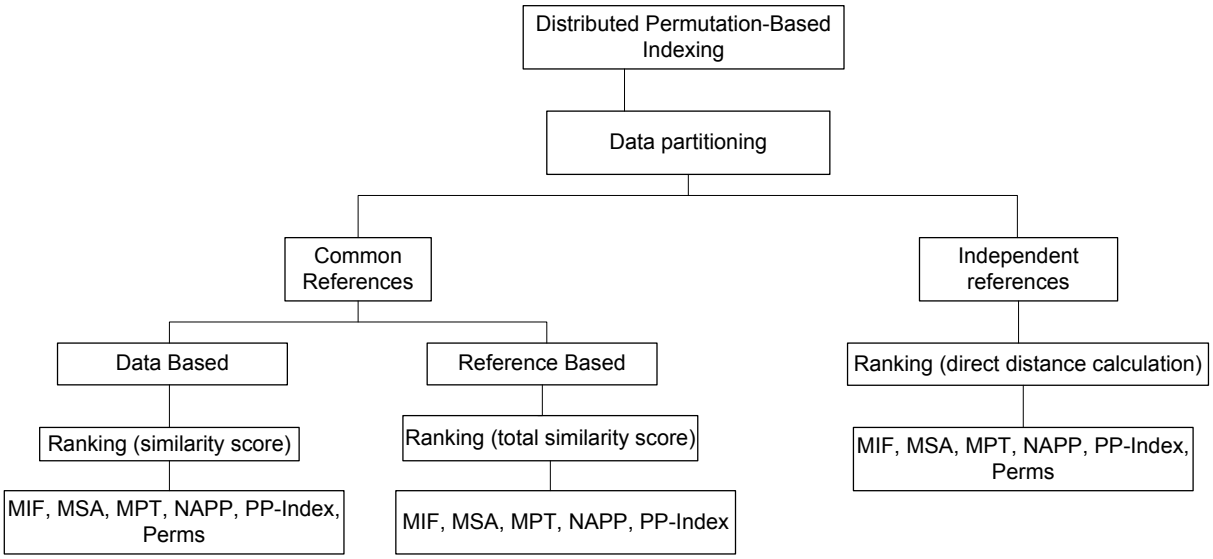


Figure 7.4: Distributed Permutation-Based Indexing Taxonomy

For independent references, each process indexes its portion using local reference points which are not shared with other processes. Accordingly in searching, the results should be ranked based on the direct distance calculation with the object. As, they were indexed using independent reference points.

For common references, the indexing is performed based on the data or the references. For data based technique, all the objects are indexed using the same reference list. That leads to distribute the data randomly between the nodes. While searching, all the nodes need to participate to answer the query. The objects can be ranked by their similarity score produced by each process as they share the same references. The reference based technique distributes the data based on the references. While searching, the processes which share the same references with the query are the processes that participate to answer the query. The objects are ranked with respect to the total similarity score, as each process contains a portion of the references.

7.4 Experimental Results

We have conducted large-scale experiments to evaluate the efficiency and the scalability of the three approaches in terms of recall (RE), position error (PE), indexing and searching time. We implemented the three approaches using MPI, Hadoop and MRO-MPI. Our experiments are conducted on 5-million objects from dataset DS_3 and DS_4. DS_4 is composed of visual shape features (21-dimensional) and DS_3 is composed of color features (84-dimensional). More details about the two datasets are available in section 2.6.3.

In section 7.4.1, we measure the average RE and PE for the three distributed approaches. The experiments were performed using the first dataset (21-dimensional) and three sets of references of sizes 1000, 2000 and 3000, that are selected randomly from the dataset. The average RE and PE are based on 250 different queries, which are selected randomly from the dataset.

In section 7.4.2, we measure the scalability of the three approaches. First, we compare the implementation of the three approaches using MPI in terms of indexing and searching time. The experiments were done using the first dataset (21-dimensional) and three sets of reference points of sizes 1000, 2000 and 3000, that are selected randomly from the dataset. The average searching time is based on 250 different queries, which are selected randomly from the dataset. The MSA data structure was used for these experiments.

Then, we compare the indexing implementation of the three approaches using MPI, MapReduce and MRO-MPI. The experiments were done using the second dataset (84-dimensional) and four sets of references of sizes 100, 500, 1000 and 2000, that are selected randomly from the dataset. The MIF data structure was used for these experiments.

All the evaluation including indexing and searching time is performed using the full permutation list, as we consider that using the full ordered list is the worst case scenario: whatever the approach is used, the search is done by all the processes. The experiments were performed on the *Coral Cluster* (detailed in appendix A.1).

7.4.1 Recall and Position Error

Figures 7.5 and 7.6 show the average RE and PE for 10 K-NN using 10 and 20 processes. Figures 7.7 and 7.8 show the average RE and PE for 100 K-NN using 10 and 20 processes.

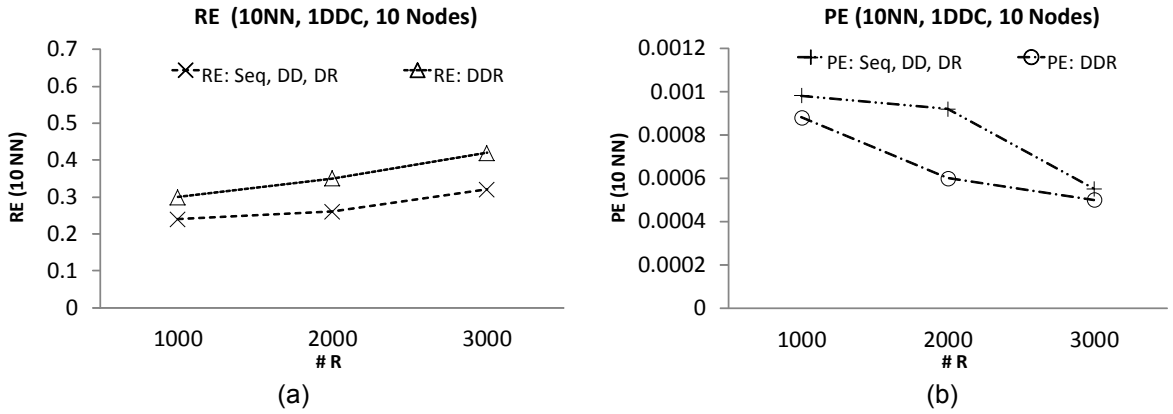


Figure 7.5: Average RE and PE (5 million objects, 21-dimensional, MSA, 250 queries) for 10 k-NN using different sets of reference points (1000, 2000 and 3000) using 10 processes.

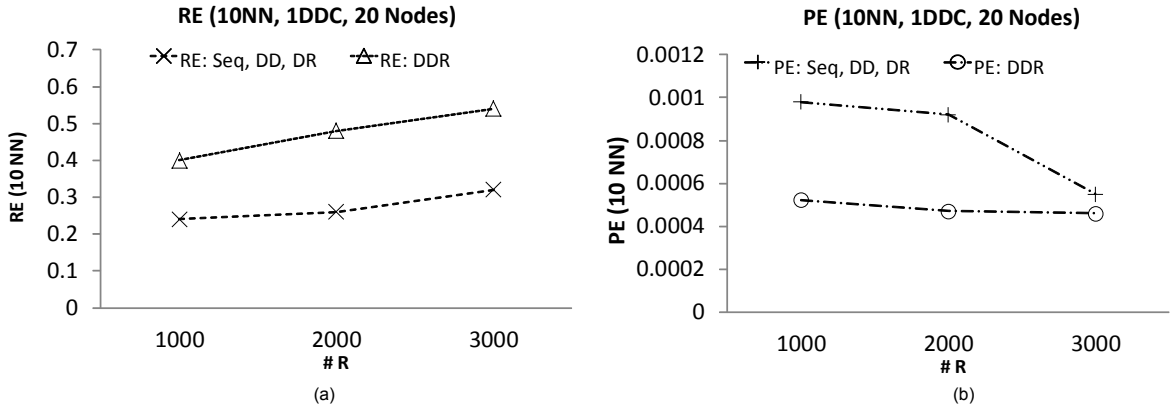


Figure 7.6: Average RE and PE (5 million objects, 21-dimensional, MSA, 250 queries) for 10 k-NN using different sets of reference points (1000, 2000 and 3000) using 20 processes.

The figures represent four different strategies, which are as follows. The first strategy is the *sequential (Seq)*, which represents the RE and PE values for the sequential algorithm. The second strategy is the *Distributed Data (DD)* approach. The third strategy is the *Distributed Reference (DR)* approach. The last strategy is the *Distributed Data-Reference (DDR)* approach. The *DDC* factor for the *DDR* approach is equal to 1 for the most fair comparison as the other approaches do not use the distance values to re-rank the objects.

The figures show that the *Seq, DD* and *DR* approaches give the same RE and PE. The number of

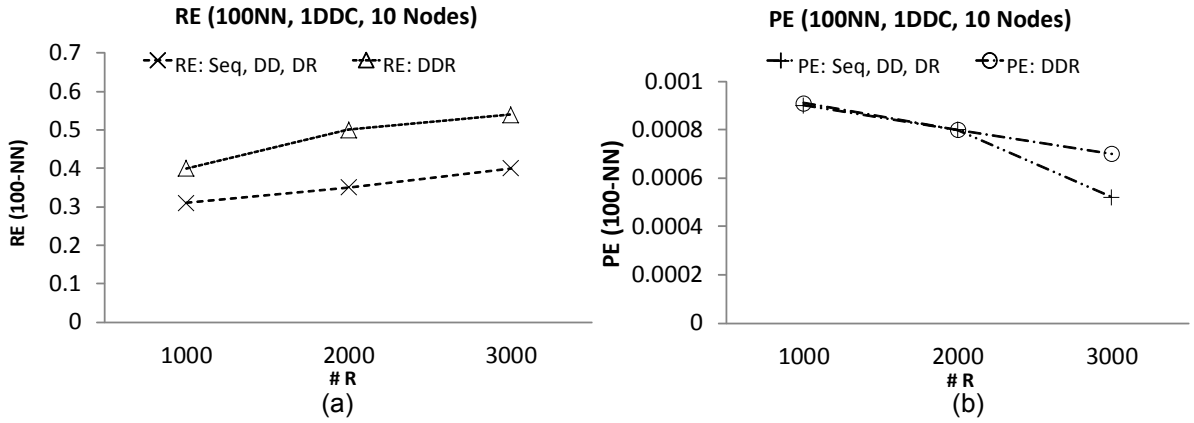


Figure 7.7: Average RE and PE (5 million objects, 21-dimensional, MSA, 250 queries) for 100 k-NN using different sets of reference points (1000, 2000, 3000) using 10 processes.

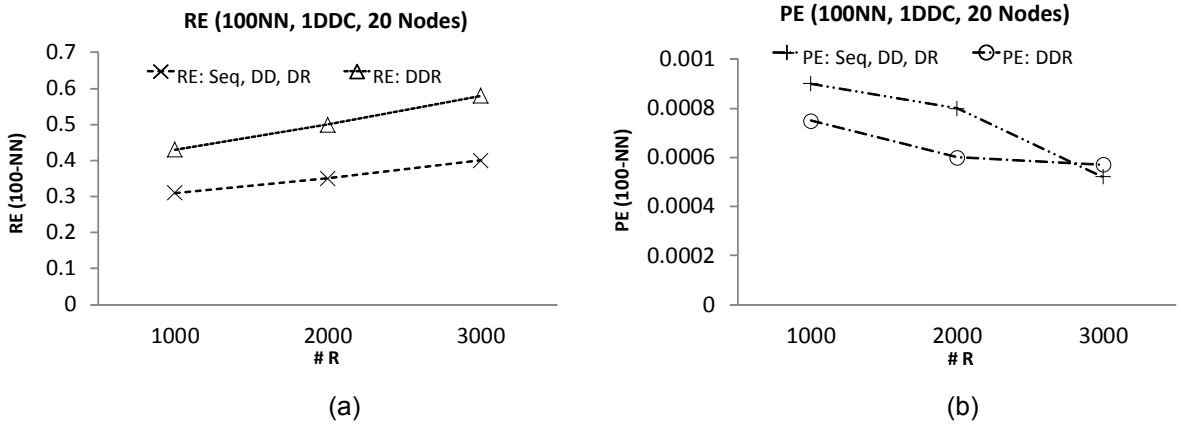


Figure 7.8: Average RE and PE (5 million objects, 21-dimensional, MSA, 250 queries) for 100 k-NN using different sets of reference points (1000, 2000, 3000) using 20 processes.

processes also does not affect their performance. This is expected as all the objects, whatever their domain is, are indexed using the same set of references. Hence, the *DD* and *DR* approaches give the same RE and PE as the sequential algorithm, whatever the number of processes.

The *DDR* approach gives a better RE and PE compared to the other three approaches (*Seq, DD* and *DR*). In addition, its efficiency increases when the number of processes increases. There are two reasons for that. First, the data domain is independent for each process, which helps decreasing the search space within each data domain. That makes the objects much better identified using local references. The second reason is the effect of direct distance calculation, between the query and the candidate list from each process. These two factors offer better ranking to the output results.

In addition, a better performance is achieved when the number of processes increases. As the number

of process increases, the number of objects assigned to each process decreases, which improves the locality and provides a better representation of the objects, leading to a better performance.

Figures 7.9 and 7.10 show the effect of the *DDC* factor used in the *DDR* approach for 10 and 100 k-NN, using 20 process and the same sets of references. It is clear from the figure that the RE increases and PE decreases when the *DDC* increases. Also, due to the data distribution, a better recall can be achieved with a low *DDC* value. For example, a recall of 0.9 is achieved with *DDC* equal to 4. On the other hand, when *DDC* increases, more objects are compared with the query per process, which affects the searching time (section 7.4.2).

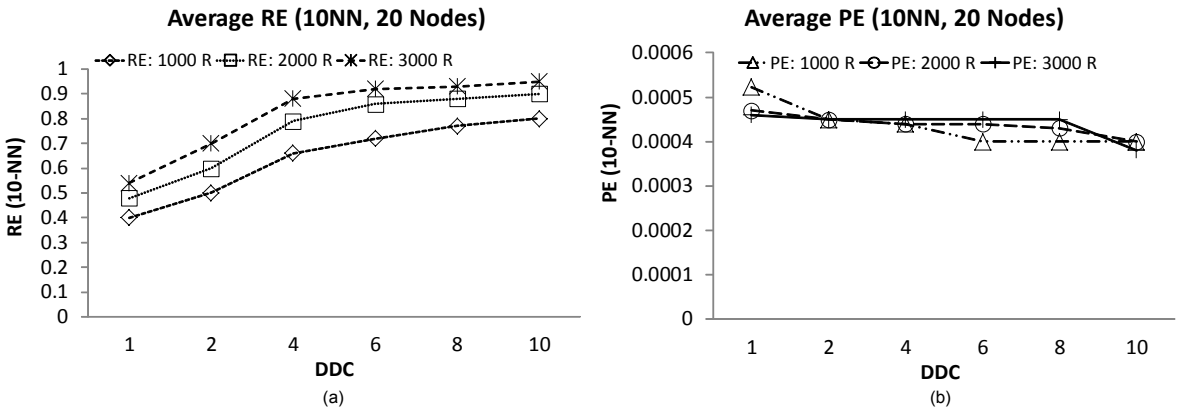


Figure 7.9: Effect of changing DDC for 10 k-NN using 20 processes (5 million objects, 21-dimensional, MSA, 250 queries).

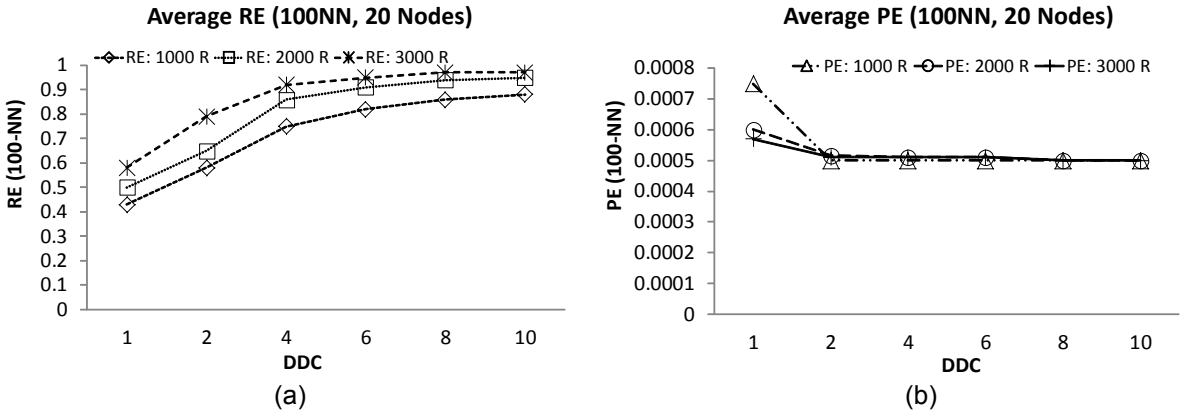


Figure 7.10: Effect of changing DDC for 100 k-NN using 20 processes (5 million objects, 21-dimensional, MSA, 250 queries).

7.4.2 Scalability

Indexing and Searching using MPI

Figures 7.11, 7.12 and 7.13 show the indexing and searching times using 5, 10 and 20 processes for the three distributed approaches presented in section 7.1 and implemented in MPI.

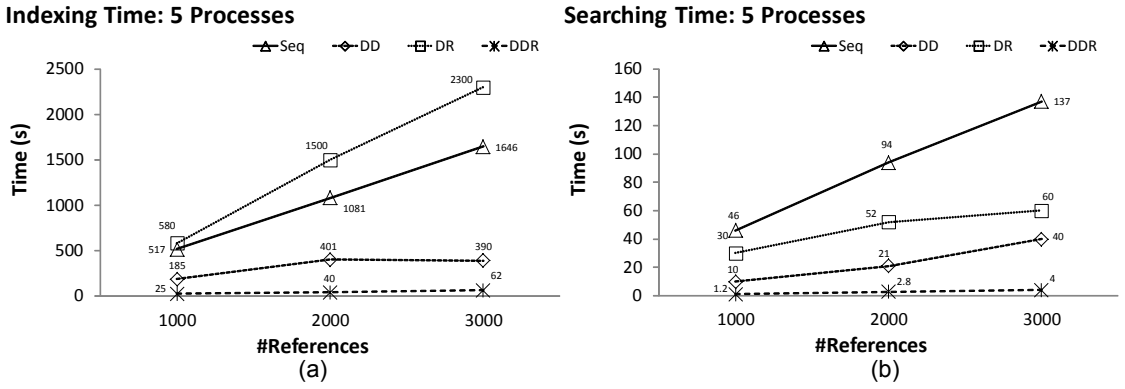


Figure 7.11: Indexing and Searching time (21-dimensional dataset, MSA) in seconds using 5 processes.

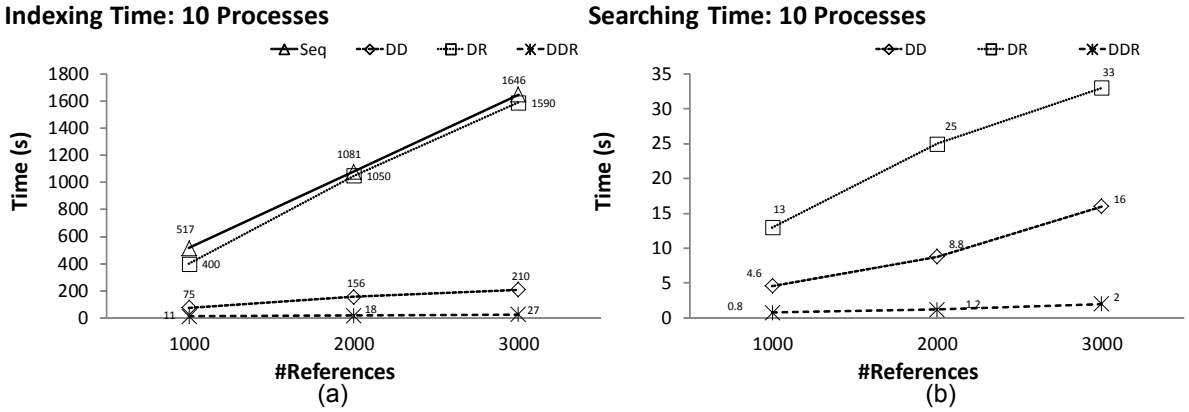


Figure 7.12: Indexing and Searching time (21-dimensional dataset, MSA) in seconds using 10 processes.

Indexing The x-axis shows the number of reference points and the y-axis shows the indexing time in seconds for the full ordered list $L(o_i, R)$. For the three algorithms, when the number of cores increases, the indexing time decreases. Also, when the number of reference points increases the indexing time increases. The *DR* approach gives the longest indexing time due to the communication required between the processes, in order to exchange the data. Using 5 processes, the *DR* approach is slower than the sequential approach, since most of the time is consumed in exchanging the data between the processes.

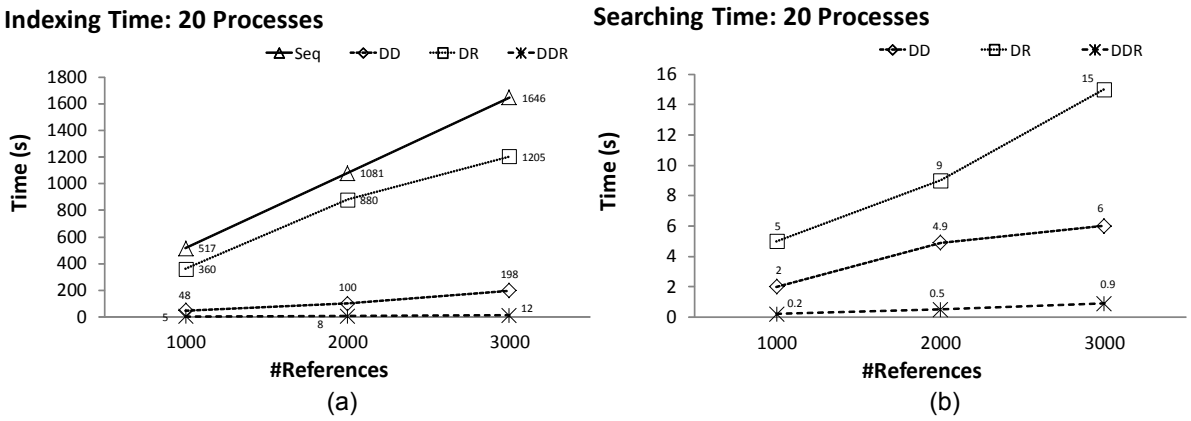


Figure 7.13: Indexing and Searching time (21-dimensional dataset, MSA) in seconds using 20 processes.

Searching For the three algorithms, when the number of process increases, the searching time decreases, with the same RE and PE for the *DD* and *DR* approaches and with a better RE and PE for the *DDR* approach (DDC=1). Also, when the number of reference points increases the searching time increases.

Using the *DR* approach, searching is very slow compared to the other two approaches. The main reason for that is the gathering process. In *DR*, all the nodes participate to answer user's query, and each node has $\frac{n}{P}$ references, where each reference is responsible for N objects, as we use the full ordered list. Hence, if there are 20 processes and 20 reference points, the broker process receives 20 lists of size N. After that, these lists are combined and sorted. The combination process is difficult, as the similarity with respect to the objects is done using only a portion of the references, and not the full references. Hence, this reduces the running time, which means that most of the time for the *DR* is consumed in gathering the data to give the final output to the user.

The *DDR* searching algorithm gives the best running time, although there is some time overhead due to the hard disk access and the direct distance calculation between the query and the top K_c objects. In reality, the effect of these two steps diminish when the number of the processes increases as they are done in parallel with a very small portion of the dataset.

Figure 7.14 shows the running time in seconds for different *DDC* and k-NN with respect to different reference points using 20 processes. When the DDC value increases the searching time increases but with a small difference due to the distribution of the data between the processes.

Indexing using MapReduce

We have implemented the three approaches using MRO-MPI and Hadoop and compared their performance to that of the MPI implementation. The MIF data structure and the 84-dimensional dataset were

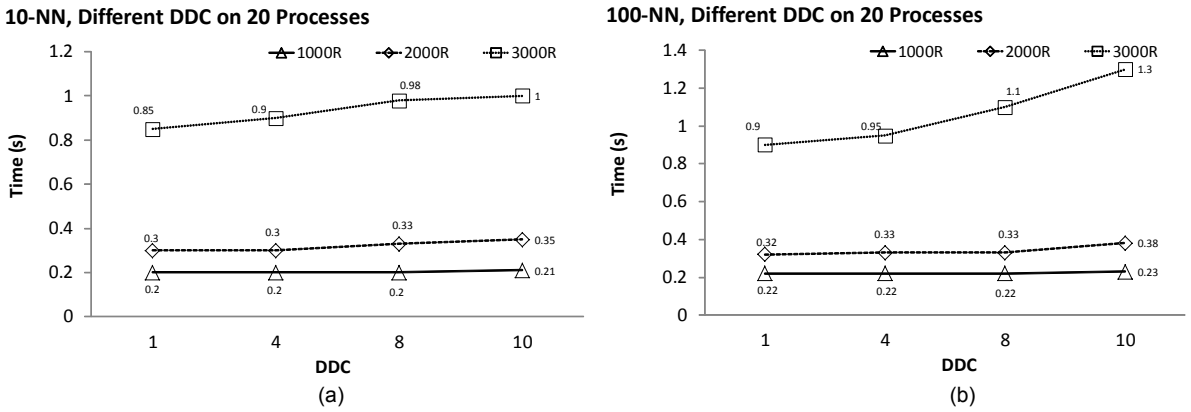


Figure 7.14: Effect of changing the DDC on the searching time (21-dimensional dataset, MSA). The time is in seconds using the full ordered list.

used using four sets of references of sizes 100, 500, 1000 and 2000 out of 5-million objects (database DS_4). Figures 7.15, 7.17 and 7.18 show the indexing time for the three approaches *DD*, *DR* and *DDR* respectively, using MRO-MPI, Hadoop and MPI with 10 and 20 processes. In MRO-MPI, the number of mappers is equal to the number of reducers and is equal to $\frac{P}{2}$. For Hadoop, the number of processes is the number of reducers as the number of mappers are defined by Hadoop.

For the *DD* and the *DDR* approaches (Figures 7.15 and 7.18), we can see that the *shared workload* algorithm is faster than *unshared workload* algorithm. There are two reasons for that. The first reason is the rate of emitting the data. In the *unshared workload* algorithm, at every emission, the map function emits the object-id, the reference-id and the position of the reference point in the ordered list of the object. If we have 1000 objects and 10 reference points, the mapping function emits 10'000 pairs. In contrast, for the *shared workload* algorithm the mapping function emits the object-id and the ordered list. If we have 1000 objects and 10 reference points the mapping function emits 1'000 pairs only. Figure 7.16 shows the average output of each mapping function in gigabytes for the two algorithms using the *DD* approach. As we can see, the average output of each mapping function for *shared workload* is for less than the average output of the *unshared workload* algorithm.

The second reason for this difference in the performance is the way of organizing the work between the mappers and the reducers. In *unshared workload*, all the work is done by the mappers and the reducers only save the received pairs. In contrast, for the *shared workload* algorithm, the work is better balanced and that decreases the running time.

Hence, the performance of the MapReduce implementation is affected by the way of choosing the $(key, value)$ pairs, the way of dividing the work between the mappers and reducers and the number of running mappers and reducers. For the two algorithms, when the number of processes increases the average output size decreases due to decreasing the number of processed objects per process.

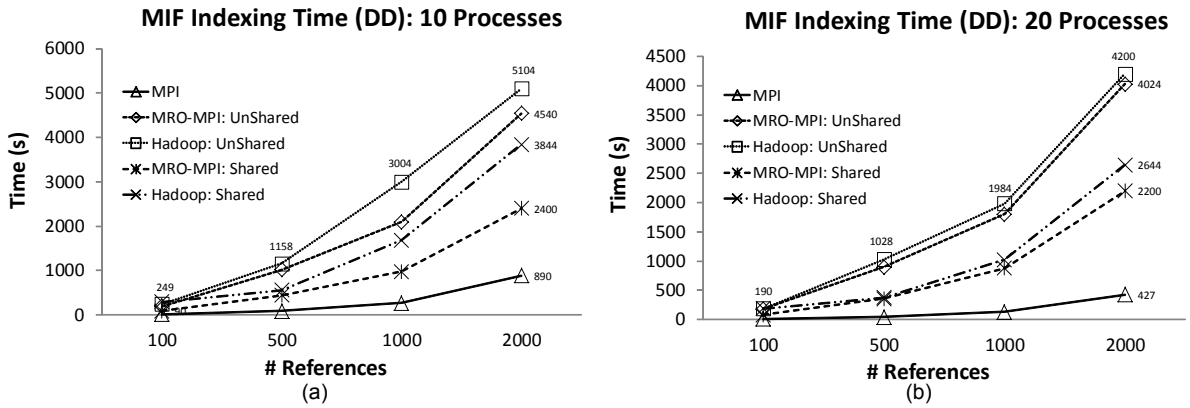


Figure 7.15: Indexing time in seconds for DD (84-dimensional dataset, MIF).

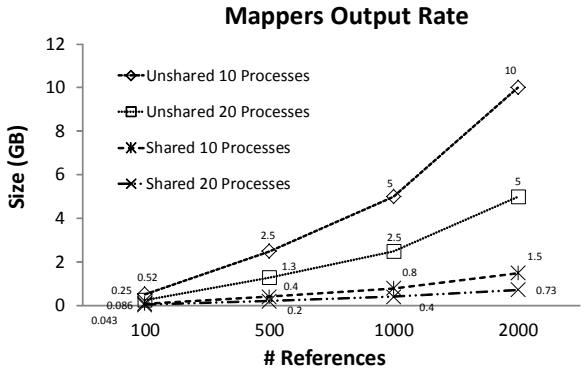


Figure 7.16: Output data size in GB DD.

The *MPI* implementations of the three approaches are much faster than the MapReduce implementations, whatever the approach which is used. The reason is that with MPI the data is sent directly to the responsible process, while for MapReduce the data is grouped into pairs and tuples to be sent to the responsible reducers, which consumes time. At the same time, MPI misses the simplicity. More time and experience is required for coding the approaches using MPI than using MPI-MapReduce. So, MapReduce is useful and makes the parallel programming an easy process, but it is not suitable for all applications, also we can get better performance using regular MPI.

When comparing MapReduce and MRO-MPI, the figures show that the MRO-MPI implementations of the two algorithms are much faster than the Hadoop implementation, thanks to the overlapping which detailed in section 5.4.2.

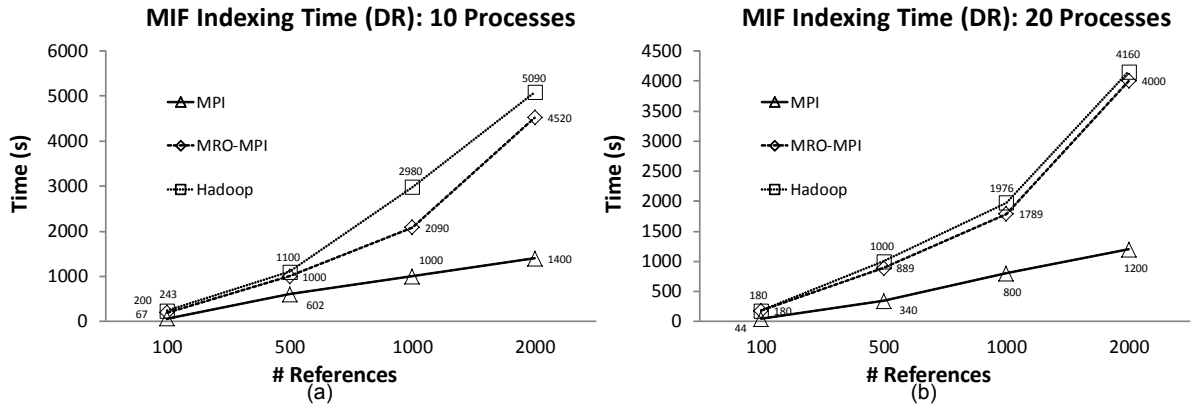


Figure 7.17: Indexing time in seconds for DR (84-dimensional dataset, MIF).

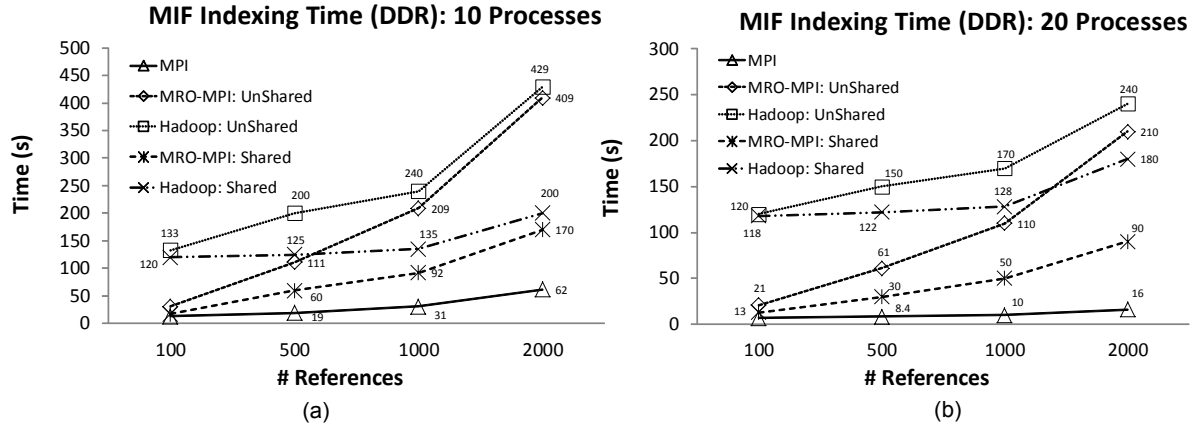


Figure 7.18: Indexing time in seconds for DDR (84-dimensional dataset, MIF).

7.5 Summary

The multicore architectures including CPU and GPU are not the best solutions due to the limitation of the memory size and the architecture. A distributed architecture is most appropriate to handle the rapid increase in the size of the data to be indexed.

In this chapter, we propose three approaches for distributed permutation-based indexing, which can be adapted on any data structure based on permutation-based indexing. The first approach is *Distributed Data*. The main idea is to distribute the data on the processes based on the number of running processes. Each portion of the data is indexed using the same full reference list. When searching, the submitted query is sent to all the processes, in order to answer a query since the data is distributed randomly. That means this approach does not allow efficiently inter-query parallelism.

To overcome that, the second approach *Distributed References* aims to distribute the data based on

the references. Each process has a partial reference set. Each reference is responsible for a group of objects located close to this reference set. Once a query is submitted, the search is done through the shared references with the query. All the processes do not need to participate to answer one query. The main bottleneck of this technique is the communication time during indexing and the results gathering time after searching.

Hence, we finally proposed a third technique called *Distributed Data-References*. The main idea is to limit the dependency between the processes. The data is divided into portions. From each portion, a number of reference points is selected independently and not shared with the other processes. Then, each process indexes each portion independently. Once a query is submitted, the search is performed within the portions that share the references listed in the query ordered list. The results from each portion cannot be combined directly, because the objects are indexed within each portion using different sets of references. Hence, a direct distance calculation between the query and the top elements from each process is performed to rank objects.

We have applied the three algorithms on two different data structures (MSA and MIF) using two different datasets of two different dimensions (21 and 84 dimensional). The implementations was done using MPI and MapReduce (Hadoop and MRO-MPI). We show that the *Distributed Data-References* approach is the best technique in terms of indexing time, search time, recall and position error. Also, we showed that the MPI implementation is faster than the MapReduce implementation. Also, we showed that the MRO-MPI gives a better performance than the normal MapReduce model.

In summary, decreasing the dependency between the data and references improves the efficiency of searching, the indexing time and the searching time. In addition, the MPI implementation is faster as there is no need to gather the data in a certain way in order to be processed, as in the MapReduce model. At the same time, MPI misses the simplicity. The choice is therefore left to a trade-off between simplicity and speedup.

Chapter 8

Indexing and Searching Big Datasets

In this chapter, we show our experience with building distributed large scale similarity search system for big databases. Our evaluation is performed on two large public datasets, which are the CoPhIR dataset [92] and the BIGANN dataset [124]. The two datasets are described in details in section 2.6.3 (DS_5 and DS_7 in chapter 2). The two datasets are indexed using the MPT data structure that was discussed in chapter 4. In section 8.1, we show the distributed design of our system to handle large scale data. In section 8.2, we show the evaluation of indexing the full CoPhIR dataset on distributed environment. In section 8.3, we show the evaluation of indexing the full BIGANN dataset on the distributed environment.

8.1 Distributed System Design

One of the main issues in any distributed system is how the computing nodes are working together, in order to efficiently handle users requests. We have a computer cluster of 20 computing nodes. Each node has two cores, 8 GB of memory and connected to 512 GB Hard-Disk. The nodes are connected via a 1 GB Ethernet connections. Hence, in total we have 40 cores, 160 gigabytes of memory and 10 TB of storage. Beside this local storage, we have a shared storage between all the nodes of the cluster of size 3.6 TB. Process 1 in node 1 is our broker process. This is the process, that receives our requests and gather the results from the workers. Figure 8.1 shows how the structure of the nodes.

Our MPT distributed algorithm follows the *Distributed Data-Reference* approach that was discussed in details in section 7.1.3. The algorithm is written in C++ and MPI standards [109] using MPICH 1.4.1p1 [125] library. As discussed in chapter 5, MPI supports virtual operations. We can therefore create as many processes as needed. However, that affects the performance especially if the number of processes is larger than the number of cores. For optimal usage of the cores, we create 40 processes at maximum. This means that process go by pairs that share the same memory and the same hard disk.

Based on the operation (indexing or searching), the computing nodes act in two different ways, which are discussed in details in the next section.

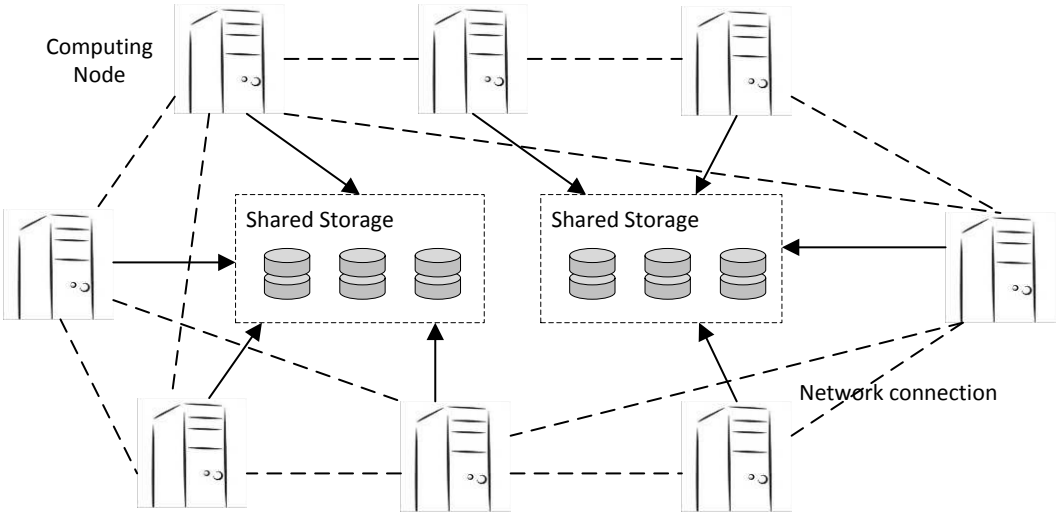


Figure 8.1: Cluster design

8.1.1 Indexing

Based on *Distributed Data-Reference* (section 7.1.3), the processes do not need to communicate while indexing. Once the indexing request is received, all the nodes, including the broker process, read a portion of the data. The portion for each process is equal to $N_l \backsimeq \frac{N}{P}$. The range for each process is identified by the rank of the process multiplied by N_l. At the end of the indexing process, the processes synchronise.

As detailed in section 7.1.3, the *Distributed Data-Reference* approach needs direct distance calculation between the query and the top objects in the candidate list. This is due to indexing the objects with respect to different sets of reference point by each process. As we are working with big data, the objects data can not be accessed through the shared storage by multiple processes at the same time while searching. This would affect the searching performance.

To efficiently access the data, a local copy is obtained from the shared storage into the local storage of each process. The local database is then divided into small chunks. Each chunk contains a maximum of c objects. In the memory, we have a lookup table, which saves l tuples. The tuples are composed of two values. The first value is the chunk name and the second value is the starting index of the chunk. For example, if we have a file of 30 objects and we divide it into 3 chunks of sizes 9, 10 and 11. The starting index of the first chunk is 0, the starting index of the second chunk is 9 and for the third chunk is 19. Hence, if we need to access object 25, we access chunk 3 directly without the need to access the full file. That helps to reduce the random disk access. Once the indexing process is finished, the system waits for users requests.

8.1.2 Searching

Figure 8.2 shows how the processes act in the searching operation. The broker process is responsible to receive query requests. After generating the query ordered list, the active processes are defined. Theses active processes are notified and they search locally through their own MPT data structure.

After getting the candidate list, the lookup tables are checked and the related data chunks are accessed. Then the distance between the query objects and these objects in the candidate list are computed and sent to the broker. The broker sorts the objects based on the distances and sends the results to the user.

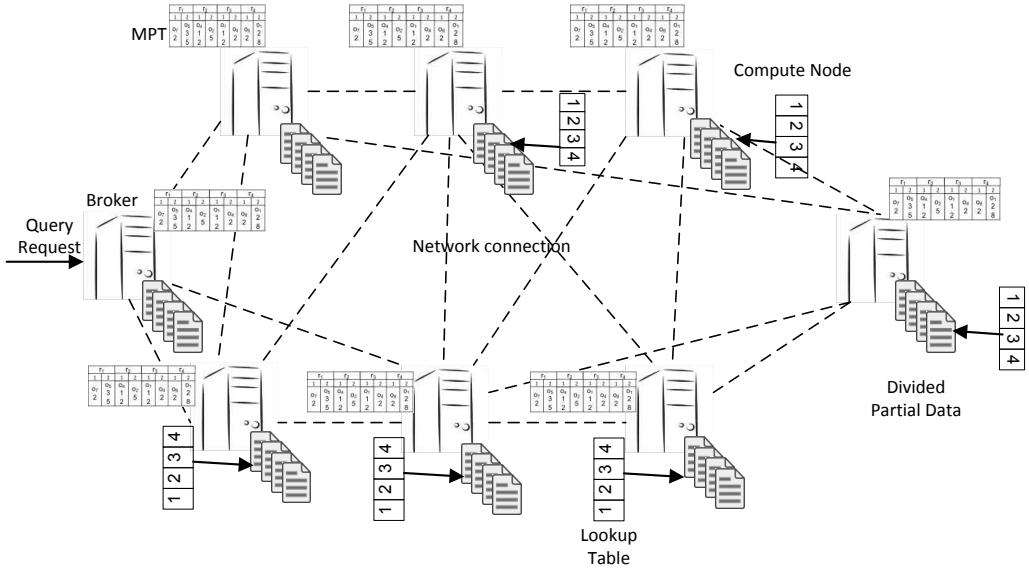


Figure 8.2: Distributed Searching

8.2 CoPhIR

In this evaluation, we have indexed the full CoPhIR dataset (106 Million objects, 280 dimensions) using 34 processors. Each process has $N_l = 3'117'648$ objects and these objects are divided into chunks of size $50'000$ (63 chunks). The number of reference points $n = 2040$, which means that each process indexes its own data portion using 60 reference points. These reference points were selected using the *dense locality* approach that was discussed in chapter 3. The number of nearest reference points $\tilde{n} = 40$. The bucket size $B = 10$. The direct distance calculation factor is $DDC = 10$.

Using a very low number of reference points and a low DDC value, we are still able to achieve a high recall. The average recall for 100 queries, for k-NN $= 1, 10, 100$ is 0.9, 0.6 and 0.71 respectively.

For the indexing time, as the references set is divided between the processes, we are able to achieve

a fast indexing time. Indexing 106 million objects was performed in 8 minutes. The average memory allocated per processor is 300 MB.

The average searching time for 100 queries, k-NN $= 1, 10, 100$ is 2.2s, 2.2s and 2.5s respectively. Note that this searching time could still be optimised (reduced) since, here the data is divided randomly, the searching performance was not that efficient. Each process is responsible for a group of objects, with no relation between them. Hence, when the broker process defines the nearest references to an object, these references might be related to different processes. More communication time is needed in order to send the query to and gather the results from these active processes.

In our evaluation, the average number of processes that participated to answer a query is 10. That means the broker needs to communicate twice (send the query and gather the results to and from the active processes) with at least 10 processes to answer each query. Hence, most of the time is consumed in communicating with the processes.

8.3 BIGANN

In this evaluation, we have indexed the full BIGANN dataset (1 Billion object, 128 dimensional vector, gathered by H. Jégou [124]) using 34 processors, the number of objects that is assigned to each object is $29'411'764$. The number of reference points $n = 7990$. Each processor indexes its partition using 235 reference points. The number of nearest reference points is $\tilde{n} = 20$. Each processor has its local portion divided into chunks of size $50'000$. Each process is then responsible for 588 chunks. The bucket size $B = 10$. The direct distance calculation factor is $DDC = 10$.

The average recall for 100 query, for k-NN $= 1, 10, 100$ is 0.6, 0.42 and 0.5 respectively. The indexing time for 1 billion object is 1 hour. The average memory allocated per processor is 1 GB. The average searching time for 100 query, k-NN $= 1, 10, 100$ is 4s, 5s and 8s respectively. Similar to the CoPhIR dataset, the data is distributed randomly. That leads to high communication time while searching. The average number of active processes per query is 16.

For the CoPhIR and the BIGANN datasets, a better searching performance could be achieved if the data is distributed between the processes in balanced clusters. Clustering the data sequentially is not feasible due to the large size of the datasets. A distributed clustering algorithm could be applied efficiently of the data [126, 127, 128]. We believe that using a pre clustering step gives a fast response time as the number of nodes that will participate to answer the queries will be reduced.

8.4 Summary

In this chapter, we showed our experience with building distributed large scale similarity search system for big databases. Our *Distributed data-reference* approach was able to achieve good performance in

terms of indexing and recall with a low number of reference points, with an acceptable searching time. However, we can expect a better response time. The main gain in the searching time would be to find an alternative to the random distribution of the data over the processes. In our case, the number of active processes per query was high. Hence, most of the searching time was consumed in sending the query and gathering the results. To achieve a good response time, each process has to be responsible for a data domain in order to decrease the number of process that participate to answer the query request. This can be achieved by clustering the data into balanced clusters using a distributed data clustering algorithm.

Chapter 9

Conclusions and Future work

The similarity search problem translated into the $k-$NN problem finds many applications for information retrieval, machine learning and data mining. Approximate similarity search relaxes the search constraints to get acceptable results at a lower cost (computation, memory, time). One recent technique for approximate similarity search is based on encoding the data using the surrounding objects. This encoding technique is applied in many domains, such as machine learning, indexing and searching. The technique is known in the indexing domain as permutation-based indexing. A set of pivots (reference points) is selected and the database objects are represented by these lists of the closest reference points ordered with respect to their distances from the object (ordered lists).

The contribution of the thesis can be classified as theoretical and technical. In terms of theoretical contribution, we propose a geometrical analysis for the permutation-based indexing model, novel efficient algorithms and data structures for managing the ordered lists and efficient multicore and distributed algorithm for indexing and searching. As technical contributions, our algorithms have been developed, implemented and evaluated on large-scale datasets using the recent technologies such as OpenMP, CUDA (GPU), MapReduce and MPI.

9.1 Summary and Achievements

In the first part of the thesis, we described our perspective on the encoding model of permutation based indexing based on the Voronoi diagrams. We showed that using one reference point can be considered as a Voronoi diagram, where each Voronoi cell contain number of objects which are the closest objects. For more than one reference point, the encoding model can be related to the construction of an Ordered k-Order Voronoi Diagram, such that the order of the Voronoi diagram is defined by the number of reference points. We also provided two strategies to select the reference points based on the distribution of the data. These strategies are *global locality* and *dense locality* approaches. The main difference between them is that the *dense locality* approach better adapts to the density of original data. We evaluated our proposed

strategies on synthetic and real datasets compared to the recent reference points selection techniques. The evaluation was performed by measuring the number of objects around each reference point. Our proposed approaches gave the best performance compared to recent selection techniques. The performance of the *dense locality* approach is similar to the *global locality* approach for uniform datasets. For the clustered datasets, the *dense locality* approach gives better performance compared to the *global locality* approach.

To handle permutation lists, we proposed two novel data structures for organizing the permutation lists in an efficient way. These data structures are the *Metric Suffix Array* (MSA) and the *Metric Permutation Table* (MPT). The MSA is based on the suffix array data structure. The main idea is to concatenate the permutation lists for the database objects to form a string and build a suffix array from this string. The algorithm is efficient in terms of searching time. However, a lot of time is consumed for indexing and this strategy is not scalable due to memory limitation. To overcome that, we proposed the *Metric Permutation Table* (MPT). It provides an enhanced organization of the permutation lists for efficient indexing and searching. The MPT is based on inverted files, data quantization and data compression. We evaluated our proposed data structures on large scale datasets of millions of objects. Our proposed structure showed that it gives efficient performance compared to the recent techniques for approximate similarity search.

The availability of multicore architectures (CPU and GPU) gives a great motivation to adapt the available algorithms to parallel architectures. We proposed multicore (CPU and GPU) strategies for permutation-based indexing. We evaluated the performance of the permutation based indexing approach on multicore architectures through two algorithm for multi-core CPU and three algorithms for GPU. For multi-core CPU, we found that the rate of accessing the hard disk and the memory allocated for each thread affects the speed up of permutation-based indexing. For GPU, we found that the direct mapping of the algorithm on the GPU is not efficient. This is due to the excessive number of sorting operations that should be done on the GPU and also to the architecture of the GPU. Nevertheless, with algorithm optimisation, we managed to achieve a modest speedup. We evaluated the algorithms on large scale data sets of million of documents. From the conducted experiments we believe that the multi-core CPU is more suitable than the GPUs for the permutation based indexing.

To cope with the mass production of data, distributed computing is demanded. We proposed three distributed strategies for permutation-based indexing for indexing and searching. The first strategy is to distribute the data based on the reference points. The second strategy is to distribute the data based on the data objects. The third strategy is to distribute the data based on the data and the reference points. Experiments showed that the approach based on distributing the data and the references gives the best performance compared to the other two approaches in terms of indexing, searching, recall and position error, essentially thanks to a lower communication time between the nodes and increase the locality of the object with respect to each distributed dataset. The three algorithms were implemented using MPI and MapReduce. The MPI implementation is more efficient than the MapReduce implementation, because

the processes communicate directly to each other without grouping the data as in the MapReduce model. However, the MapReduce implementation is much simpler than the MPI implementation.

To improve the performance of MapReduce, we proposed an enhanced MapReduce model based on MPI. Our model is based on running the *map* and the *reduce* functions concurrently in parallel by exchanging partial intermediate data between them in a pipeline fashion using MPI. At the same time, we maintain the usability and the simplicity of MapReduce. We evaluated our proposed algorithms on three different applications (WordCount, Distributed Inverted Indexing and Permutation-Based Indexing). The results show a good speedup compared to the earlier versions of MapReduce such as Hadoop and the available MPI-MapReduce implementations.

Finally, we showed our experience with building a large-scale similarity search system using our proposed algorithms and data structures. Our system was evaluated through two big datasets of millions and billions of objects.

9.2 Perspective and Future work

We believe that the methodology of encoding the database objects based on the surroundings is an efficient technique that can be easily applied in a variety domains. In the thesis, we tried to give a geometric guideline on how the problem can be understood. Based on this geometric justification, we managed to derive polices for the selection of the reference points. Nevertheless, there are still many open issues that can be studied based on the work that we proposed.

Our model can be extended to form a mathematical model that is based on the statistics of the data, which could allow to a better choice of the reference points based on the dataset. Also, an important factor that need to be studied is the amount of information that we lose when we move from the distance domain to the ranking domain. Studying this approximation should be done based on the number of reference points, the nearest reference points and the distribution of the dataset. That could allow to justify the efficiency of the algorithm in terms of recall and position error.

In terms of multicore architecture, we believe that the performance of the algorithm on GPU needs more investigation. This can be done by testing the quality of the proposed algorithms on more powerful GPUs and using multiple GPUs that run concurrently. In addition, the effect of the programming model on the performance should be investigated. A comparative study should be conducted to see which programming model is better (CUDA, OpenCL, OpenACC). Also, a hybrid multicore implementation is worth investigating. This would run multi-core CPU and GPU at the same time and measure the difference in performance.

In terms of distributed environment, we believe that we proposed an extensive study on the validity of the permutation-based indexing in distributed environment. However, more advanced techniques could be investigated such as hybrid programming models. Studying the effect of using distributed

environments, multicore CPU and GPUs at the same time is worth investigating. The overhead of using all of these parallel environments concurrently affects the performance. In addition, the effect of inter-query parallelism on the response time should also be studied.

In general, to the best of our knowledge, this thesis is the first attempt to propose a model, multicore and distributed indexing and searching algorithms for permutation-based indexing. The work and the achievements in this thesis can be summarized in two points. The first point is studying the permutation-based indexing model for large-scale similarity search on single, multicore and distributed environment. The second point is to prove the validity of our proposed approaches through large scale public data. We believe that this work is a new step forward in the large-scale similarity search domain.

Appendix A

Appendix

A.1 System Specification

Octopus: a machine holds 8 Cores (2.3GHz), 8MB Cache, 32GB of memory and linked with TeraByte storage capacity (SATA).

Squid: a machine holds 32 Cores (2.70GHz) Intel Xeon processor, 20MB Cache, 128GB RAM and linked with 512GB (SATA Hard Drive) and 265GB (SSD Disk) storage capacity.

Coral Cluster: a Linux cluster of 20 DualCore nodes (40 cores in total). Each node has 3.00GHz Intel Core Duo CPU, 6MB Cache, 8GB RAM and linked to 512GB (SATA Hard Drive) storage capacity.

GPU Machine: a machine holds 4 cores (3GHZ) Intel i7 processor, 4MB L3-Cache, 16GB RAM, NVIDIA NVS 5200M (96 CUDA Cores, 2 Streaming Multiprocessor, 1 GB *global memory*), and linked with 512GB storage capacity (SSD Disk).

A.2 Publications

- Mohamed, H., & Marchand-Maillet, S. (2015). Scalable Indexing for Big Data Processing. Book chapter in Big Data: Algorithms, Analytics, and Applications (Chapman & Hall/CRC Big Data Series)CRC Press, Taylor & Francis Group, USA.

- Mohamed, H., & Marchand-Maillet, S. (2014). Quantized Ranking for Permutation-Based Indexing. In Journal of Information Systems (submitted).

- Mohamed, H., Osipyan, H., & Marchand-Maillet, S. (2014). Multi-Core (CPU and GPU) For Permutation-Based Indexing. In 7 th International Conference on Similarity Search and Applications (SISAP), Los Cabos, Mexico.

- Mohamed, H., & Marchand-Maillet, S. (2014). MRO-MPI: MapReduce overlapping using MPI and an optimized data exchange policy. In Journal of Parallel Computing.

- Mohamed, H., Osipyan, H., & Marchand-Maillet, S. (2014). Fast Large-Scale Multimedia Indexing and Searching. In Content-based Multimedia Indexing (CBMI'2014), Klagenfurt, Austria.

- Mohamed, H., & Marchand-Maillet, S. (2013). Quantized Ranking for Permutation-Based Indexing. In 6 th International Conference on Similarity Search and Applications (SISAP), Coruna, Spain.

- Mohamed, H., & Marchand-Maillet, S. (2013). Permutation-Based Pruning for Approximate K-NN Search. In 24th International Conference on Database and Expert Systems (DEXA), Prague, Czech Republic.

- Mohamed, H., & Marchand-Maillet, S. (2013). Metric Suffix Array For Large-Scale Similarity Search. In ACM WSDM 2013 Workshop on Large Scale and Distributed Systems for Information Retrieval, Rome, Italy.

- Mohamed, H., & Marchand-Maillet, S. (2012). Enhancing MapReduce using MPI and an optimized data exchange policy. In ICPP 2012: The 41st International Conference on Parallel Processing, (P2S2 Workshop), Pittsburgh.

- Mohamed, H., & Marchand-Maillet, S. (2012). Distributed Media indexing based on MPI and MapReduce. In Content-based Multimedia Indexing (CBMI'2012), Annecy, France.

- Mohamed, H., & Marchand-Maillet, S. (2012). Parallel Approaches to Permutation-Based Indexing using Inverted Files. In 5th International Conference on Similarity Search and Applications (SISAP), Toronto, Canada.

- Mohamed, H., von Wyl, M., Bruno, E., & Marchand-Maillet, S. (2011). Learning-Based Interactive Retrieval in Large-Scale Multimedia Collections. In Adaptive Multimedia Retrieval. Large-Scale Multimedia Retrieval and Evaluation. Lecture Notes in Computer Science, Volume 7836.

- von Wyl, M., Mohamed, H., Bruno, E., & Marchand-Maillet, S. (2011). A Parallel Cross-Modal Search Engine over Large-Scale Multimedia Collections with Interactive Relevance Feedback. In Demo at ACM International Conference on Multimedia Retrieval (ACM-ICMR'11), Trento, Italy.

Bibliography

[1] Hisham Mohamed and Stephane Marchand-Maillet. Quantized ranking for permutation-based indexing. In Nieves Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications*, volume 8199 of *Lecture Notes in Computer Science*, pages 103–114. Springer Berlin Heidelberg, 2013.

[2] Hisham Mohamed and Stéphane Marchand-Maillet. Permutation-based pruning for approximate k-nn search. In *In 24th International Conference on Database and Expert Systems (DEXA(1)), Prague, Czech Republic*, pages 40–47, 2013.

[3] Hisham Mohamed and Stephane Marchand-Maillet. Metric suffix array for large-scale similarity search. In *ACM WSDM 2013 Workshop on Large Scale and Distributed Systems for Information Retrieval*, Rome, IT, February 2013.

[4] Hisham Mohamed and Stéphane Marchand-Maillet. Mro-mpi: Mapreduce overlapping using mpi and an optimized data exchange policy. *Parallel Computing*, 39(12):851–866, 2013.

[5] Hisham Mohamed and Stéphane Marchand-Maillet. Enhancing mapreduce using mpi and an optimized data exchange policy. In *ICPP Workshops*, pages 11–18, 2012.

[6] H. Mohamed and S. Marchand-Maillet. Distributed media indexing based on mpi and mapreduce. In *Content-Based Multimedia Indexing (CBMI), 2012 10th International Workshop on*, pages 1–6, June 2012.

[7] H. Mohamed, H. Osipyan and S. Marchand-Maillet. Fast large-scale multimedia indexing and searching. In *Content-Based Multimedia Indexing (CBMI), 2014 12th International Workshop on*, pages 1–6, 2014.

[8] Hisham Mohamed and Stephane Marchand-Maillet. Parallel approaches to permutation-based indexing using inverted files. In *5th International Conference on Similarity Search and Applications (SISAP)*, Toronto, CA, August 2012.

[9] Marc von Wyl, Hisham Mohamed, Eric Bruno, and Stephane Marchand-Maillet. A parallel cross-modal search engine over large-scale multimedia collections with interactive relevance feedback. In *Proceedings of the 1st ACM International Conference on Multimedia Retrieval*, pages 73:1–73:2.

[10] H. V. Jagadish, Alberto O. Mendelzon, and Tova Milo. Similarity-based queries. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 36–45, New York, NY, USA, 1995. ACM.

[11] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[12] Marius Muja and David G. Lowe. Fast matching of binary features. In *Computer and Robot Vision (CRV)*, pages 404–410, 2012.

[13] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[14] K. Figueroa, G. Navarro, and E. Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.

[15] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

[16] Marco Patella and Paolo Ciaccia. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms*, 7(1):36 – 48, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP) 1st International Workshop on Similarity Search and Applications (SISAP).

[17] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[18] H. Samet. *Foundations of multidimensional and metric data structures*. The Morgan Kaufmann series in computer graphics and geometric modeling. Elsevier/Morgan Kaufmann, 2006.

[19] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[20] A. Henrich, H. W. Six, and P. Widmayer. The lsd tree: Spatial access to multidimensional and non-point objects. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[21] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 4:21–36, 1998.

[22] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[23] Alexander Hinneburg, Charu C. Aggarwal, and Daniel A. Keim. What is the nearest neighbor in high dimensional spaces? In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 506–515, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[24] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[25] R. Bellman and R.E. Bellman. *Adaptive Control Processes: A Guided Tour*. 'Rand Corporation. Research studies. Princeton University Press, 1961.

[26] Tibshirani R. Hastie, T. and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, Berlin, 2nd edition, 2009.

[27] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *In Int. Conf. on Database Theory*, pages 217–235, 1999.

[28] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[29] O. Egecioglu, H. Ferhatosmanoglu, and U. Ogras. Dimensionality reduction and similarity computation by inner-product approximations. *Knowledge and Data Engineering, IEEE Transactions on*, 16(6):714–726, 2004.

[30] Ümit Y. Ogras and Hakan Ferhatosmanoglu. Dimensionality reduction using magnitude and shape approximations. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 99–107, New York, NY, USA, 2003. ACM.

[31] Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[32] S. Santini and R. Jain. Beyond query by example. In *Multimedia Signal Processing, 1998 IEEE Second Workshop on*, pages 3–8, Dec 1998.

[33] Hisham Mohamed, Marc von Wyl, Eric Bruno, and Stphane Marchand-Maillet. Learning-based interactive retrieval in large-scale multimedia collections. In Marcin Detyniecki, Ana Garca-Serrano, Andreas Nrnberger, and Sebastian Stober, editors, *Adaptive Multimedia Retrieval. Large-Scale Multimedia Retrieval and Evaluation*, volume 7836 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2013.

[34] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[35] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: Self-tuning indexes for similarity search. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 651–660, New York, NY, USA, 2005. ACM.

[36] Christos Faloutsos and King-Ip Lin. Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. *SIGMOD Rec.*, 24(2):163–174, May 1995.

[37] Jason T. L. Wang, Xiong Wang, Dennis Shasha, and Kaizhong Zhang. Metricmap: An embedding technique for processing distance-based queries in metric spaces. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 35:2005.

[38] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 503–511, 2001.

[39] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[40] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 614–623, New York, NY, USA, 1998. ACM.

[41] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[42] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[43] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, April 1973.

[44] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools Appl.*, 14(2):113–135, June 2001.

[45] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[46] Jeffrey K. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175 – 179, 1991.

[47] Hartmut Noltemeier, Knut Verbarg, and Christian Zirkelbach. Monotonous bisector* trees - a tool for efficient partitioning of complex scenes of geometric objects. In *Data Structures and Efficient Algorithms, Final Report on the DFG Special Joint Initiative*, pages 186–203, London, UK, UK, 1992. Springer-Verlag.

[48] Michal Batko, David Novak, Fabrizio Falchi, and Pavel Zezula. On scalability of the similarity search in the world of peers. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[49] Xuemin Shen, Heather Yu, John Buford, and Mursalin Akon. *Handbook of Peer-to-Peer Networking*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[50] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[51] David Novak and Michal Batko. Metric index: An efficient and scalable solution for similarity search. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, pages 65–73, Washington, DC, USA, 2009. IEEE Computer Society.

[52] Rudolf Bayer and Karl Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, March 1977.

[53] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[54] Enrique Vidal Ruiz. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145 – 157, 1986.

[55] L. Mico, J. Oncina, and E. Vidal. An algorithm for finding nearest neighbours in constant average time with a linear space complexity. In *Pattern Recognition, 1992. Vol.II. Conference B: Pattern Recognition Methodology and Systems, Proceedings., 11th IAPR International Conference on*, pages 557–560, Aug 1992.

[56] E.C. Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1647 –1658, sept. 2008.

[57] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 28–36, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[58] Persi Diaconis. *Group Representations in Probability and Statistics*. Institute of Mathematical Statistics, 1988. Volume 11 in the Lecture Note–Monograph Series.

[59] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 613–622, New York, NY, USA, 2001. ACM.

[60] DouglasE. Critchlow. *Metric Methods for Analyzing Partially Ranked Data*, volume 34 of *Lecture notes in statistics*. Springer New York, 1985.

[61] Giuseppe Amato and Pasquale Savino. Approximate similarity search in metric spaces using inverted files. In *Proceedings of the 3rd international conference on Scalable information systems*, InfoScale '08, pages 28:1–28:10, ICST, Brussels, Belgium, Belgium, 2008. ICST.

[62] Giuseppe Amato, Claudio Gennaro, and Pasquale Savino. Mi-file: using inverted files for scalable approximate similarity search. *Multimedia Tools and Applications*, 2012.

[63] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.

[64] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, December 1998.

[65] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, CA, 2. edition, 1999.

[66] Eric Sadit Téllez, Edgar Chávez, and Antonio Camarena-Ibarrola. A brief index for proximity searching. In *Proceedings of the 14th Iberoamerican Conference on Pattern Recognition: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, CIARP '09, pages 529–536, Berlin, Heidelberg, 2009. Springer-Verlag.

[67] R. W. Hamming. Error detecting and error correcting codes. *Bell System Tech. J.*, 29:147–160, 1950.

[68] Eric Sadit Tellez and Edgar Chavez. On locality sensitive hashing in metric spaces. In *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, pages 67–74, New York, NY, USA, 2010. ACM.

[69] Andrea Esuli. Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. *Proceedings of LSDSIR 2009*, i(July):1–48, 2009.

[70] Eric Sadit Tellez, Edgar Chávez, and Gonzalo Navarro. Succinct nearest neighbor search. *Inf. Syst.*, 38(7):1019–1030, 2013.

[71] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *PROCEEDINGS OF ALENEX07, ACM*. Press, 2007.

[72] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, June 2005.

[73] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, Jan 2011.

[74] David Novak and Pavel Zezula. M-chord: A scalable distributed similarity search structure. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[75] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[76] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley and Sons, New York, 1990.

[77] Benjamin Bustos, Oscar Pedreira, and Nieves Brisaboa. A dynamic pivot selection technique for similarity search. In *Proceedings of the First International Workshop on Similarity Search and Applications*, SISAP '08, pages 105–112, Washington, DC, USA, 2008. IEEE Computer Society.

[78] Luis G. Ares, Nieves R. Brisaboa, María F. Esteller, Óscar Pedreira, and Ángeles S. Places. Optimal pivots to minimize the index size for metric access methods. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, pages 74–80, Washington, DC, USA, 2009. IEEE Computer Society.

[79] Benjamin Bustos, Gonzalo Navarro, and Edgar Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[80] Sergey Brin. Near neighbor search in large metric spaces. In *In Proceedings of the 21th International Conference on Very Large Data Bases*, 1995.

[81] Sanjoy Dasgupta and Philip M. Long. Performance guarantees for hierarchical clustering. *J. Comput. Syst. Sci.*, 70(4):555–569, June 2005.

[82] Giuseppe Amato, Andrea Esuli, and Fabrizio Falchi. In Nieves R. Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *SISAP*, Lecture Notes in Computer Science, pages 91–102. Springer.

[83] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38(0):293 – 306, 1985.

[84] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[85] Roberto F. Santos Filho, Agma J. M. Traina, Caetano Traina Jr., and Christos Faloutsos. Similarity search without tears: The omni family of all-purpose access methods. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *ICDE*, pages 623–630. IEEE Computer Society, 2001.

[86] Liang Jin, Nick Koudas, and Chen Li. Nnh: Improving performance of nearest-neighbor searches using histograms. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Bhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004*, volume 2992 of *Lecture Notes in Computer Science*, pages 385–402. Springer Berlin Heidelberg, 2004.

[87] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, December 2003.

[88] Jules Vleugels and Remco Veltkamp. Efficient image retrieval through vantage objects. *Pattern Recognition*, 35:69–80, 1999.

[89] Liang Jin and Chen Li. Selectivity estimation for fuzzy string predicates in large data sets. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 397–408. VLDB Endowment, 2005.

[90] Andrea Esuli. Mipai: Using the pp-index to build an efficient and scalable similarity search system. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, pages 146–148, Washington, DC, USA, 2009. IEEE Computer Society.

[91] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, 2009.

[92] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627v2, 2009.

[93] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–, Washington, DC, USA, 1999. IEEE Computer Society.

[94] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[95] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.*, 38:293–306, 1985.

[96] Roberto Grossi, Jeffrey, and Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *in Proceedings of the 32nd Annual ACM Symposium on the Theory of Computing*, pages 397–406, 2000.

[97] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[98] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Softw., Pract. Exper.*, 37(3):309–329, 2007.

[99] H. Mohamed and M. Abouelhoda. Parallel suffix sorting based on bucket pointer refinement. In *Biomedical Engineering Conference (CIBEC), 2010 5th Cairo International*, pages 98 –102, dec. 2010.

[100] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, San Diego, CA, USA, 1997.

[101] Giuseppe Amato, Fausto Rabitti, Pasquale Savino, and Pavel Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2):192–227, April 2003.

[102] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[103] David B. Skillicorn. A taxonomy for computer architectures. *Computer*, 21(11):46–57, November 1988.

[104] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966.

[105] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[106] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: General purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[107] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[108] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[109] The MPI Forum. Mpi: A message passing interface, 1993.

[110] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[111] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[112] Richard McCreadie, Craig Macdonald, and Iadh Ounis. Mapreduce indexing strategies: Studying scalability and efciency. *Information Processing and Management*, 2011.

[113] Dan Gillick, Arlo Faria, and John Denero. Mapreduce: Distributed computing for machine learning, 2006.

[114] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, first edition edition, june 2009.

[115] R. Rajasekaran and John Reif. *Handbook of Parallel Computing: Models, Algorithms and Applications*. CRC Press, 2007.

[116] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. Towards efficient mapreduce using mpi. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *PVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pages 240–249. Springer, 2009.

[117] Steven J. Plimpton and Karen D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[118] Xiaoyi Lu, Bing Wang, Li Zha, and Zhiwei Xu. Can mpi benefit hadoop and mapreduce applications? In *40th International Conference on Parallel Processing Workshops (ICPPW), 2011*, pages 371–379.

[119] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[120] Project gutenberg. http://www.gutenberg.org/.

[121] Surendra Byna. Improving the performance of mpi derived datatypes by optimizing memory-access cost. In *In Proceedings of the IEEE International Conference on Cluster Computing*, pages 412–419, 2003.

[122] Robert Ross and Neill Miller. Implementing fast and reusable datatype processing. In *In EuroPVM/MPI*, pages 404–413. Springer Verlag, 2003.

[123] JesperLarsson Trff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: efficient handling of mpi derived datatypes. In Jack Dongarra, Emilio Luque, and Toms Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116. Springer Berlin Heidelberg, 1999.

[124] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. *CoRR*, abs/1102.3828, 2011.

[125] MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.

[126] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. *SIGMOD Rec.*, 25(2):103–114, June 1996.

[127] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Comput.*, 21(8):1313–1325, August 1995.

[128] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In MartinGilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 674–679. Springer Berlin Heidelberg, 2009.