DISTRIBUTED INDEX FOR MATCHING MULTIMEDIA OBJECTS

by

Ahmed Abdelsadek

B.Sc., Cairo University, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

APPROVAL

Name:	Ahmed Abdelsadek
Degree:	Master of Science
Title of Thesis:	Distributed Index for Matching Multimedia Objects

Examining Committee: Dr. William N. Sumner
Chair

Dr. Mohamed Hefeeda,
Senior Supervisor
Professor, Computing Science
Simon Fraser University

Dr. Greg Mori,
Supervisor
Associate Professor, Computing Science
Simon Fraser University

Dr. Joseph G. Peters,
Internal Examiner
Professor, Computing Science
Simon Fraser University

Date Approved: October 27th, 2014

Abstract

This thesis presents the design and evaluation of DIMO, a distributed system for matching multimedia objects. DIMO provides multimedia applications with the function of finding the nearest neighbors on large-scale datasets. It also allows multimedia applications to define application-specific functions to further process the computed nearest neighbors. DIMO presents novel methods for partitioning, searching, and storing high-dimensional datasets on distributed infrastructures that support the MapReduce programming model. We implemented DIMO and extensively evaluated it on Amazon clusters with up to 128 machines. We experimented with large datasets of sizes up to 160 million data points extracted from images. Our results show that DIMO produces high precision when compared against the ground-truth nearest neighbors and it can elastically utilize varying amounts of computing resources. Additionally, DIMO outperforms the closest system in the literature by a large margin (up to 20%) in terms of the achieved average precision, and requires less storage.

Acknowledgments

It is my pleasure to express my profound gratitude to my supervisor, Dr. Mohamed Hefeeda, for his invaluable guidance, incessant encouragement, and persistent support throughout the course of this research. Without his critical reviews and intellectual inputs, completion of this thesis would not have been possible for me.

I would like to express my gratitude to Dr. Greg Mori, and Dr. Joseph Peters, for being on my committee and reviewing this thesis. I also would like to thank Dr. William Sumner for taking the time to chair my thesis defense.

I am grateful to all the members at the Network Systems Lab for providing me a stimulating and fun environment. I would like to thank all my wonderful friends who comforted me during the difficult times, and offered me great support and help.

Finally, and most importantly, I would like to offer my endless gratitude to my family for their ceaseless love and support

Contents

List of Figures

Preface

All the work in this thesis has been conducted under the supervision of Dr. Mohamed Hefeeda. Parts of the materials of Chapters 2-4 have been used in the following publication and have been presented in the corresponding conference:

Ahmed Abdelsadek, and Mohamed Hefeeda. DIMO: distributed index for matching multimedia objects using MapReduce. In Proceedings of the 5th ACM Multimedia Systems Conference, pp. 115-126. ACM, 2014.

Chapter 1

Introduction

In this chapter, we first describe the problem of multimedia object matching and its main applications. Then, we summarize the contributions of this thesis and describe how the thesis is organized.

1.1 Overview

The rapidly increasing volume of digital content generated daily over the Internet poses many research challenges for efficiently storing, processing, and searching such sheer volume. In this thesis, we address one of these challenges: matching of multimedia objects, which is the problem of finding similar objects to a given query multimedia object. We address this problem for large-scale multimedia datasets.

Multimedia object matching involves a crucial step of similarity search. Similarity search is an important problem with numerous real-life applications, such as video copy detection [38], personalization and collaborative filtering [20], near-duplicate detection [33], visual recognition [74], music similarity search [48], text categorization [10], and data mining [16]. In such applications, objects are represented by one or more feature vectors (high-dimensional points), and the similarity search is conducted via a variant of *nearest neighbors* search algorithm, where a method for finding the nearest neighbors of a given data point is applied. After obtaining the nearest neighbors for a given point or group of points, further processing steps may need to be applied to refine the obtained object matching results. These steps depend on the specific application that uses the nearest neighbors results.

As an example, for a video copy detection application [38], the first step is to search the reference videos for frames similar to each individual frame from a suspected query video, which is a nearest neighbors search problem. Then, to decide whether the suspected video is a copy of one of the reference videos, the temporal aspects of the found frames are checked to see if they are in the same temporal order as the query video, which is an object matching problem.

Many algorithms have been presented in the literature for solving the nearest neighbors problem [13, 30, 72, 18, 15]. However, most of them are not time efficient for data with high dimensions. It is shown in [14, 34] that due to the *curse of dimensionality*, there is no known algorithm for exact nearest neighbors to be faster than a linear scan of the data when the dimensionality exceeds 10–15 dimensions. Therefore, researchers consider *approximate nearest neighbors* by trading the accuracy with the running time. This is done through allowing an acceptable degradation of the accuracy of the results compared to the exact ones. The goal of approximate nearest neighbors search is to reduce search time by possibly introducing an error in the result. This error is acceptable because in most of the applications mentioned above, the approximate results are almost as good as the exact ones.

Several approximate nearest neighbors search methods exist in the literature [36, 29, 49, 6, 58, 50]. However, most of them are memory-based and used in centralized settings, where all the dataset points can fit in one machine's memory. With the huge amount of digital content created and made online everyday, this assumption no longer holds, as the processing, storage, and memory needed for searching and retrieving from such volumes of data go beyond the capacity of a single machine. Thus, distributed algorithms and systems are needed to handle such massive volumes of data. Beside having more processing and storage capacities, distributed systems can tolerate failures, and can scale up/down to handle different sizes of datasets.

The most common way to solve the large-scale nearest neighbors search problem is to construct a distributed index. The index allows for quickly finding the data points closest to a given query point by processing the search query in parallel over a set of machines. In addition to the usual performance aspects of a centralized index to be time and space efficient, the distributed index also needs to be bandwidth efficient as network communication is one of the critical bottlenecks that may slow down the whole system. Several distributed approaches have been proposed in the literature, such as [32, 64, 2, 73, 46, 9, 65, 51]. However, most of these approaches either do not support high (100s) dimensions [73, 46], are customized for a specific application [2, 65], or do not consider reliability and fault-tolerance [32, 65, 51].

We propose a distributed index for matching multimedia objects, which we call DIMO (Distributed Index for Matching Objects). Examples of multimedia objects are: images, 2D videos, 3D videos, and audio files. Multimedia objects are typically characterized by many features and each feature is represented by a high-dimensional vector. For example, a single image can be characterized by hundreds of SIFT [45] descriptors, each has 128 dimensions. A video object will even have more features. This nature of feature-rich high-dimensional multimedia data adds more processing and storage requirements to the problem of matching objects.

DIMO provides the function of finding K-nearest neighbors (KNN). It also supports application-specific matching functions to be applied on the computed nearest neighbors in a distributed and transparent manner. In contrast to previous works, the proposed DIMO system offers the following desirable properties:

- **General.** DIMO can be used for different multimedia object matching applications. It produces approximate K-nearest neighbors, where the accuracy of the neighbors can be traded off with the computation and storage requirements. DIMO can support high-dimensional multimedia data. In our experiments, we use data points with 128 dimensions, extracted from images.

- **Scalable.** DIMO is designed for large-scale datasets. In some of our experiments, we use more than 160 million data points.

- **Elastic.** DIMO can automatically use varying number of computing machines. We show in our experiments that DIMO can scale almost linearly with the available number of computing machines.

- **Dynamic.** Data points can be added and removed from the system in dynamic manner, without the need to re-build the whole system.

We have implemented DIMO and extensively evaluated it on Amazon clusters [5] with varying number of computing machines ranging from 8 to 128. We have experimented with large datasets of sizes up to 160 million data points extracted from publically-available image datasets [24, 4], where each point is a 128-dimensions SIFT descriptor [45]. Our experimental results show that DIMO: (i) results in high recall when compared against the ground-truth nearest neighbors, (ii) can elastically utilize varying amounts of computing resources, (iii) does not impose high network overheads, (iv) does not require large main memory even for processing large datasets, and (v) balances the load across the used computing machines. Furthermore, DIMO can run on tens to thousands of machines, because it has no hot-spot central machine or single point of failure.

1.2 Problem Statement and Thesis Contributions

In this thesis, we investigate the problem of matching multimedia objects, which is the problem of finding similar objects to a given query multimedia object. Matching multimedia objects involves an important step of similarity search, where objects are represented by one or more feature vectors (high-dimensional points), and the similarity search is conducted via a nearest neighbors search algorithm. Nearest neighbors search is the problem of finding the closest points to a given query point from a set of reference points. Driven by the large scale and high dimensionality of the data nowadays, centralized and exact solutions are no longer efficient nor scalable enough to solve it. Distributed and approximate solutions are more practical for such scale and dimensionality of data. We focus on solving the problem of large-scale multimedia object matching, which includes designing of efficient methods for solving the distributed approximate nearest neighbors problem. We focus on solving the approximate nearest neighbors problem for data points in the Euclidean space. We address this problem for large-scale multimedia datasets that can not fit in one machine.

The contributions of this thesis can be summarized as follows [1]:

- We present DIMO: a novel distributed scalable system for matching multimedia objects over large-scale datasets. DIMO is suitable for various content-based multimedia information retrieval applications, because it has a flexible design that can support different components for partitioning the data and matching the objects. DIMO indexes large volumes of high-dimensional data points extracted from multimedia objects. It compares the indexed reference points to the ones extracted from the query objects for finding, sorting and reporting the matched multimedia objects.

- We present adapted designs and implementations of the state-of-the-art space-partitioning techniques: (i) K-dimensional forest (KD forest), and (ii) multi-probe locality-sensitive hashing (MP-LSH). These space-partitioning techniques are used in DIMO.

- We implement DIMO and extensively evaluate it on clusters of different numbers of machines and on large-scale publically-available image datasets. Our results show that DIMO yields high accuracy, scales almost linearly with data size and number of machines, and efficiently utilizes the computing resources.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents a background on the object matching problem and the general methods to solve it. It also surveys the related works in the literature. Chapter 3 presents the details of the proposed DIMO system, and Chapter 4 presents our evaluation setup and results. Chapter 5 concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, we start by providing the background needed to understand the concepts we discuss later in the thesis. We discuss the closest works in the literature and mention the applications for our problem and how it is used in other systems.

2.1 Nearest Neighbors Search

As we mentioned in Chapter 1, the general approach in most similarity search applications is to extract features that represent the original data objects, each feature is represented by a multi-dimensional point. A similarity query is defined as finding the most similar data objects in the dataset to a given query object. For example, in image databases, a possible similarity query is to find the most similar images to a given query image. The images are represented by one or more d-dimensional point and the similarity between the images is defined by a distance function (e.g., Euclidean distance) between the corresponding features. The k-nearest neighbor (K-NN) problem is defined as finding the k most similar data points to a given query point. A closely related type of query is the r-range query, where all data points that are within distance r from the query point q are retrieved.

2.1.1 Formal definition

Given a set of reference points R in a metric space $\mathbf{M}$, and a set of query points $Q \in \mathbf{M}$. For each query point $q \in Q$ we define the following types of retrieving mechanisms:

- K-NN: find the closest K points in R to q.
- r-NN: find all points in R within distance r from q.

In most of the cases, $\mathbf{M}$ is taken to be a d-dimensional Euclidean space and the distance between the points is measured by Euclidean distance or any Minkowski distance.

Metric Space

A set of objects is called a metric space if a notion of distance (called a *metric*) $d(.,.)$ is defined between any two objects in the set and satisfies the following four conditions:

- Non-negativity: $\forall\, q, r \in \mathbb{U} : \quad d(q,r) \geq 0$
- Symmetry: $\forall\, q, r \in \mathbb{U} : \quad d(q,r) = d(r,q)$
- Identity: $d(q,r) = 0 \;\Rightarrow\; q = r$
- Triangle inequality: $\forall\, s, q, r \in \mathbb{U} : \quad d(q,r) \leq d(q,s) + d(s,r)$

Euclidean space

Euclidean d-dimensional space is the space of all d-tuples of real numbers, $(X_1, X_2, ..., X_d)$, and it is a special case of a metric space. It is commonly denoted $\Re^d$. The Euclidean distance between two points r and q is the length of the line segment connecting them. If $r = (r_1, r_2, ..., r_d)$ and $q = (q_1, q_2, ..., q_d)$ are two points in Euclidean d-dimensional space, then the distance from r to q, or from q to r is given by:

$$d(r,q) = d(q,r) = \sqrt{(q_1 - r_1)^2 + (q_2 - r_2)^2 + \cdots + (q_d - r_d)^2} = \sqrt{\sum_{i=1}^{d} (q_i - r_i)^2}. \quad (2.1)$$

2.1.2 Curse of Dimensionality and Approximate Nearest Neighbors Search

The curse of dimensionality is the set of phenomena that occur while dealing with data points in high-dimensional spaces that do not occur in low-dimensional spaces. The problem is when the dimensionality increases, the volume of the space increases exponentially fast, and the space of the data points becomes very sparse. This sparseness causes the classical techniques to fail to group similar points, as most of them relies on detecting the areas in the space where points can form groups with small distances between them. This effect complicates the nearest neighbors search in high-dimensional spaces, as in the most distance functions that depends on all the dimensions, it is hard to quickly reject candidate points by considering the difference in one or few dimensions as a lower bound for the distance function [17].

This leads to the fact that using the classical structures is not efficient when the feature space is a high-dimensional space [34]. In such cases, the most efficient way to solve exact nearest neighbors is to sequentially scan the entire dataset, comparing each reference point against the query point. Obviously, such solution is not applicable for large-scale datasets.

In order to accelerate the search, several works introduced the approximate nearest neighbors search algorithms, where the problem is addressed by trading the accuracy with the running time. This is by allowing an acceptable degradation of the accuracy of the results compared to the exact ones. The goal of approximate similarity search is to reduce search times for similarity queries by possibly introducing an error in the result through reducing the cost of similarity queries by relaxing the correctness constraint, i.e., the K points in the approximate result might not be the closest to the query point.

2.1.3 Standard techniques for finding nearest neighbors

Techniques for solving the nearest neighbors problem can be divided into two main categories: (i) hierarchical space division and (ii) space mapping. In the first category, algorithms use a general strategy of hierarchically dividing the search space into regions of close-by points, and represent them in a tree structure. Then, branch and bound methods are used to search and manipulate these tree structures. These techniques can be used either in an Euclidean space, e.g., R tree [30] and KD tree [13], or in a general metric space, e.g., VP tree [72] and M tree [18]. These techniques are mainly designed to return exact nearest neighbors, and are suitable only for low-dimensional data.

As we mention in Section 2.1.2, due to the curse of dimensionality, for high-dimensional data, exact neighbors are costly to find because most nearest neighbors search methods will do no better than linear scan of the whole dataset [14, 34]. Thus, approximate nearest neighbors search methods have been proposed in the literature. These methods constitute the second category, space mapping, for solving the nearest neighbors problem. These methods first modify, or map, the search space into smaller or lower-dimensions one, then solve the problem on the new (approximate) space, where the search is simpler. Examples of such techniques are those use approximate filters to reduce the amount of scanned points in the dataset such as vector approximation (VA-Files) [71]; ones that are based on dimensionality reduction using projections [49, 23]; or those based on locality-sensitive hashing (LSH) [36, 29, 6].

2.2 K-Dimensional Trees and Forests

KD tree (short for k-dimensional tree) [27] is a space-partitioning tree [54] for arranging k-dimensional points in a k-dimensional Euclidean space. KD trees are a generalization of binary search trees where every node represents is a $k$dimensional point. It is a binary tree that hierarchically divides the points space on each tree level over one of the dimensions' axis, where every non-leaf node can be considered as separating hyperplane that divides the space into two regions. Points to the left of this hyperplane represent the left sub-tree of that node and points to the right of the hyperplane represent the right sub-tree. At each node, the hyperplane direction is perpendicular to one of the dimension's axis that corresponds to its tree level. KD tree was firstly introduced in [13] and since that time various implementations with several improvements have been presented in the research community. KD tree represents the hierarchical space division category and arguably on top of it.

The major drawback of the traditional KD tree for solving the NNS is that it suffers curse of dimensionality when the points dimension exceed 10-15 dimensions. Several enhancement are introduced to the KD tree to make it suitable for the high-dimensional data at the price of getting approximate neighbors not the exact ones [7, 12, 8, 58]. Some works address the long running time by putting limits on the search execution. In [12], Beis and Lowe put a *time bound* on the execution, that is stopping the search after certain number of backtracking steps in the tree search, and return the best answers so far. In [8], Arya et al. put an *error bound* on the search, that is stopping the search if the error is within $1 + \epsilon$ of the true neighbors. Both techniques however still consider the full dimensions of the data points. In [58], Silpa-Anan and Hartley use a subset of dimensions based on their variance. To mitigate the precision loss due to this selection. They build multiple instances of such a tree, each with different subset of the original dimensions. The authors show the effect of independence between the trees, that is if the KD trees are built with different subsets of the dimensions, then the space-partitioning is going to be different for each tree, and the order of search node and search results on these KD-trees is also going be different, and the final results is going to be the union of the results of all the KD trees. This leads to the idea of using multiple KD trees (or a KD forest) to enhance the approximation results. However this solution may introduce extra memory or storage used for storing multiple trees instead of one.

In [4], Aly et al. did a benchmark of several NNS techniques including the KD forest and his results shows the superiority of the KD forest algorithm in compared to other techniques. Based on their results, the authors argue that the KD forest is state-of-the-art for nearest neighbors search.

2.3 Locality-sensitive hashing

When dimensions of data points gets even higher ($\gg 1000s$), the previous techniques sometimes fail to achieve good performance. Authors of [36, 29] tried to find solutions to the nearest neighbors search problem that are independent of the data dimensionality. They introduced the locality-sensitive hashing (LSH) as a possible cure for that problem. Locality-sensitive hashing is a method of performing probabilistic hashing of high-dimensional data. The basic idea is to hash the input points so that similar points are mapped to same buckets with high probability, with the number of buckets being much smaller than the universe of possible input points. Some authors argue that it is the state-of-the-art in NNS when the dimensionality is very high [47, 65]. Locality-sensitive hashing represents the space mapping category and arguably on top of it.

The general idea of LSH relies on the existence of locality-sensitive hash functions. Let *H* be a family of hash functions mapping $\Re^d$ to some universe *U*. For any two points *p* and *q*, consider a process in which we choose a hash function *h* from *H* uniformly at random, and analyze the probability that $h(p) = h(q)$. The family H is called locality-sensitive if it satisfies the following conditions:

A family H is called (R, cR, P_1, P_2)-sensitive if for any two points $p, q \in \Re^d$.

- $if \parallel p$ - $q \parallel \leq R$ then $Pr_H[h(q) = h(p)] \geq P_1$
- $if \parallel p$ - $q \parallel \geq cR$ then $Pr_H[h(q) = h(p)] \leq P_2$

In order for a locality-sensitive hash family to be useful, it has to satisfy $P_1 \gg P_2$.

Typically we can not use *H* as is, since the difference between the two probabilities P_1 and P_2 for a single hash function could be quite small. Thus, an amplification process is needed in order to achieve bigger difference between probabilities of collision P_1 and P_2. We describe this step as follows: Given a family *H* of hash functions with parameters (R, cR, P_1, P_2), the gap between the high probability P_1 and low probability P_2 is amplified by concatenating several hash functions. In particular, for parameters *M* and *L*, we choose *L* functions $g_j(q) = (h_{1,j}(q), \ldots, h_{M,j}(q))$, where $h_{i,j}(1 \leq i \leq M, 1 \leq j \leq L)$ are chosen independently and uniformly at random from *H*. These are the actual functions to be uses for hashing the data points. The data structure is constructed by placing each point *p* from the input set into a bucket $g_j(p)$, for $j = 1, \ldots, L$. To process a query *q*, the buckets $g_1(q), \ldots, g_L(q)$ are scanned for computing their distances to the query point, and report any point that is a valid answer to the query [6].

2.4 MapReduce

Driven by growing demand for efficient processing and searching large-scale data, where the amount of data can not fit in one machine's memory, the distributed similarity searching systems have received considerable attention. To support similarity and nearest neighbors queries in the large-scale datasets of high-dimensional and unstructured data points, an efficient distributed index structure is presented, where building and searching is executed across multiple machines in computing cluster. For such systems, at the level of network and machines topology transparency, most of the works are done using two kinds of distributed structures: either considering the network topology, which mostly implemented using either socket programming or massage passing interface (MPI), or using fully transparent approach from the network topology, which are mainly implemented using frameworks like MapReduce [22].

The MapReduce framework provides an infrastructure that runs on a cluster of machines, which automatically manages the execution of multiple computations in parallel as well as the communications among these computations. It also provides transparent redundancy and fault tolerance to computations. In its simplest form, a computation in MapReduce (called MapReduce job) is composed of two functions: mapper and reducer. The inputs and outputs of both functions are in the form of key-value pairs, where the key and value can be complex objects. The programer specifies the computations that should be performed in the mapper and reducer functions as well as the format of the input and output pairs. The MapReduce infrastructure creates multiple mapper instances, divides the data among them, runs them on the available machines, aggregates their outputs and passes them to reducers, and finally produces the outputs from the reducers. The MapReduce infrastructure also monitors the execution of mappers/reducers on all machines and it can handle machine failures/slowness by restarting failed/delayed mappers/reducers on different machines. Although the MapReduce infrastructure provides quite useful services, MapReduce programs need to be carefully designed to achieve good performance as there are several important issues that should be considered, such as the volume of data exchanged among mappers and reducers and the number of I/O operations that need to be performed. In addition, MapReduce programs may have no reducers (only mappers). Other functions, e.g. combiners, are also possible in MapReduce programs.

2.5 Applications for Nearest Neighbors Search

Nearest neighbors search (NNS) is an old and quite established optimization problem, with hundreds of applications in many fields in computer science. For example, image search [2], and text categorization [10]. In some works, authors introduce their works as general purpose nearest neighbors search algorithms. In such a case, they do not report results on the application-level. In some other works, authors tend to tie their works to certain applications, which gives a bigger window for more approximation accuracy through ad-hoc cues suitable only to their specific applications. For example, in [2], Aly et at. developed an algorithm for NNS where they tied their application to the problem of finding similar images. They report the accuracy of the whole image search system, however, they did not report the accuracy of the NNS component on its own.

2.5.1 General Purpose Nearest Neighbors Search

In a standalone nearest neighbors search algorithm, the input is a set of data points and the output is the set of the closest points to a given point followed by distance (e.g., similarity score). Many works are done and presented as a standalone NNS algorithm [36] [6] [50] [66] [64] [8], where the authors did not tie the usage of their algorithms to a specific application, so it can be used with any arbitrary application as long as it can fit in the problem description mentioned earlier.

In the general NNS, testing and evaluation have many approaches. In some works, the authors go only for theoretical analysis and formalization of the bounds on the algorithm accuracy error [36] [6], where they did not do actual implementations nor try to evaluate the practical performance of their algorithms. In some other works, the authors use randomly generated data synthetically from some known distributions [66] [50] [64] [8]. In [50], the authors described the randomly generated datasets as one of the most challenging ones for the high-dimensions NNS. That is because in randomly generated data, there is no correlation between the dimensions of the points and no value can give any prediction about the others. In [8] the authors experimented and reported results from many distributions like Uniform, Gaussian, and Laplacian.

Although in general NNS the authors do not consider a specific application, some use real data (i.e. from images or videos) for their evaluations [50] [66] [64]. Using real data can give better approximation results, because of the correlation and predictability between the dimensions of a point, and between the points themselves. In [50], the authors show significant accuracy increase when using real data rather than random data on the same algorithm. Moreover, this makes their approximation results are more realistic and practical. However, in such works where the authors used real data sets from images [50] [66] [64], they still did not report the whole accuracy of detecting similar images. They only report the NNS accuracy, and they did not provide a complete scheme nor results for an image search application, which makes their work fall under the general NNS category.

2.5.2 Application-specific Nearest Neighbors Search

With the big number of applications based on nearest neighbors search, they can generally be divided into two main categories: searching and classification. In this section we will give some examples and applications for both types.

Searching and Retrieval

For searching and retrieval of complex objects using NNS, the general technique is: first, we extract many features from the data objects each is represented by a high-dimensional point. Then, we store the points in an index structure capable of performing NNS. At query time, for each query object, we extract the features from the object and for each feature we search the index for similar features, which the nearest neighboring points in the feature space. Finally, a consistency check may be executed to see whether the matched individual points are consistent with respect to the query object.

In [2], Aly et al. developed a system for similar images retrieval based on NNS algorithm in their index. They index large sets of images, the datasets they used are Caltech Game Covers Dataset and Caltech Buildings Dataset [4] among millions of other images downloaded from the internet. They use the scale-invariant feature transform (SIFT) [45] to extract visual features from the images. For each query image, they search the index for similar features and report the nearest among them. They use RANSAC algorithm for further refining the retrieved features and removing the outlier points. They report the accuracy of their image retrieval system as one component. In [38], Khodabakhshi and Hefeeda developed a system for finding copies of 3D videos, where they depend on NNS algorithm in the underlying index. They split the videos into frames, and for each frame they extract the speeded-Up Robust Features (SURF) [11]. For each query suspected video, they split the video to frames and extract the SURF features from the frames. They search the index for similar features and report the nearest among them. They use matrix alignment method to check if the retrieved reference video frames are in a temporal order consistent with query video frames. They report the accuracy of the whole copy detection system.

Classification and Regression

Other main usage for nearest neighbors search is classification task for supervised machine learning. In such usage, each object is giving a label or a class. The classification task is, when a new object is found, we need to predict its label or class. This task is solved by an NNS algorithm by searching for the closest objects to the new object, then the label for the new object is assigned to the label of the majority among the retrieved ones. There are many parameters affect the quality of the nearest neighbors classifier like the number of retrieved neighbors or the searching radius. The performance of the nearest neighbors classifier is also affect by the classification task nature and the objects' features representation.

One of the classical machine learning tasks is handwritten character recognition, the authors [62] used nearest neighbor algorithm in character classification. They did their experiments on large databases with tens of thousands of training examples. They report accuracy for the nearest neighbors classifier reached 99% when using large enough training examples. In [74], Zhang et al. combined the NNS with support vector machine (SVM) algorithm for different tasks of visual category recognition. They experimented with tasks like handwritten character recognition, texture analysis, and object categorization. They used four different dataset which are, MNIST [42] and USPS [41] for handwritten digits, CUReT [19] for textures, and finally Caltech-101 [43] for object classes. In [40] Kulis and Grauman integrate locality-sensitive hashing (LSH) with a kernel machine algorithm for scalable image search and object recognition tasks. They used correspondence-based local feature kernel (CORR) [74] for recognition. Their dataset was Caltech-101 [43] for object classes. They report the precision and recall of their object recognition task.

2.6 Related Work

Most of the nearest neighbors search techniques were mainly designed to run on a single machine. Several works have attempted to solve the nearest neighbors problem in a distributed manner in order to support the rapidly-increasing volumes of data being created nowadays. For example, some works exploit peer-to-peer (P2P) networks for distributed similarity search [26, 32, 69]. In [26], Flachi et al. introduce M-CAN, which is based on the Content-addressable Network (CAN) P2P architecture. M-CAN uses a pivot-based technique to project objects from the metric space to an N-dimensional vector space, and it then maps them to peers. Haghani et al. [32] use LSH on top of the Chord P2P architecture. In [69], Wang et al. propose RT-CAN, a distributed similarity index, which implements a variation of the R-tree on top of CAN. For massive datasets, the approaches that use structured P2P networks could suffer from multiple practical issues. First the mismatch between the overlay and physical networks, i.e., neighboring nodes in the overlay can be far away in the physical network, can increase the communication delay and overhead between distributed machines. Second, node failures cause the employed P2P networks to invoke failure recovery schemes, which impose communication and computation overheads. Unlike our proposed system, the works in [26, 32, 69] did not focus on large-scale multimedia datasets that are characterized by high-dimensional points. These works either used low dimensional data, e.g., five dimensions in [69], or simulation on a single machine in [32].

In some recent work [51], Muja and Lowe built a distributed system for solving the high-dimensional nearest neighbors search on multiple machines. They used both hierarchical K-means [52] and randomized KD tree [58] algorithms for space-partitioning. Their distributed implementation based on client/server model and using message passing interface (MPI), however they lack a clear routine of handling machine failures specially the master (server) machine. Also they broadcast all the query points to all worker machines which causes a server bottleneck and wastes much of the bandwidth. They did not perform large-scale experiments, they only used 8 machines with all data points stored in memory. In a similar work [65], Sundaram et al. proposed a system for streaming similarity, where the query points arrive in a stream fashion, unlike batches in our scenario. They implement what they call parallel locality-sensitive hashing (PLSH), over multiple machines and multiple cores. They use a dataset of 1 billion tweets and use TF-IDF as their features. They report efficient performance and small latency for queries, however many of their optimization techniques are designed towards the specific data they used. They use a variant of LSH, in their implementation they use up to 780 hash tables. To achieve low latency they store all the hash tables in the main memory of the computing machines, which costs more machines to store the data in memory rather than disks. They also use a client/server model with central node as a server, and they also do not provide a clear scheme for failures recovery.

Another class of works for solving the nearest neighbors problem in a distributed manner relies on distributed processing frameworks such as MapReduce. In [44], Liao et al. build a multi-dimensional index using R-tree on top of the Hadoop distributed file system (HDFS) [57]. Their index, however, can only handle low dimensional datasets–they performed their experiments with two dimensional data. In addition, their index is optimized for queries in a static environment. In contrast, our proposed system is dynamic and scalable, where data points can be added/removed without re-building the main data structures.

The authors of [46] and [73] solve the K nearest neighbors joins over large datasets using MapReduce [22]. Lu et al. [46] construct a Voronoi-like diagram, using some selected pivot objects. They then group the data points around the closest pivots and assign them to partitions, where searching can be done in parallel. Zhang et al. [73] split both query and reference datasets into a number of disjoint equal-sized subsets, where operations can be done in parallel. The systems in [46, 73] are also designed for low dimensional datasets; they did not consider data with more than 30 dimensions. In contrast, in our experiments we used image descriptors with 128 dimensions.

In [64], Stupar et al. present a method for implementing a distributed LSH index on a computing cluster. They maintain a number of hash tables over a set of machines, and they use MapReduce primitives for searching the tables for similar points. A major drawback of this approach is that it requires storing multiple replicas of the datasets in hash tables. This incurs significant storage cost and it increases the number of I/O operations. In contrast, our system stores the dataset only once and it produces higher precision in the returned neighbors.

Finally, Aly et al. [2] present two approaches to construct distributed KD trees on MapReduce for finding similar images. The first, called Independent KD-tree (IKdt), partitions the image dataset into equal-sized subsets. Each partition is assigned to a machine that builds an independent KD tree.

At query time, all machines search in parallel for the closest match. The second approach, called Distributed KD-tree (DKdt), builds a global KD tree across all machines. A single root machine stores the top of the tree, while multiple leaf machines store the rest of the tree. At query time, the root machine forwards data points to a subset of the leaf machines. In consequent work [3], The authors argue that KD tree has best performance among their tested algorithms. One of the drawbacks of this work is the single root machine that directs all query points, which makes it a single point of failure as well as a bottleneck that could slow down the whole system. Our system does not use a central node, and thus it is more robust and scalable. In addition, the work in [2] is tailored to image search, while our system is more general. For example, the authors of [2] report only the accuracy at the application level (image matching). Whereas we quantify the accuracy of the returned nearest neighbors (low level), which is general and important for various applications.

Chapter 3

Proposed Solution

In this chapter, we describe the details of the proposed DIMO system. We start by presenting an overview of the system and its main components in Section 3.1. We describe the methodology and algorithms used in DIMO in Sections 3.2, 3.3, and 3.4. We describe the distributed implementation and operations in Sections 3.5, 3.6 and, 3.7.

3.1 Overview and System Architecture

DIMO is a general system for similarity searching and matching of multimedia objects that are characterized by many features and each feature is represented by a high-dimensional point (i.e., feature vector or descriptor). To achieve this general matching task, DIMO first provides an efficient, distributed, implementation for computing K nearest neighbors for high-dimensional data points. Then, DIMO provides a generic interface for post-processing these neighbors based on the different needs of various applications. Thus, many applications can be built on top of DIMO, such as image search and video copy detection. DIMO is designed to handle large datasets with millions of points, and it can elastically utilize varying number of computing machines.

Basically, DIMO takes two sets: *reference* points R and *query* points Q. Each set contains d-dimensional data points. There are no constraints on the sizes of Q and R. However, R is assumed to change at a slower rate, by adding/removing objects to/from it. Whereas Q can change faster. For example, in an image search application, an archive of stored images would constitute R, while images submitted to the application for finding ones similar to them would make Q. DIMO builds

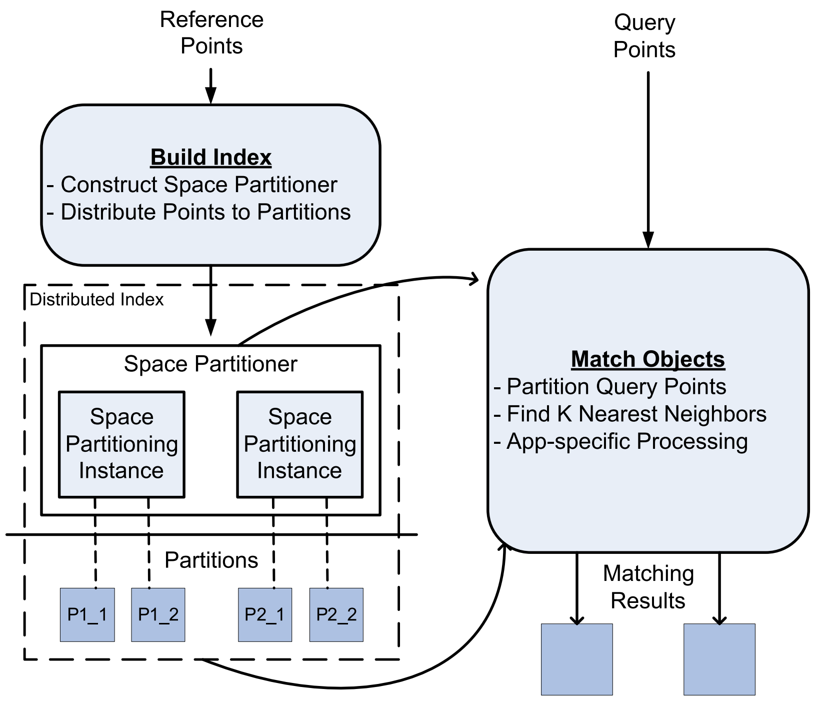

Reference Points
Query Points
Build Index
- Construct Space Partitioner
- Distribute Points to Partitions
Distributed Index
Space Partitioner
Space Partitioning Instance
Space Partitioning Instance
Partitions
P1_1
P1_2
P2_1
P2_2
Match Objects
- Partition Query Points
- Find K Nearest Neighbors
- App-specific Processing
Matching Results

Figure 3.1: High-level architecture of DIMO. Round boxes are MapReduce jobs. The top part of the distributed index, space partitioner, is serialized and used by multiple machines, while partitions at the lower part contain data points and are stored as files on the distributed file system.

a distributed index over R and it then uses it to search and match objects in Q. The cost of building the index is amortized over processing many queries. Note that data points in R and Q represent features of multimedia objects. Thus, multiple data points can belong to the same object. We assign the ID of an object to each data point generated from that object. This allows us to group the results of the K nearest neighbors phase based on object IDs to support various multimedia applications.

At a high level, DIMO splits the reference points R into partitions. Each partition is mapped to a file and the partitions are stored on a distributed file system. The partitions are searched in parallel against query points Q. The core component in our system that enables efficient partitioning of data points and searching for objects is the *space partitioner*, which we describe in details in Section 3.2.

Figure 3.1 shows the high-level architecture of the DIMO system. There are two main computational tasks: Build Index and Match Objects. Build Index first constructs the space partitioner by estimating its parameters from a sample of the reference points set. Then, it creates the *Distributed Index* of all the reference points set using the space partitioner it built earlier. This is done by assigning each point to its partition and then store each partition in a file on the distributed file system. Match Objects takes the query points set and computes the nearest neighbors for each query point. It also executes application-specific object matching functions on the found nearest neighbors. DIMO is implemented using the MapReduce distributed programming model [22]. The two computational tasks mentioned above are MapReduce jobs that run on multiple machines.

As shown in Figure 3.1, the distributed index is divided into two parts: (i) *Space Partitioner* and (ii) *Partitions*. Space partitioner is an abstract data structure that is used to group similar points in the same or close-by partitions. It is also used to forward query points to the partitions with potential matches. We build it to be a compact and an in-memory data structure that can be loaded and used by various computational tasks in the system. Partitions are the end regions of the space partitioner that contain the actual data points. Each partition is mapped to and stored as a single file on the distributed file system.

The design of our index has has two main features that make it simple to implement in a distributed manner, yet efficient and scalable. First, data points are stored only at the files associated with the partitions. The space partitioner itself does not store any data, it only stores meta data to guide the search. This significantly reduces the size of the space partitioner and makes it fit easily in the main memory of a single machine even for large datasets. This feature allows us to distribute copies of the space partitioner to multiple computing machines to process queries in parallel. Replicating the space partitioner on different machines not only facilitates parallel processing, but it also greatly improves the robustness and efficiency of the system. The robustness is improved because there is no single point of failure. The efficiency is improved because there is no central machine or set of machines that other machines contact during the computation. The second feature of our index design is the separation of the partitions and storing them as files on the distributed file system. This increases reliability as well as simplifies the implementation of the parallel computations in our system, because the concurrent access of data points is facilitated by the distributed file system. In addition, having data points only at files makes updating the index easier as explained in Section 3.7.

The proposed design of the DIMO system achieves the desired properties mentioned in Chapter 1. For example, it is general, because multiple applications can be supported by implementing the application-specific object matching functions in the Match Objects task, after the basic nearest neighbors search function is performed. The elasticity feature of DIMO, i.e., the ability to automatically use various number of machines, is achieved by the MapReduce framework. Finally, in Section 3.7, we discuss how DIMO can scale to very large datasets and how it can dynamically add/remove data points to the reference set.

The rest of this chapter is organized as follows: first we describe the space partitioner in general, then we describe our adapted designs of two concrete methods of space-partitioning, which are K-dimensional forest (KD forest) and multi-probe locality-sensitive hashing (MP-LSH). Finally, we describe the distributed implementation and operations of the DIMO system.

3.2 Space Partitioner

In our design, *space partitioning* is the abstract concept of dividing the search space into partitions where each partition groups similar points, such that the points in the same partition are closer to each other than the points in the other partitions.

Space partitioner consists of multiple space-partitioning instances, each instance represents a different partitioning of the dataset. Each space-partitioning instance is built independently with different parameters. Building multiple instances gives a room for more accurate results, as the final results is the union of each individual instance results. This increase in the accuracy comes at the cost of storing multiple copies of the dataset on the distributed file system. Nevertheless, space partitioner can consist of only one space-partitioning instance.

Space partitioner is used to group similar points in the same or close-by partitions. It is also used to forward query points to the partitions with the most potential matches. It is an in-memory data structure that can be loaded and used by various computational tasks in the system. Partitions are the end points of the space partitioner, and they are stored as files on the distributed file system.

Space partitioner can be implemented using different data structures (e.g., LSH [21], KD tree [8]). We design our space partitioners to be scalable by having a controllable number of partitions. This enables handling different dataset sizes without the need to rebuild the whole index. We also design it to support dynamic change of the datasets (i.e., adding/removing data points), and to maintain a good balance between the number of points in each partition.

As we discussed in Section 2.2 and Section 2.3, space partitioners split the data points to partitions based on one of two indexing schema: (i) using tree-based partitioning or (ii) using hash-based partitioning. We design and implement an example space partitioner from each indexing scheme.

For tree-based approach, we choose KD Forest [58] as the base of our space partitioner, as it is considered to be the state-of-the-art algorithm in its category. In our design of the space partitioner based on KD forest, each space-partitioning instance is a KD tree and each partition is a tree bin.

For hash-based approach, we choose multi-probe locality-sensitive hashing (MP-LSH) [47] as the base of our space partitioner, which is also considered to be the state-of-the-art algorithm in its category. In our design of the space partitioner based on MP-LSH, each space-partitioning instance is a hash table and each partition is a range of hash buckets.

3.3 KD Forest as a Space Partitioner

In this section, we introduce our design of a space partitioner based on the KD forest [58].

3.3.1 Overview

KD forest [58] is a group of KD trees that are built together. It is designed to enhance approximate nearest neighbors search quality by building multiple KD trees rather than building a single KD tree. We use the KD forest as the space partitioner part of the distributed index, where each space-partitioning instance is a KD tree, and partitions are the bins associated with the tree leaf nodes.

We are interested in matching data points with high dimensions. Thus, if we use the traditional KD tree, it will be too deep with too many nodes and each has only one data point, which is not efficient especially in distributed processing environment where accessing any node may involve communications over the network. We control the depth of the tree based on the size of the dataset such that the size of bins at the bottom of the tree roughly matches the storage block size of the distributed file system.

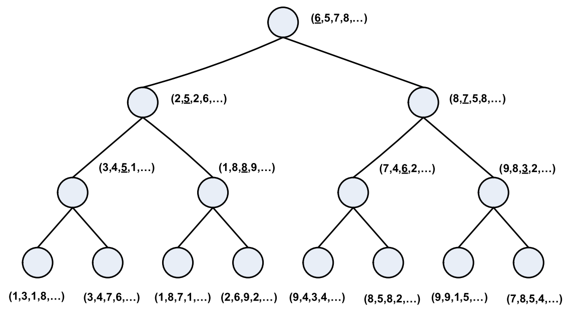
Figure 3.2(a) shows a simple KD tree constructed from 15 high dimensional data points. The figure shows that each node stores a data point and that every level of the tree uses a different dimensions to split the dataset around it. Figure 3.2(b) shows the modified KD tree for the same 15 data points used in Figure 3.2(a). Notice that interior nodes only contain the splitting values and all data points are stored in the leaf nodes. In real deployment, the size of a leaf node is in the order of 64 to 128 MBs, which means that each leaf node will contain thousands of data points. Thus, the size of our top part of the KD tree will be small; only three nodes in this example (compared to 15) and each node stores only one value (compared to a multi-dimensional point).

We construct multiple KD trees to form our KD forest, where each tree is built with different dimensions. Searching multiple trees gives more accurate results than searching one tree. This is because the final results is the the union of all the KD trees results. To avoid the communication and synchronization cost between the KD trees, each KD tree holds a complete copy of the dataset rather than pointers to it as in other implementations [58]. This imposes storage overhead, however, it simplifies the distributed computations and minimizes the communication and network overheads.

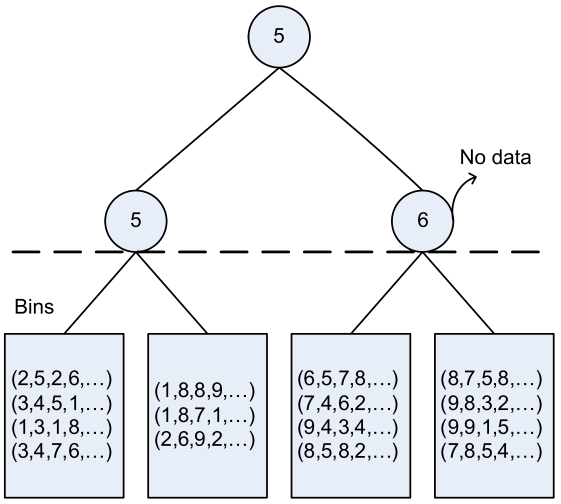
We choose the KD tree as the base for our space partitioner because of its efficiency and simplicity. Different types of trees [54] can be used instead of the KD tree, after we perform our ideas of keeping data points only at leaves, aggregating data points into bins, and storing bins on the distributed file system.

3.3.2 Constructing the KD trees

We construct multiple KD trees to form a KD forest. Since we limit the depth of the trees, trees are built using subset of the dimensions of the data points, and each tree uses a different subset of them. Multiple methods can be used to choose these subsets of dimensions. For example, we may use the dimensions that have the highest variance in the dataset. In order to have high accuracy with using few dimensions, we use the principal component analysis (PCA) to choose the most



(a) Classical KD Tree



(b) Modified KD Tree

Figure 3.2: The Classical KD Tree and the Modified KD Tree.

representative principal components to project the dataset points on them. PCA is a well studied technique for dimensions reduction. It finds a hyperplane of the required target dimensionality to project the actual points on, such that the variance among them after the projection is maximized. It finds this hyperplane by calculating the singular value decomposition (SVD) of the covariance matrix of the input points.

We use a random sample of the reference data points to perform the PCA. Note that the KD forest is used in distributing all data points to partitions as we explain later in Section 3.5.1, however, the construction of the trees is done from a sample of the reference data points.

The following steps are performed to construct our version of the modified KD tree: First, we decide on the number of levels $numLevels$ in the KD tree. This is calculated from the size of the reference dataset $refSize$ and the storage block size $blockSize$ of the distributed file system. If the tree has $numLevels$ levels, then it will have $2^{numLevels}$ leaf nodes or bins, and each bin can store up to $blockSize/pointSize$ data points, where $pointSize$ is the size of each data point. Thus, the the total number of points that can be stored in the distributed index is $2^{numLevels} \times blockSize/pointSize$ which must be greater than or equal to the number of data points in the reference dataset $refSize/pointSize$. Thus, the number of levels is given by: $numLevels \geq \lceil \log_2 (refSize/blockSize) \rceil$.

Second, we construct the KD tree using a depth first algorithm. For each level in the tree, we project all data points in the random sample on the principal component that corresponds to that level. Then, we sort all data points with respect to this principal component, and we use the *median* as the splitting value. Since we use the median value for partitioning, roughly half of the data points will be directed to the left subtree and the other half to the right subtree in each level, which leads to a balanced tree. We say *roughly* half of the data, because the principal component analysis is not performed on the whole dataset. However, randomly choosing a reasonable-size sample will unlikely lead to unbalanced tree. A fully-balanced tree can be achieved by applying PCA on the whole dataset, which is also doable especially that this is done only once on the reference dataset and the tree will be used to answer many queries. Nonetheless, our experiments with large image datasets indicate that random sampling is practically sufficient to provide a balanced tree.

3.3.3 Finding the Closest Bins

In order to store a reference point into the index, we first find its bin in each KD tree. Also to find the best matches for a query point, we first find the closest bins to it in each KD tree. Both tasks we achieve using the GetClosestBins() function.

The GetClosestBins() function returns the requested number of closest bins to a given point in a specified KD tree. If we set the number of requested bins to one, it returns the bin that contains that point. Procedure 1 shows the pseudo code of this function, which is a variant of the *best bin first* algorithm in [7]. The idea of the algorithm is to search the candidate bins in ascending order of their distances to the query point, instead of their original order in the KD tree. This is done by maintaining a priority queue, based on the distance between the query point and the nodes in the tree. The queue is initialized by inserting the root of the tree in it. Then, while the algorithm traverses down the tree to reach the closest bin, it adds more nodes to the priority queue, which will be inspected later to find other close bins. Once it reaches a leaf node at the bottom of the tree, it adds the bin ID corresponding to that leaf node to the list of closest bins. If more bins are still needed, the algorithm traverses the tree again starting from the node on the top of the priority queue. Otherwise, the algorithm returns the list of closest bins.

Unlike the original method for searching multiple KD trees introduced by Silpa-Anan and Hartley [58], we do not maintain a single priority queue for searching the bins. Instead, we maintain a separate priority queue for each KD tree, and search each KD tree independently. Finally, we merge

Procedure 1 Finding Closest Tree Bins to Given Data Point

```
 1: function GetClosetBins(KDTree $kdt$, Point $p$, int $count$)
 2:     List closestBins = []
 3:     PriorityQueue $Q$ = root of $kdt$
 4:     KDTreeNode $currentNode$ = null
 5:     while $Q$ is not empty do
 6:         $currentNode$ = top of $Q$
 7:         while $currentNode$ is not leaf do
 8:             $distance$ = Calculate distance between $p$ and $currentNode$
 9:             if $distance < 0$ then                               ▷ closer to left child
10:                 Add right child of $currentNode$ to $Q$
11:                 $currentNode$ = left child of $currentNode$
12:             else                                                 ▷ closer to right child
13:                 Add left child of $currentNode$ to $Q$
14:                 $currentNode$ = right child of $currentNode$
15:             end if
16:         end while
17:         Add bin ID of $currentNode$ to $closestBins$
18:         if size of $closetBins$ = $count$ then return $closetBins$
19:         end if
20:     end while
21: end function
```

the results at the end of the search from the the KD trees. This significantly reduces the communication overhead between the computing machines compared to synchronizing a shared data structure between the machines. It also makes the searches truly independent and can be fully parallelized. To find B bins from a KD forest with T KD trees, we find the best $\lceil B/T \rceil$ bins from each KD tree. The final result is the union of the points in all the B bins from all the trees after removing the duplicate points if there are any.

3.3.4 Scaling Up/Down the Number of Bins

When the number of points in the index increases/decreases significantly, the KD forest structure of the index needs to be scaled up or down to handle the change of the data size. As we mentioned before, we maintain the partition size to be the storage block size of the distributed file system, so that we can read the whole file in single I/O operation. We scale the KD forest by controlling the number of levels $numLevels$ in each KD tree.

If the number of points increases beyond the initial capacity of the index, the $numLevels$ parameter will be increased by 1, which means doubling the number of bins at the bottom of each tree and hence doubling the total number of points that can be managed by the index. To achieve this, for each tree, we select a dimension to be considered for the new level. Recall that the dimensions are computed by performing PCA on a sample of the reference dataset. If the number of tree levels became greater than the number of dimensions, we iterate again over the dimensions. That is, the dimension for the new level of the tree will be the same as the first level of the tree. Then, we project and sort the data points of each partition based on the value of the new dimension. Then, we compute the median and split each leaf node into two based on it. Similar steps are applied for scaling down the index when more than half of the points are removed. In this case, the number of levels in the tree is decreased by one and each pair of leaf nodes will be merged into one.

3.4 Locality-sensitive Hashing as a Space Partitioner

In this section we introduce a different design of a space partitioner based on locality-sensitive hashing (LSH) from p-stable distributions [21].

3.4.1 Overview

As we mentioned earlier in Section 2.3, the main idea of locality-sensitive hashing is to use hash functions that map similar objects into the same hash bucket with high probability. In our LSH space partitioner, we take each space-partitioning instance to be a hash table, and each partition to be a group of buckets.

Different LSH families can be used to generate different hash codes representations (e.g., Binary or Integer codes) and using different distance functions (e.g., Hamming or Euclidean distances). Focusing on the Euclidean space under l_2 norm, we use Datar et al. [21] proposed LSH families for l_p norms based on p-stable distributions, where the hashing is done through random projections and each hash function is defined as:

$$h^{a,b}(\vec{q}) = \left\lfloor \frac{\vec{a}\ .\ \vec{q} + b}{W} \right\rfloor, \tag{3.1}$$

where $\vec{a}$ is a d-dimensional point with components randomly and independently chosen from a 2-stable distribution. The 2-stable distribution we use is the Gaussian distribution, which works for the Euclidean distance [21]. b is a real number uniformly chosen from the range $[0, W]$. Each hash function $h^{a,b}$ maps a d-dimensional point $\vec{q}$ to an integer. W is the hash bucket width, which controls the number of points in each bucket. Increasing W increases the number of points that fall into each bucket, and decreasing it decreases the numbers of points per bucket consequently. So varying W causes a trade-off between a large table with a large number of small buckets, or a more compact table and fewer buckets but with more points to consider for each bucket in the final search.

We perform M such projections per point, and concatenate their integer mapping to an integer sequence of length M. This sequence of integers is the hash code of the input point, and the set of M such hash functions constitutes a hash table. Two points fall in the same bucket if they have the same hash code, i.e., map to same values in all the M projections.

To further magnify the probability of collision of similar points, and overcome the unlucky event of two similar points falling into two different values in one of the M projections due to quantization, we build L such hash tables. Where in each hash table, we use hash functions of the same form, but created randomly and independently from the Gaussian distribution. Similar to KD forest, to avoid the communication and synchronization cost between the hash tables, each hash table holds a complete copy of the dataset rather than pointers to it as in other implementations [21]. The number of tables L is a trade-off between storage and accuracy. As the more tables we build, the more likely we find the true neighbors, at the price of extra copy of the dataset per each table.

3.4.2 Constructing the Hash Tables

As we mentioned in the overview, we build L locality-sensitive hash tables, each consisting of the concatenation of M hash functions on the form of the Equation 3.1. Each bucket has a hash code of a sequence of M numbers produced by that formula. Unlike the KD tree, we can not control the number of the produced buckets in each hash table, also this scheme of LSH produces a large number of buckets per table. Naively mapping each bucket to a partition and then to a file consumes excessive amounts of storage and network calls. And also keeps the number of files on the distributed file system and their sizes uncontrollable, which violates a strict design requirement in our design of the space partitioner. To make storing and retrieving more practical, we group multiple buckets per partition and store them in a single file in the distributed file system. However, in order to keep the number of points balanced between the files, we do not assign the same number of buckets to each partition. Instead, we keep the partitions to be of equal size by assigning a varying number of buckets to each partition. This is caused by the fact that each bucket has a different number of points in it. The following steps are performed to construct our version of the LSH: First, we decide on the number of partitions $numPartitions$ in a hash table. This is calculated from the size of the reference dataset $refSize$ and the storage block size $blockSize$ of the distributed file system. We maintain each partition size to be slightly smaller than the block size. Thus, the number of partitions is given by: $numPartitions \geq \lceil (refSize/blockSize) \rceil$. Each of these partitions will have roughly $numPoints \approx blockSize/pointSize$ number of points where $pointSize$ is the size of a data point.

Second, we assign a range of constitutive buckets to each partition, so that the total number of points in these buckets is roughly equals the number of points to be assigned to each partition. This is done as follows: we take a sample from the reference dataset, apply the chosen hash functions to it, and assign each point to its bucket. Then we order the buckets by their hash code, and we also count the number of data points fall into each bucket. We maintain a cumulative counts array of the number of points in each bucket. Based on this cumulative array of counts we constructed, we assign range of buckets to each partition, such that partitions have roughly the same number of points ($numPoints$) and consequently the same size. Since we assign a range of buckets to each partition, instead of using the natural order of the buckets, we use a locality-preserving order of them to the partitions. So that, the buckets in each partition have minimal Manhattan distance between their hash codes. Consequently, for each query points, the number of requested buckets for search would lay in a small number of files. We use orderings based on the idea of space-filling curves as it more locality-preserving order than the natural order of multi-dimensional points. Space-filling curves shows good locality of arranging points of low-dimensions to a one dimension order. In our implementation of LSH, we keep the number of projections per table to be a small number, hence, each hash bucket is represented by a few integers. The small number of integers makes it practical to use a space-filling curve to order them in a more locality-preserving way. After applying the space-filling curve order, each bucket's hash code becomes a single integer. Then, we sort the hash buckets by their integer representation and assign them in ranges to partitions.

3.4.3 Finding the Closest Buckets

Similar to the KD forest, in order to store a point, or find its neighbors, we first locate its partition, and the closest partitions to it to search for close neighbors. We use step-wise multi-probe LSH [47] for searching for the closest buckets to a point. The main idea of the multi-probe LSH method is to search multiple buckets per table for each query point q instead of searching only one buckets as in the basic LSH [36, 29]. Ultimately, these buckets should be chosen in a way such that they are likely to contain the nearest neighbors of the query point with high probability. By the definition of locality-sensitive hashing, if a reference point r is close to a query point q but not hashed to the same bucket as q, it is highly likely to be hashed in a bucket that is "near by" the query bucket (i.e., the hash values of the two buckets differ slightly in few components). The goal in an effective multi-probe algorithm is to locate these buckets to consequently increase the chance of finding the points that are close to q [47]. A bucket difference vector is define as a vector of the same dimensionality of the hash values of two buckets, and captures the sign and magnitude of the difference between each two corresponding components of the two buckets' hash codes. For example, assume two buckets with hash codes $(2, 2, 1, 4)$ and $(1, 2, 2, 4)$, the difference vector between the first and the second is $(1, 0, -1, 0)$.

An n-step bucket difference vector has exactly n coordinates that are non-zero. This corresponds to probing a hash bucket which differs in n coordinates from the hash bucket of the query. Based on the property of locality sensitive hashing, buckets that are 1-step away (i.e., only one hash value is different from the M hash values of the query point hash code) are more likely to contain points that are closer to the query point than buckets that are 2-step away. This motivates the step-wise probing method, which first probes all the 1-step buckets, then all the 2-step buckets, and so on [47]. We find all buckets that are n-step away from a query point using a breadth-first search algorithm. The number of buckets searched in an n-step probing sequence is exponential in n, however, the maximum allowed n-steps are typically small (e.g., 1 -- 10 steps) as the first few buckets contain most of the nearest neighbors.

3.4.4 Scaling Up/Down the Number of Partitions

Similar to KD forest design, when the number of points increases beyond the initial capacity of the index, we double the number of partitions and hence doubles the total number of points that can be managed by the index. To achieve this, for each hash table, we split each partition into two partitions. Recall that the ranges of buckets assigned to each partition are based on the number of points in each bucket and their cumulative sum in the sample of the reference dataset. That is, as we do for the initial index construction, we scan each partition's range of buckets and maintain a cumulative sum of the points count. we project and sort the data, then we compute the median point

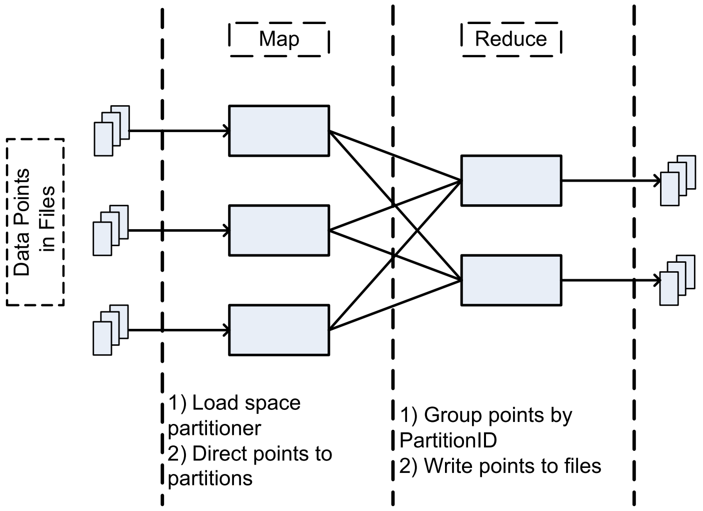

Map
Reduce
Data Points in Files
1) Load space partitioner
2) Direct points to partitions
1) Group points by PartitionID
2) Write points to files

Figure 3.3: Illustration of the Distribute Data MapReduce job.

of that range and split the buckets to two partitions based on it. Similar steps are used for scaling down the index when more than half of the points are removed. In this case, we merge each each two consecutive ranges into one, and join their points into one partition.

3.5 Building the Distributed Index

The distributed index is constructed from the reference R dataset, which is done before processing any queries. Constructing the index involves two steps: (i) creating the space partitioner and (ii) distributing the reference dataset to partitions. The space partitioner is created using a sample from the reference dataset and it is done on one machine. The details are given in Section 3.2 and two concrete implementations are described in Section 3.3 and Section 3.4. Once created, the space partitioner is serialized as one object and stored on the distributed file system. This serialized object can be loaded in memory by various computational tasks running on multiple machines in parallel. Distribution of data is done in parallel on multiple machines using the Distribute Data MapReduce task; details are presented in Section 3.5.1.

Procedure 2 Distribute Data MapReduce Job

MAPPER

```
1: function SETUP                                        ▷ Loaded once per machine
2:     SpacePartitioner $sp$ = LoadSpacePartitioner ()
3: end function
```

// Input: Files containing data points
// Output: List of $[(PartitionID, point)]$

```
1: function MAP(FileID $i$, File $f$)
2:     for each Point $p$ in File $f$ do
3:         $Partitions$ = GetClosestPartitions ($sp$, $p$, 1)
4:         for each $Partition$ in $Partitions$ do
5:             Emit ($Partition$.getID(), $p$)
6:         end for
7:     end for
8: end function
```

REDUCER

//Input: pairs of (PartitionID, point)
//Output: Create a file for each Partition and store all points that correspond to that Partition in it.

```
1: function REDUCE([(PartitionID $pid$, Point $p$)])
2:     Write a file for each partition and and store all points that correspond to that partition in it
3: end function
```

3.5.1 Distributing Data Points

Distribution of data points to partitions is achieved in parallel using the Distribute Data MapReduce job, which is illustrated pictorially in Figure 3.3 and described in pseudo code in Procedure 2.

Data points are assumed to be stored in files. Each data point is a multi-dimensional vector. Each mapper instance will process a group of data points. The mapper starts by loading the space partitioner from the distributed file system. Then, for each data point, the mapper searches the space-partitioning instances to find the closest partitions to this point in each space-partitioning instance using GetClosestPartitions() function. After finding the closest partitions for each point, the mapper emits key-value pairs in the form of (PartitionID, point). Pairs having the same PartitionID are sent to the same reducer by the MapReduce infrastructure. The reducer, in turn, groups all points with the same PartitionID and writes them to a file on the distributed file system, which is identified by the PartitionID.

3.6 Similarity Searching and Matching of Objects

The DIMO system is better suited for processing large query datasets in batch mode. This batch mode is useful for applications such as image/video de-duplication in multimedia databases and video copy detection, in which many data points in the query set are processed together. Extending the DIMO system to process online queries is possible and is part of our future work; this may require optimization of the MapReduce jobs in the system to reduce the overhead involved in executing them for short online queries.

DIMO takes a query dataset and matches it against the distributed index, which represents the reference dataset. This matching process is done in three steps: (i) partitioning query dataset, (ii) finding K nearest neighbors for each data point in the query dataset, and (iii) performing application-specific object matching routines using the found K nearest neighbors. Each of these three steps is executed in parallel on the MapReduce infrastructure.

The first step does not randomly divides the query dataset. Instead, it uses the space patitioner to partition the query dataset such that each group contains a partition ID and a list of data points that are likely to have neighbors in that partition. This partitioning is accomplished using the Partition Queries task, which is similar to the Distribute Data task used in constructing the distributed index (Figure 3.3 and Procedure 2), except for two modifications. The first modification is that the reducer

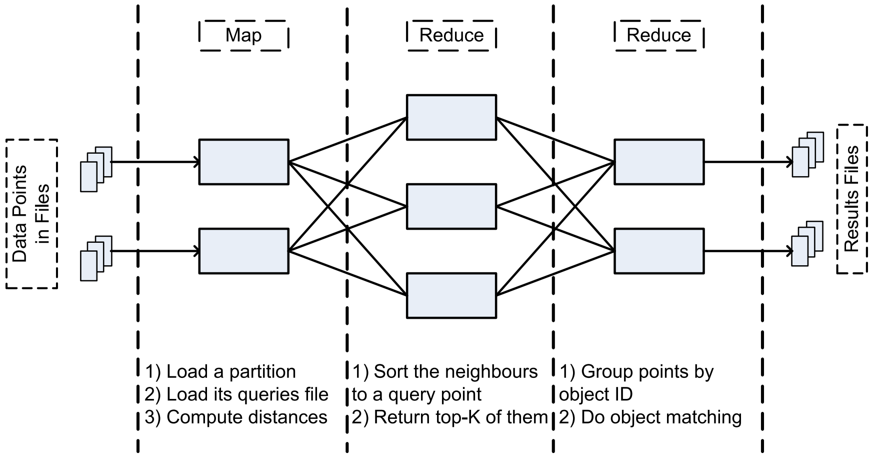


Figure 3.4: Illustration of the second and third steps of the object matching process in the DIMO system.

in the Partition Queries task does not store partitions on the distributed file system. Rather, it emits partition IDs and lists of query data points; one list for each partition ID. The space partitioner is used to create the list of data points that corresponds to each partition, in the same way as described in Section 3.5.1. The second modification is the setting of the number of partitions parameter in GetClosetPartitions() function in line 4 of Procedure 2. While the number of partitions is set to one in the Distribute Data task, since each reference data point is stored in only one partition for each partitioning instance, the number of partitions is variable and used to control the accuracy of the computed K nearest neighbors in the Partition Queries task. If the number of partitions is set to n, then each query data point is compared against all reference points in n partitions, which increases the accuracy but requires more computing resources.

The second and third steps of matching objects are finding the K nearest neighbors and applying application-specific function(s) on them to produce the final object matching results. These two steps are illustrated in Figure 3.4. The figure shows that these two steps are achieved through one MapReduce job that has one mapper and two consecutive reducers.[1] The mapper and first reducer compute the K nearest neighbors for all points in the query dataset. The second reducer is a place holder for any application-specific post processing function on the K nearest neighbors. For example, in a video copy detection application [38], individual matching of query frames with

[1]We note that the current implementation of Hadoop requires having an empty mapper before the second reducer.

reference frames is not sufficient to determine video copies. In this case, the temporal aspects of the frames should also be considered which is done in the second reducer.

The pseudo code of the Matching Objects MapReduce job is shown in Figure 3. The mapper takes the output of the Partition Queries task (the first step) in the form: $(PartitionID, [p1, p2, \dots])$. It then loads the file corresponding to $PartitionID$ from the distributed file system. The distance between every query point in $[p1, p2, \dots]$ and every reference point in the loaded file is computed, and if the distance between any pair of points is less than a pre-defined threshold, this pair will be emitted for further processing in the following reducer(s). Note that every query point can be matched against points in multiple partitions for increasing the accuracy. Thus, the first set of reducers combine the results from all mappers, which are keyed on query points IDs. The reducers then sort neighbors collected from all partitions, and output the nearest K neighbors for each query point. The output of the first set of reducers contains the object ID of the query point as well as the object IDs of the K nearest reference points and the distance between each reference point and the query point. Recall that multiple data points (e.g., SIFT descriptors) can belong to one object (e.g., image) and that we identify all data points of the same object with the ID of that object. Specifically, we give each point an ID that is composed of two components: (ObjectID, Offset), where Offset is an identifier for the point within the object. The output of the first set of reducers is the query object ID as key of the out key-value pair. That is, for a query object all the data points from all candidate reference objects are grouped together and fed to the second set of reducers for any post-processing of the K nearest neighbors desired by the considered application.

3.7 Updating the Distributed Index

The distributed index is built from the reference dataset, which is assumed to change at slower rate than the query dataset. Our design of the index, which aggregates all data points and stores them in partitions that each is mapped to a single file, makes updating the index less complex than other space-partitioning data structures that store data in interior structure. When a data point is added or removed, we only change the end files, and we do not need to manipulate or adjust the internal structure of the space partitioner, which can involve significant computation and communication overheads in distributed environments. In addition, since partitions are stored as separate files on the distributed file system, accessing and updating them can easily be done in parallel, as concurrent

Procedure 3 Object Matching MapReduce Job

MAPPER

//Input: PartitionID and list of query points
//Output: List of $[(pointID, neighbor)]$, where $neighbor = (pointID, distance)$

```
1: function MAP(PartitionID $pid$, Point $qList[]$)
2:     Point $rList[]$ = Load points from file corresponding to $pid$
3:     for each $q$ in $qList$ do
4:         for each $r$ in $rList$ do
5:             $d$ = Calculate distance between $q$ and $r$
6:             if $d \leq$ threshold then
7:                 Emit($q$.getPointID(), ($r$.getPointID(), $d$))    ▷ We emit only point IDs to reduce communications overhead
8:             end if
9:         end for
10:    end for
11: end function
```

REDUCER 1

//Input: List of pair of points and distance between them
//Output: List of $[(pointID, [(neighborID, d)])]$

```
1: function REDUCE(PointID $qID$, Neighbor $nList[]$)
2:     Sort $nList$ based on distance to $q$
3:     Emit ($qID$, $[nList[1], nList[2], \ldots, nList[K]]$)
4: end function
```

REDUCER 2

//Input: List of $[(objectID, [(neighborID, d)])]$
//Output: Application Specific

```
1: function REDUCE(ObjectID $qID$, Neighbor $knnList[]$)
2:     Application-specific processing
3: end function
```

file accesses are managed by the distributed file system. Furthermore, the sizes of files containing the data points are used to monitor the balance of the index and whether restructuring of the index is needed, as will be explained below.

3.7.1 Adding/Removing Points

For adding/removing a data point, we first traverse the space partitioner to find the partition that this point belongs to. Then, the file corresponding to that partition is accessed through the file system and the point is added/removed. For adding/removing multiple points (batch mode), partition identification of all points is done first using the space partitioner. Then, all points that belong to the same partition are added/removed to the corresponding file at once. In actual deployment of the DIMO system, the number of partitions should be conservatively chosen such that the partitions are not fully filed with data after the first construction of the index. Thus, since the partitions are relatively large (64 to 128 MBs each), the internal structure of the index will not be impacted by small updates.

3.7.2 Monitoring Index Imbalance

If there are major updates to the index which involve adding/removing sizable fractions of the reference points, the balance of the index can be affected, especially if the added/removed data points change the distribution of the reference data points. For example, if the added/removed data points make the data distribution more/less skewed. An imbalance in the index can result in some partitions become quite large while others are empty or have few points. This means that large partitions would require multiple storage blocks of the distributed file system to be read and processed, which translates to longer processing times. Whereas little processing is performed for small partitions, while the system still pays the I/O overhead to access the almost-empty blocks of the distributed file system. The imbalance in the index can easily be detected by monitoring the sizes of the partitions on the distributed file system. If the partition size distribution significantly deviates from the expected uniform distribution, the index should be re-built from scratch. The deviation from the uniform distribution is quantified by measuring the variance in the partition sizes and if it exceeds a pre-defined threshold, the index re-building process is initiated.

Chapter 4

Evaluation

In this chapter, we evaluate the performance of the proposed DIMO system using both K-dimensional forest (KD Forest) and multi-probe locality-sensitive hashing (MP-LSH) implementations of the space partitioner. We also compare it against the closest system in the literature. We note that DIMO is designed to be a general object matching system that can work for different applications. All applications built on top of DIMO rely on the accuracy of the nearest neighbors computed by DIMO, while these applications may use different metrics to assess their performance. Thus, in our experiments, we focus on evaluating the accuracy of the nearest neighbors computed by DIMO and we do not consider application-level performance metrics as they vary from one application to another.

We first describe our experimental setup and datasets. Then, we compare the nearest neighbors computed by DIMO versus the ground-truth neighbors. Then, we compare our system versus the best results reported by the RankReduce system [64]. Then, we analyze the elasticity and scalability of DIMO. Finally, we analyze various overheads imposed by DIMO.

4.1 Experimental Setup and Datasets

4.1.1 Platform

We have fully implemented the DIMO system with both KD forest and MP-LSH space partitioners using Java 1.7 and Apache Hadoop 1.0.3 [31]. We conduct several experiments on clusters of various sizes from the Amazon Elastic MapReduce (EMR) cloud service. To show the elasticity and scalability of our system, we conduct experiments on EMR clusters of sizes 8, 16, 32, 64, and 128 machines. Each machine in the cluster is an Amazon EC2 Medium Instance, which has 3.75 GB of memory, 2 EC2 Compute Units, and 410 GB storage, and it runs 64-bit Debian 6.0.5 (Squeeze) linux as its operating system. In addition, we have created a virtual Hadoop cluster on a single machine in our lab. This machine is a Dell T1600 server with 8-core Intel Xeon E3-1245 3.3 GHz processor, and 16 GB main memory. The operating system is 64-bit Ubuntu linux 12.04. We used this virtual Hadoop cluster for small-scale experiments.

4.1.2 Datasets

We assess the performance of the DIMO system using data points extracted from images. We emphasize that DIMO is a general system and we use data from images as an example of real data. We extract SIFT [45] features from images and use each SIFT feature descriptor as a data point. We use the VLFeat 0.9.17 [68] implementation of the SIFT algorithm. On average, we extract 200 SIFT features from each image, and each feature has 128 dimensions. We use two image datasets in our experiments:

- Caltech Dataset [4]. This dataset is composed of data points extracted from images. The images are obtained from the Caltech Buildings and Game Covers datasets. Caltech Buildings is a set of images taken for 50 buildings around the Caltech campus. Five different images were taken for each building from different angles and distances, giving a total of 250 images. Caltech Game Covers is a set of CD/DVD covers of video games. It includes around 11,400 images for games on different consoles. We extract around 100 -- 200 data points from each image. Thus, this dataset contains 1,000,000 reference data points. The query set contains 1,000 data points randomly selected from the 1,000,000 points. We refer to this dataset as the small dataset and it is used to compare the nearest neighbors computed by our system versus the actual nearest neighbors (ground truth), which are computed using an expensive brute force approach that tries all possibilities.
- ImageNet Dataset [24]. ImageNet is an open image database with millions of images organized according to the WordNet hierarchy, where each group of images illustrate a concept in WordNet. We downloaded 1 million images from ImageNet. With 200 data points extracted

from each image, this dataset contains 200 million data points and it is used to test the scalability of our system. The query set contains 100,000 data points randomly selected from the 200 million points.

4.1.3 Performance Metrics

The main goal of the DIMO system is to provide high quality nearest neighbors with low cost. The quality of the retrieved K nearest neighbors for a query point q is assessed by the percentage of true KNNs found, and the cost is assessed by the percentage of the dataset scanned for finding the KNNs.

To formalize the performance measures, let q be a query point, we define $T(q)$ to be the set of true KNNs which is calculated by a full scan brute force algorithm of the reference dataset. We also define $C(q)$ to be the candidate set which is the aggregation of all reference points in the candidate partitions, and to be the shortlisted candidate set after sorting the candidate points in $C(q)$, removing the redundant points, and selecting only the top K among them.

We measure the $Recall(q)$ as the percentage of $T(q)$ appeared in $S(q)$, which is given by:

$$Recall(q) = \frac{|S(q) \cap T(q)|}{|T(q)|}. \tag{4.1}$$

The precision of the selected points of $S(q)$ is given by:

$$Precision(q) = \frac{|S(q) \cap T(q)|}{|S(q)|}. \tag{4.2}$$

Since we sort $C(q)$ and choose the top K candidates and put them in $S(q)$, then the precision is the same as recall, as the size of $S(q)$ equals the size of $T(q)$. Recall that $T(q)$ contains the true K nearest neighbors, and $S(q)$ contains the closest K candidates to the query point.

We compute the average recall of the retrieved K nearest neighbors across all points in the query set Q, and it is given by:

$$AvgRecall = \frac{\sum_{i=1}^{|Q|}\{Recall(i)\}}{|Q|}. \tag{4.3}$$

The main computationally-expensive part of finding the nearest neighbors is scanning through the candidate set and maintaining the closest K of them to the query point, and this is proportional to the size of the candidate set. Thus, the percentage of the whole dataset scanned is a good indicator of query search time. We define the $selectivity(q)$ as the ratio between the candidate set size and the reference set size, and it is given by:

$$Selectivity(q) = \frac{|C(q)|}{|R|}, \tag{4.4}$$

where R is the set of reference points.

We compute the average selectivity of the retrieved K nearest neighbors across all points in the query set Q, and it is given by:

$$AvgSelectivity = \frac{\sum_{i=1}^{|Q|}\{Selectivity(i)\}}{|Q|}. \tag{4.5}$$

We use the $AvgRecall$ metric in our experiments as the measure of search quality, and the $AvgSelectivity$ as the measure of search cost.

In addition, we measure various other performance metrics, including the total running time, and the amount of data exchanged over the network.

4.2 Parameter Tuning

In this section we describe how we tune the system parameters for different accuracy requirements and allowed computational resources. We study the behavior of our techniques on the small dataset, and we estimate the expected behavior on the large ones.

4.2.1 KD Forest Parameter tuning

The main parameters to tune in the KD forest are the number of trees T, the tree height H, and the number of tree bins to scan for a given query point B. The number of trees controls how many KD trees in the KD forest. Larger number of trees gives more accuracy at the price of extra copy per tree. However as shown by Muja and Lowe in [50], the accuracy gain starts to fall quickly after few number of trees (10 -- 15 trees). Thus, we set the number of trees as the maximum number of copies that can fit in the allocated storage with an upper bound of 15 trees.

The tree height, number of tree levels, is used to control the maximum depth of each tree and consequently the total number of the index bins. Note that, each leaf node of a tree is associated with a single file on the distributed file system that contains all the points assigned to its bin. The tree height should be chosen carefully, as a small number of levels leads to a small number of leaf nodes associated with large files with large number of points in it to compare against and consequently long execution time. Similarly, a large number of levels leads to a large number of leaf nodes associated with small files with small number of points in each to compare against which leads to low recall. As we showed in Section 3.3.2, we choose the number of tree levels that leads to an average file size that can fit in one distributed file system block, so that we can read the whole file in single I/O operation.

The number of scanned bins controls the fraction of index points to be scanned. It is the number of bins considered for searching for each query point. We configure the number of bins with respect to the required accuracy. Increasing this parameter improves the accuracy as it compares the query points against larger number of reference points, consequently it affects the running time as well. The number of scanned bins is a key input parameter to our KD tree as it controls the trade-off between accuracy and running time. For example, if we set the number of considered bins to the total number of bins, the search algorithm will visit all bins which takes longer running time but all points will be scanned, and exact solution is guaranteed. Our experiments shows that when scanned bins represent $1 - 2\%$ of total numbers of bins, we achieve around $80 - 90\%$ average accuracy.

While the number of trees and the tree height are chosen at the KD forest construction time, the number of scanned bins can be changed during the query time to optimize for different queries and different quality and cost requirements.

4.2.2 Multi-probe LSH Parameter tuning

To tune the parameters of our LSH, we use an approach similar to the one proposed by Dong et al. [25]. In our implementation of LSH, there are four main parameters that need to be tuned for good performance. Theses parameters are the number of hash tables L, the number of projections M, the hash bucket width W, and the number of allowed steps in the multi-probe search N-steps. Among them, L, M and W need to be chosen at the building time of the hash tables. However N-steps can either be chosen offline while building, or online at querying and changed from query to query.

According to our method, larger number of tables L always results in higher recall, at the price of extra copies of the reference dataset. Thus L should be tuned to the maximum affordable value, as limited by the storage available. Note that in practice if L is really large ($L \gg 20$, which is not likely to happen for large datasets), query time will again start increasing as the cost to generate the probing sequences becomes significant.

We tune M and W for best values when creating the hash tables, while N-steps can be adaptively changed during the query time. However, to get a good estimate of the performance when building the hash tables, we need a fixed N-steps value to predict the performance to tune M and W accordingly. To overcome this, we choose a fixed N-steps value of medium size to tune for near-optimal M and W. Setting N-steps = $M/2$ is a reasonable choice, as the first few buckets are the most promising and having most of the true matches as shown by Lv et al. [47].

Now we need to optimize the values of M and W given fixed L and N-steps search. We can see the problem as minimizing the selectivity with the condition that the expected recall is above a given threshold $1 - \delta$ where δ is the allowed expected error. In our implementation, we use the following approach to find the best parameters M and W given the previous conditions. First, assuming M is fixed, then the relationships between W and both selectivity and recall are monotonicly increasing. So, the W value that gives average recall greater than or equal to the target recall can be found using binary search, and the selectivity of this W is the minimum achievable given the restriction on the recall. Since M is discrete, we enumerate the value of M from 1 to some reasonably large value (20 is a reasonable maximum) and for each value of M we get the best W that gives recall greater than or equal to the target recall. Finally, among the calculated pairs of (M, W), we choose the pair (M, W) with minimum average selectivity.

At query time, we may change N-steps to different values. N-steps is a trade-off between recall and selectivity, we can change it for every query batch on the same built reference set.

We note that there are other methods to tune the parameters of LSH. For example, Sundaram et al. [65] present a memory-based implementation and they use a slightly different approach where there are constraints on the total memory used. Slaney et al. [61] use a mathematical approach to model the data distribution, and then fine tune the LSH parameters accordingly. Their methods can also be used in our system.

4.3 Comparison against Ground Truth

In this section, we compare the accuracy of the returned nearest neighbors by DIMO against the true nearest neighbors that are computed by a brute force algorithm. We also measure the selectivity of the DIMO system for retrieving the nearest neighbors. The true nearest neighbors are computed

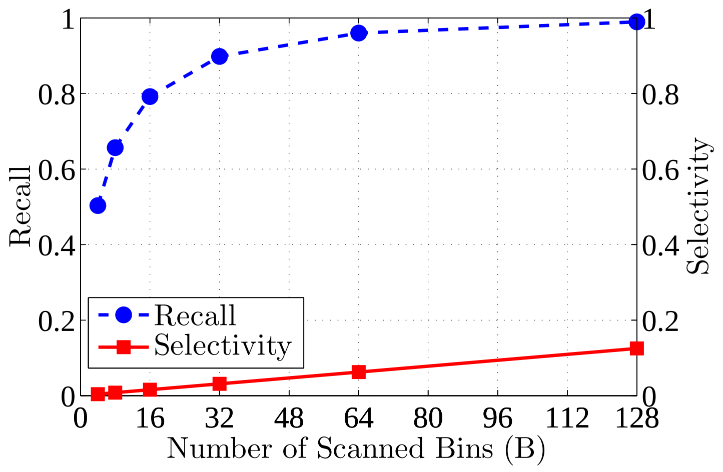


Figure 4.1: The effect of varying the number of scanned bins (B) of the KD forest on both the selectivity and recall.

by calculating the distance between each pair of reference point and query point. Since computing true neighbors is computationally expensive and not possible for large datasets, we use the small Caltech Dataset in these sets of experiments. This dataset has 1,000,000 reference points and 1,000 query points. We calculate the $AvgRecall$ and $AvgSelectivity$ for the retrieved K nearest neighbors. Unless stated otherwise, we use $K = 10$ for the experiments. We evaluate DIMO system performance using both KD forest and multi-probe LSH implementations of the space partitioner.

4.3.1 KD Forest

In these sets of experiments we show the practical aspects of building the index using a KD forest as its space-partitioning data structure. We show the effect of changing multiple parameters and their impact on the system performance. We also show how to trade-off the resources to achieve the target performance.

Effect of Number of Scanned Bins B

In this experiment, we show the effect of changing the number of scanned bins on both recall and selectivity of the KD forest space partitioner. In Figure 4.1, we plot the $AvgRecall$ and the $AvgSelectivity$ on both sides of the y-axis, and we vary the number of scanned bins on the x-axis.

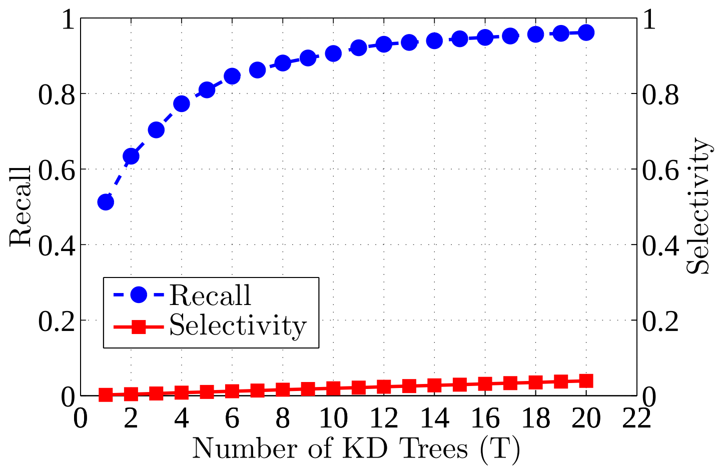


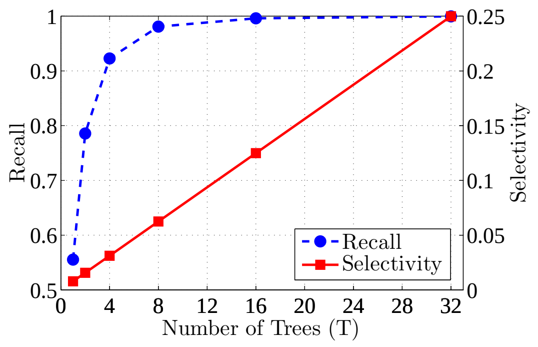
Figure 4.2: The effect of varying the number of trees T in the KD forest on both the recall and selectivity.

To focus on the effect of the number of bins, We build a KD forest of one KD tree with 10 levels, and having 1,024 leaf bins.

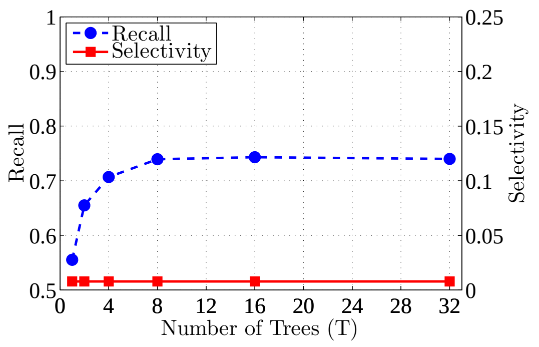
We scan different number of bins at values of 4, 8, 16, 32, 64 and 128 bins out of the 1,024 total bins. The scanned bins represent selectivity percentages of 0.39%, 0.78%, 1.5%, 3.125%, 6.25%, and 12.5% of the data size, respectively. We measure the average recall and selectivity of 1,000 query data points against the 1,000,000 reference data points. The results show that DIMO using the KD forest achieves high recall with low selectivity. For example, when we scan 16 bins out of 1,024 representing selectivity of only 1.5% of the data size, the recall is about 80%. In addition, when we set the number of scanned bins to 64, DIMO achieves an average recall of more than 95% at the price of only scanning 6.25% of the bins. This means that, on average, 95% of the true K nearest neighbors are found in the returned K neighbors by DIMO. We note that the number of scanned bins controls the trade-off between the average accuracy achieved and the computing resources needed, which makes DIMO suitable for various applications with different requirements.

Effect of Number of Trees T

In this experiment, we show the effect of changing the number of KD trees in the KD forest on both the selectivity and recall. In Figure 4.2, we plot the $AvgRecall$ and the $AvgSelectivity$ on both



(a) Varying number of trees with variable number of scanned bins.



(b) Varying number of trees with fixed number of scanned bins.

Figure 4.3: The effect of varying the number of trees T with fixed and variable number of scanned bins on the recall and selectivity.

sides of the y-axis, and we vary number of KD trees on the x-axis. In this experiment we build each tree with 13 levels with different dimensions, each tree has 8,192 leaf bins. For each tree we scan the same number of bins which is 16 bins representing only 0.19% of the total number of bins per tree. We build different number of trees varying from 1 tree up to 20 trees; each tree cost an extra copy of the dataset. We measure the average recall and selectivity of 1,000 query data points against the 1,000,000 reference data points.

The results show that increasing the number of trees significantly increases the recall with slight increase in the selectivity. For example, when we build one tree we get 51% recall with 0.19% selectivity. However, if we build four trees we get 77% recall with 0.78% selectivity, that is, 27% increase in recall with only 0.59% increase in the selectivity. If we increase four more trees the recall increase to 88% with only 1.5% total selectivity for all the eight trees. We note that after certain number of trees (10 -- 15), the gain in the recall of building more trees starts to fall significantly while the selectivity and storage continue to grow linearly with the number of trees.

In Figure 4.3, we conduct a similar experiment to show the comparison between changing the number of trees with increasing number of scanned bins, and with fixed number of scanned bins. We also plot the $AvgRecall$ and the $AvgSelectivity$, and we vary the number of trees on the x-axis. In this experiment we build each tree with 13 levels having 8,192 leaf bins. In the experiment plotted in Figure 4.3(a), we scan 64 bins per tree. That increases the total number of scanned bins of the whole KD forest with increasing the number of trees, and consequently increases the selectivity

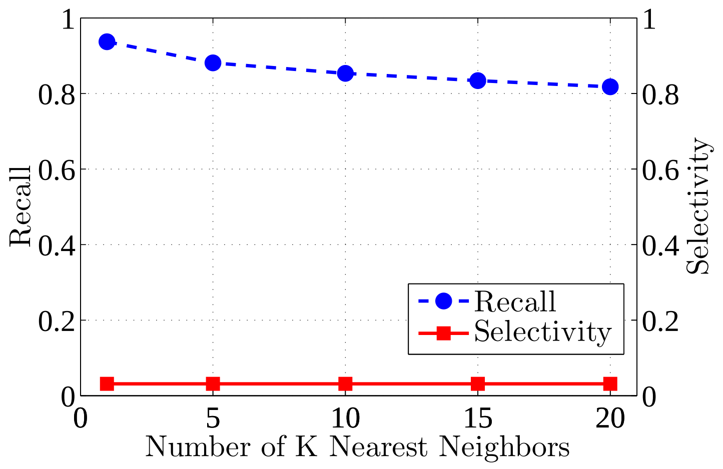


Figure 4.4: The effect of varying the number of K nearest neighbors on the recall while keeping fixed selectivity.

and the running time. As shown in Figure 4.3(a), more trees gives higher recall but with increasing selectivity as well. In the experiment plotted in Figure 4.3(b), we scan a total of 64 bins for all trees in the KD forest, which keeps a fixed selectivity and running time while changing the number of trees. Figure 4.3(b) shows the trade-off between recall and storage while keeping the running time fixed. Note that, with each extra tree we store extra copy of the data. After few number of trees (8 trees) the increase in the recall tends to be zero.

We note that changing the number of trees controls the trade-off between the storage and the recall given that the total number of bins is fixed, which again makes DIMO suitable for various applications with different running time, storage, and accuracy requirements.

Effect of Number of K Nearest Neighbors KNN

In this experiment, we study the effect of changing the number of the retrieved K nearest neighbors. We create a KD forest of 4 KD trees, and we scan 32 bins per tree for all values of K. We measure the average recall at different values of K while maintaining a fixed selectivity.

The results are plotted in Figure 4.4 and show that we achieve high recall for returning the closest neighbor (i.e., $K = 1$), with value of 94%. The results also show that the average recall achieved by DIMO using the KD forest is not significantly impacted by increasing K. For example, at $K = 5$ the average recall is 88%, and at $K = 20$ the average recall is 82%, losing only 6% of the recall when returning 4 times more neighbors. Note that we fix the selectivity for all K values, we can maintain the same recall for different K values if we allow the selectivity to increase with it.

Summary of the KD Forest Results

In the previous experiments we evaluated the performance of DIMO using the KD forest space partitioner. The experiments showed that the KD forest achieves high recall with low selectivity. We report for 1.5% selectivity around 80% recall using only one tree, and around 88% using eight trees. We showed how we can trade-off between storage and accuracy while having fixed running time by using different number of trees with fixed number of total scanned bins. We also showed how to trade-off between accuracy and running time while having fixed storage by scanning different number of bins for the same number of trees. Finally, we showed that our KD forest space partitioner is robust against changing the number of the retrieved K nearest neighbors.

Our experiments show that a choice of 4 to 8 trees and 1% to 3% selectivity is reasonable for most applications achieving from 80% to 95% average recall on our dataset, which is a good representative of general multimedia datasets.

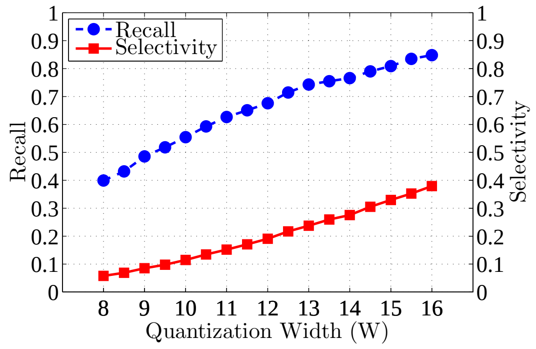
4.3.2 Locality-sensitive Hashing

In these sets of experiments we show the practical aspects of building the index using a locality-sensitive hash tables as its space-partitioning data structure. We show the effect of changing multiple parameters and their impacts on the system performance. We also show how to trade-off the resources to achieve the target performance.

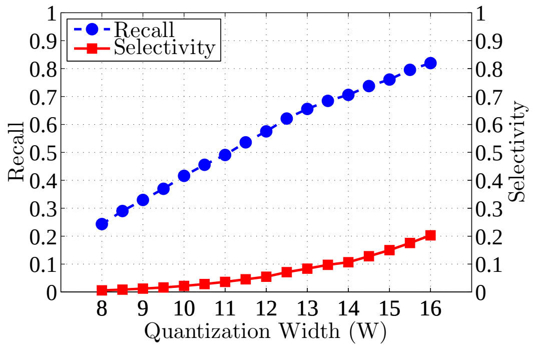
Effect of Number of Projections M and Bucket Width W

In this set of experiments, we show the effect of changing the bucket width W and the number of projections M on both the recall and selectivity. As shown in Figure 4.5, we plot the average recall on the left y-axis and the average selectivity on the right y-axis. On the x-axis, we vary the bucket width used in building the hash tables. We fix the number of hash tables to one, and the N-steps to $M/2$.

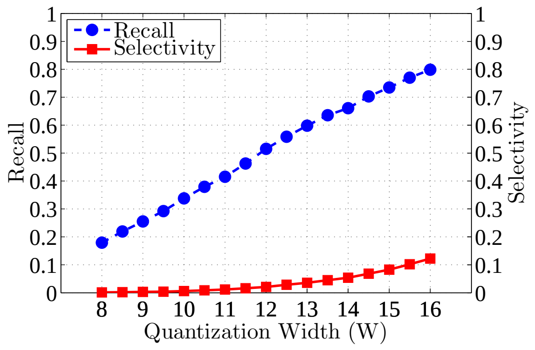
We change the bucket width W values from 8 to 16, and we repeat the experiment with different



(a) Varying W at $M = 4$



(b) Varying W at $M = 8$



(c) Varying W at $M = 12$

Figure 4.5: The effect of varying the bucket width W and the number of projections M on both the recall and selectivity.

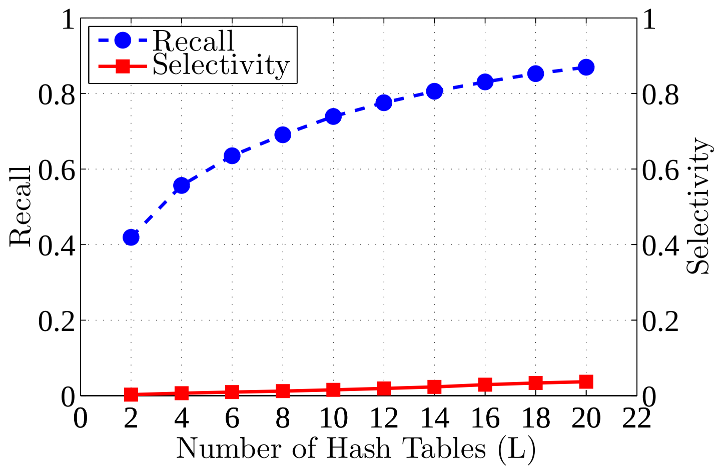


Figure 4.6: The effect of varying the number of hash tables L on both the recall and selectivity.

number of projections $M = 4, 8$ and 12. We measure the average recall and selectivity of 1,000 query data points against the 1,000,000 reference data points.

As we explained in Section 4.2.2, both recall and selectivity increase monotonicly with W. Increasing M results in higher recall for the same selectivity. For example, for 10% selectivity, $M = 4$ gets a recall of around 50%, $M = 8$ gets a recall of of around 70%, and for $M = 12$ gets recall of around 80%. However, the number of explored buckets is exponential in M, thus we can not increase M to much higher values.

We note that the the parameters W and M control the trade-off between the average recall achieved and the required selectivity. We give in Section 4.2.2 an algorithm to automatically set W and M to achieve the minimum expected selectivity for a given target recall.

Effect of Number of Hash Tables L

In this experiment, we show the effect of changing the number of hash tables on both recall and selectivity. In Figure 4.6, we plot the $AvgRecall$ and the $AvgSelectivity$ on both sides of the y-axis, and we vary number of hash tables on the x-axis. We build each table with number of projections $M = 12$ and bucket width $W = 9$. For each table we search the same number of hash buckets within the range of N-steps = $M/2 = 6$. The number of buckets probed represent a slightly different selectivity ratio of each hash table due the nature of the random projections.

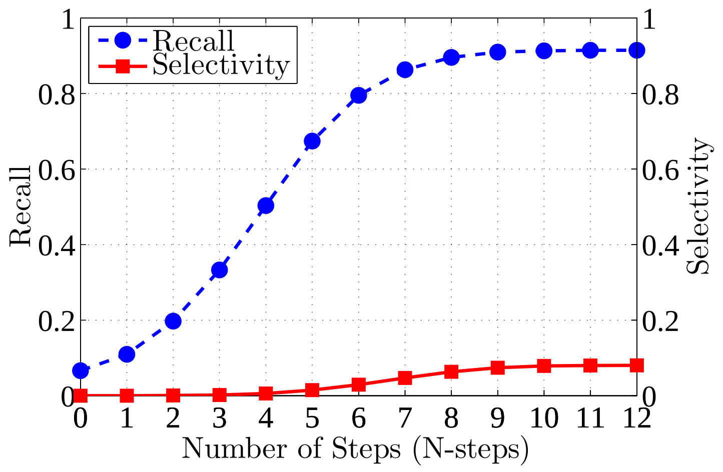


Figure 4.7: The effect of varying the number of steps (N-steps) for searching buckets on both the recall and selectivity.

We build different number of hash tables varying from 2 up to 20 tables, each hash table costs an extra copy of the dataset. We measure the average recall and selectivity of 1,000 query data points against the 1,000,000 reference data points.

The results show that increasing the number of hash tables increases the total recall with a slight increase in the selectivity. For example, when we build two hash tables we get 41% recall with 0.26% selectivity. However, if we build eight tables we get 69% recall with 1.2% selectivity, that is, 28% increase in recall with only 0.94% increase in the selectivity. If we increase eight more hash tables the recall increases to 83% with only 2.9% total selectivity for all the sixteen hash tables.

Effect of Multi-probe LSH with Different N-Steps

In this experiment, we show the effect of changing the number of probed hash buckets by the changing the allowed range of search N-steps on both recall and selectivity. In Figure 4.7, we plot the $AvgRecall$ and the $AvgSelectivity$ on both sides of the y-axis, and we vary number of N-steps of the considered hash buckets on the x-axis. We build a locality-sensitive hashing index with four hash tables, each one is built with number of projections $M = 12$ and bucket width $W = 11$.

We scan different number of buckets with different N-steps at values from 0 to 12. The considered buckets represent an increasing percentage of the total buckets. We measure the average recall

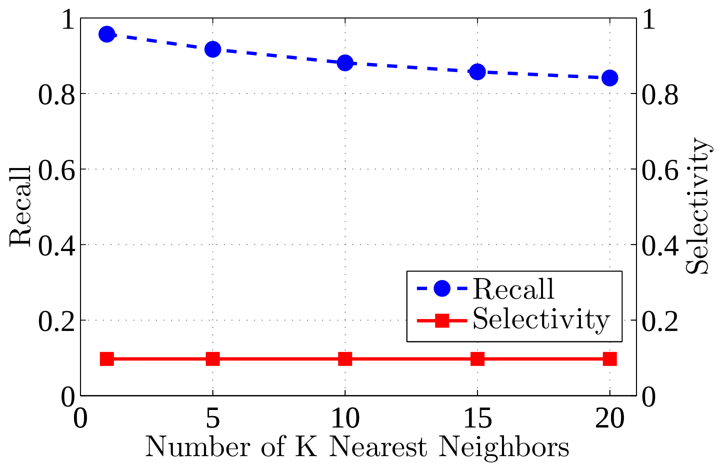


Figure 4.8: The effect of varying the number of K nearest neighbors on the recall while keeping fixed selectivity.

and selectivity of 1,000 query data points against the 1,000,000 reference data points. The results show that DIMO using MP-LSH achieves high recall with low selectivity. For example, when we scan the buckets at most 6-steps away with selectivity representing only 2.9% of the data size, the recall is about 80%. In addition, when we set the number of steps to 8, DIMO achieves an average recall of more than 90% at the price of only 6.3% average selectivity.

We note that the number of N-steps controls the trade-off between the average recall achieved and the selectivity. We also note that the number of N-steps can be chosen dynamically during the query time with different value for each query batch. This makes DIMO suitable for various applications with different requirements.

Effect of Number of K Nearest Neighbors KNN

In this experiment, we study the effect of changing the number of the retrieved K nearest neighbors. We create an MP-LSH index of 8 hash tables, and we use $M = 10$ projections, and bucket width $W = 11.00$ per table for all values of K. We measure the average recall at different values of K while maintaining a fixed selectivity.

The results are plotted in Figure 4.8, which show that we achieve high recall for returning the closest neighbor (i.e. $K = 1$), with value of 96%. The results also show that the average recall achieved by DIMO using the MP-LSH is not significantly impacted by increasing K. For example, at $K = 5$ the average recall is 92%, and at $K = 20$ the average recall is 84%, losing only 8% of the recall when returning 4 times more neighbors. Note that we fix the selectivity for all K values, we can maintain the same recall for different K values if we allow the selectivity to increase with it.

Summary of LSH results

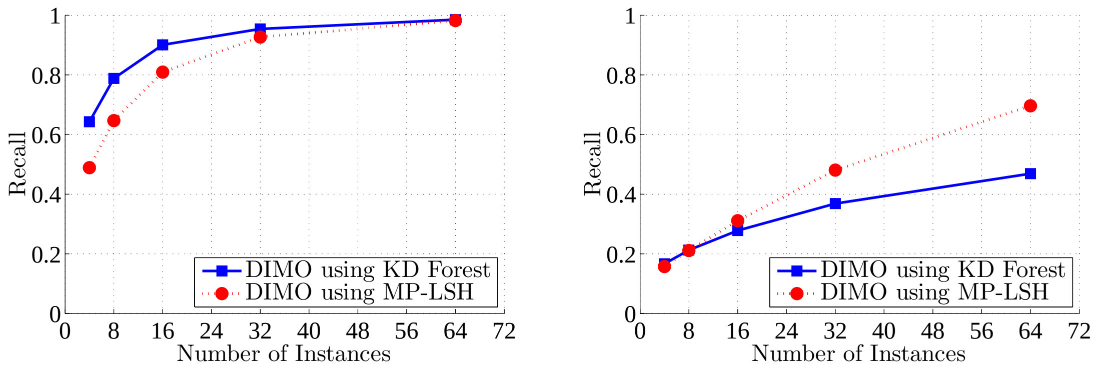
In the previous experiments we evaluated the performance of DIMO using the multi-probe locality-sensitive hashing (MP-LSH) space partitioner. The experiments showed that the MP-LSH achieves high recall with low selectivity. We report average recall from 80% to 90% using 2% to 5% average selectivity from only 8 to 16 hash tables. This is a better performance than the original LSH algorithm the requires to build hundreds of hash tables [21]. We showed how we can trade-off between storage and accuracy by using different number of hash tables. We also showed how to trade-off between accuracy and running time by scanning different number of N-step away hash buckets. We also showed that our MP-LSH algorithm is robust again changing the number of the retrieved K nearest neighbors.

Our experiments show that a choice of 8 to 16 hash tables and 2% to 5% selectivity is reasonable for most applications achieving from 80% to 90% average recall on our dataset, which is a good representative of general multimedia datasets.

4.3.3 Comparison between KD Forest and Multi-probe LSH

In this section, we discuss the difference between the KD forest and the multi-probe LSH space partitioners. As we discussed in Section 2.2 and Section 2.3, the main aspect that differentiate the performance of the two algorithms is the dimensionality of the data and the correlation between the dimensions. We run a series of experiments on both algorithms with different data dimensionality. Figure 4.9 shows the achieved recall by each of them on different datasets at varying numbers of trees and tables. We tune both algorithms parameters to the best values using the techniques we mentioned in Section 4.2. We build an equivalent number of trees and tables for each algorithm, with fixed selectivity of 0.07% per each tree/table.

In Figure 4.9(a), we use a reference set of 100,000 128-dimensions SIFT descriptors versus a query set of 1,000 points chosen randomly from the reference set. The plot shows that the KD forest has a better performance on such dataset with moderate dimensionality; 128-dimensions.



(a) Performance of both techniques on data points of 128 dimensions SIFT descriptors.

(b) Performance of both techniques on data points of 512 dimensions randomly generated points.

Figure 4.9: Comparison between KD forest and multi-probe LSH performance on different datasets with different dimensionality while varying the number of instances.

This is also because of the principal component analysis (PCA) we perform on the dataset, as the image descriptors has high correlation between their dimensions. The PCA algorithm exploits this correlation between the dimensions, resulting in a better performance of KD forest compared to the MP-LSH which does not contain such mechanism.

In Figure 4.9(b), we use a reference set of 100,000 512-dimensions points randomly generated from a uniform distribution versus a query set of 1,000 points chosen randomly from the reference set. The plot shows the superiority of the MP-LSH over the KD forest algorithm for this dataset with higher dimensionality; 512-dimensions. This is because the randomization in creating the LSH tables. The analysis of this randomization shows that the LSH results are not highly dependent on the data dimensionality [36] as in the earlier methods. This is also because the PCA fails to capture any linear correlation between the data point dimensions. This significantly affects performance of KD forest compared to the MP-LSH.

In summary, the main test to determine either to use KD forest or general tree-based approaches versus MP-LSH or general hashing approaches, is the data dimensionality, and the correlation between the dimensions. KD forest shows better performance with correlated data in low to mid dimensions (100 -- 200). However, MP-LSH shows better performance with uncorrelated data in higher dimensions (100s or 1000s).

4.3.4 Summary of the Results

So far we presented two methods and implementations of a space partitioner. We first presented a KD forest index and we showed its parameters, how to tune it, and their effects on the DIMO system. We also presented a multi-probe LSH index and we also showed its parameters, how to tune it, and their effects on the DIMO system. We presented a comparison between both of them, and which is better for different data dimensionality. Both techniques showed high performance in different situations.

In summary, we introduced a flexible design of space partitioners for different requirements and computing machines capabilities. We have three main aspects of the execution, and we can trade-off between them in a systematic way which are: accuracy, running time, and storage.

At construction time, we can trade-off between the accuracy and the storage while having fixed running time by having multiple space-partitioning instances; KD trees in the KD forest design, or hash tables in the MP-LSH design. We can also trade-off between the accuracy and the running time while having fixed storage by building a finer-grain partitioning. This is done by changing the tree height (H) in the KD forest design, or by changing the number of projections (M) and the bucket width (W) in the MP-LSH design. Generally, more fine grain partitioning gives more accurate results at the price of longer running time.

At query time, we can trade-off between the accuracy and the running time by changing the number of scanned bins (B) in the KD forest design, or by changing the maximum allowed N-steps bucket search in the MP-LSH design. Generally, scanning more partitions gives more accurate results at the price of longer running time.

4.4 Comparison against RankReduce

We compare the proposed DIMO system against the closest one in the literature, which is RankReduce [64]. RankReduce implements a distributed LSH index. It maintains a number of hash tables over a set of machines on a distributed file system, and it uses MapReduce for searching the tables for similar points. We compare the results achieved by DIMO against the *best* results mentioned in [64] using the same dataset and the same settings. We did not implement RankReduce; rather we use the the best stated results in its paper. We use the same dataset size of 32,000 points extracted from visual features of images. The visual features used are 64-dimensional color structure feature

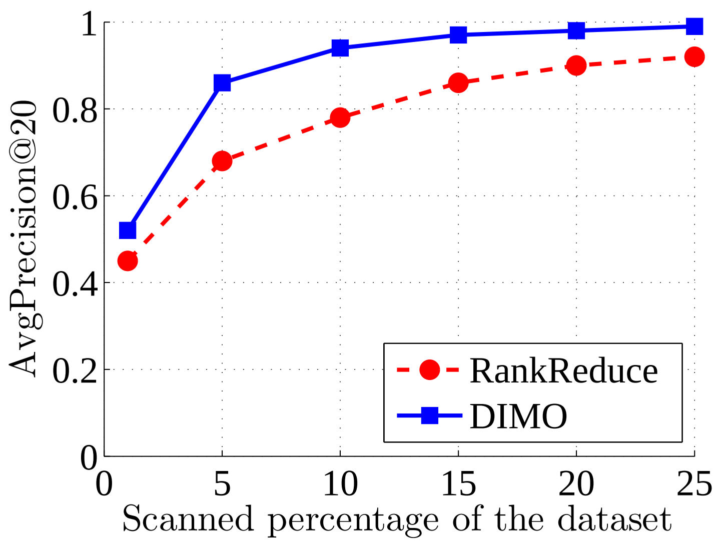


Figure 4.10: Comparing DIMO versus the closest system in the literature, RankReduce.

vectors. We build a space partitioner using the KD forest design, using only 1 tree. We tune the rest of the parameters using the methods we mentioned in Section 4.2. We measure the average precision at 20 nearest neighbors at the same percentage of scanned bins, which are called probed buckets in RankReduce terms.

We plot the comparison results in Figure 4.10. The results show that DIMO consistently outperforms RankReduce in average recall while maintaining the same average selectivity. The performance improvements are significant (15 -- 20%) especially in the practical settings when we scan 5 -- 10% of the data points. For example, when the fraction of scanned data points is 5%, the average recall achieved by DIMO is about 84%, while the average recall achieved by RankReduce is less than 65% for the same fraction of scanned data points. For RankReduce to achieve 84% average recall, it needs to scan at least 15% of the dataset (3X more than DIMO), which incurs significantly more computation and I/O overheads than DIMO. Similarly, when scanning 10% of the data points, DIMO achieves more than 97% average recall, while RankReduce achieves less than 80% average recall. We note that the performance of DIMO and RankReduce is close when we scan a large fraction of the data (both will get close to 100% average recall), but this is not relevant in practice because of the huge computational costs needed. Similarly, when scanning a tiny fraction of data

($< 1\%$), both systems produce low average recall, which may also not be useful for many practical applications. Nonetheless, in all cases, the performance of DIMO is better than that of RankReduce.

In addition to the superior performance in terms of average recall, DIMO is also more efficient in terms of storage and computation. For storage, RankReduce needs to store the whole reference dataset multiple times in hash tables; up to 32 times. On the other hand, DIMO stores the reference dataset only once in bins. Storage requirements for a dataset of size 32,000 points indicate that RankReduce needs up to 8 GB of storage, while DIMO needs up to 5 MB , which is more than 3 orders of magnitude less. These storage requirements may render RankReduce not applicable for large datasets with millions of points, while DIMO can scale well to support massive datasets.

For computation resources, DIMO and RankReduce use similar scan method to reference points found in bins or buckets. However, as discussed above, RankReduce needs to scan more buckets to produce similar precision as DIMO. This makes DIMO more computationally efficient for a certain target precision, as it scans fewer bins.

In summary, the results in this section show that DIMO outperforms the closest system in the literature (RankReduce [64]) by a large margin in terms of the achieved average recall of the computed nearest neighbors. Furthermore, DIMO requires at least 3 orders of magnitude less storage than RankReduce and it is more computationally efficient.

4.5 Scalability and Elasticity of DIMO

We conduct multiple experiments to show that DIMO is scalable and elastic. Scalability means the ability to process large volumes of data, while elasticity indicates the ability to efficiently utilize various amounts of computing resources. Both are important characteristics: scalability is needed to keep up with the continuously increasing volumes of data and elasticity is quite useful in cloud computing settings where computing resources can be acquired on demand.

We run DIMO on datasets of different sizes from 10 to 160 million data points, and on clusters of sizes ranging from 8 to 128 machines. In all experiments, we compute the $K = 10$ nearest neighbors for a query dataset of size 100,000 data points. In the rest of experiments unless stated other wise, we use the KD forest method with one tree to build with 10 levels ending in a 1,024 bins (partitions). We scan 16 out of the 1,024 bins, holding a fixed selectivity of 1.5%.

We measure the total running time to complete processing all queries, and we plot the results

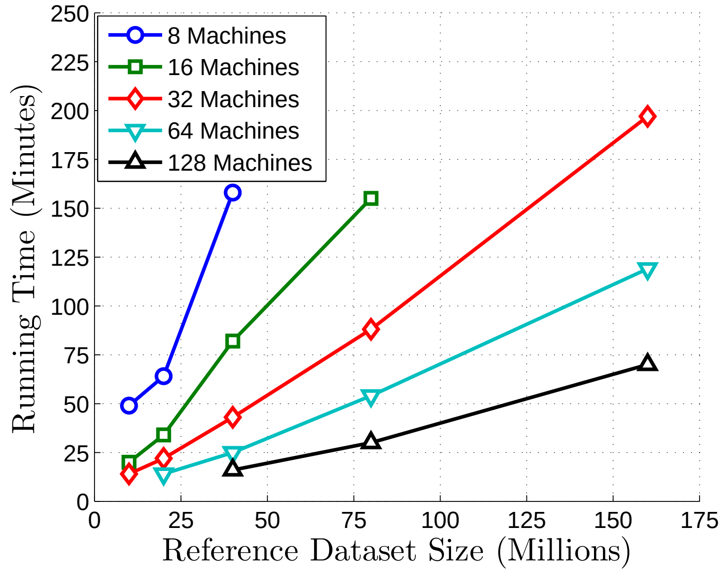
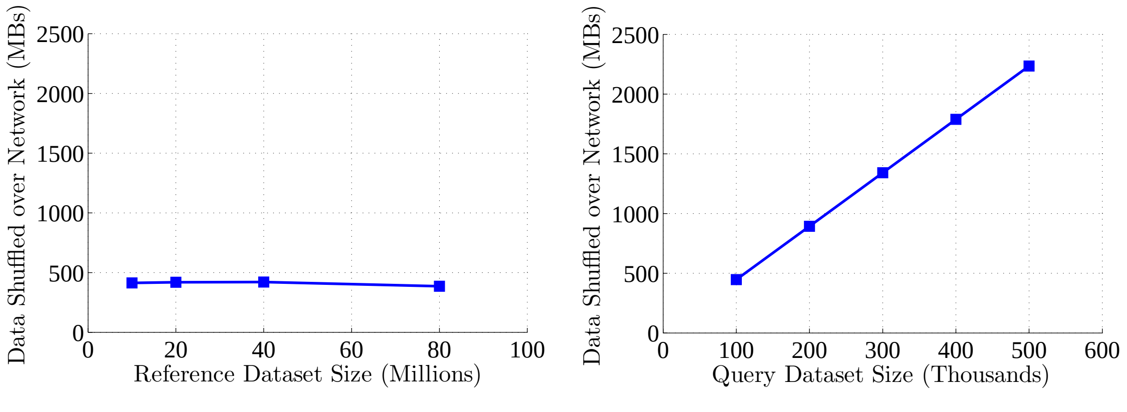


Figure 4.11: Scalability and elasticity of DIMO: Running times of different dataset sizes on different number of machines.

in Figure 4.11. The figure shows that DIMO is able to handle large datasets, up to 160 million reference data points are used in creating the distributed index. More importantly, the running time grows almost linearly with increasing the dataset size on the same number of machines. Consider for example the curve showing the running times on 32 machines. The running times for the reference dataset of sizes 40, 80, and 160 million data points are about 40, 85, and 190 minutes, respectively.

In addition, the results in Figure 4.11 clearly indicate that DIMO can efficiently utilize any available computing resources. This is shown by the almost linear reduction in the running time of processing the same dataset with more machines. For example, the running times of processing a reference dataset of size of 80 million data points are 160, 85, 52, and 27 minutes for clusters of sizes 16, 32, 64, and 128 machines, respectively.

The scalability and elasticity of DIMO are obtained mainly by our design of the distributed index, which splits the datasets into independent and non-overlapping partitions. These partitions are allocated independently to computing machines for further processing. This data splitting and allocation to partitions enable flexible and dynamic distribution of the computational workload to the available computing resources, which is supported by the MapReduce framework.



(a) Different sizes of reference datasets

(b) Different sizes of query datasets

Figure 4.12: Network overhead imposed by DIMO.

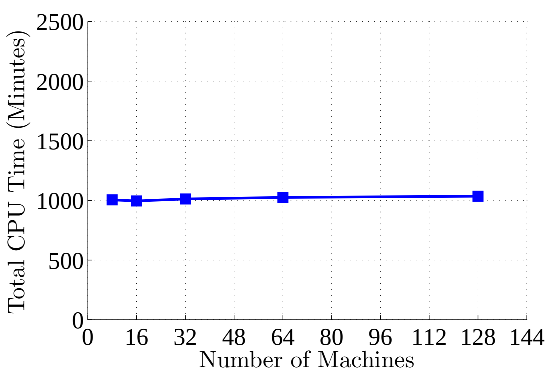
In summary, the experiments in this section show that DIMO can process large datasets and the processing time proportionally decreases as more computing resources become available to DIMO.

4.6 Overhead Analysis of DIMO

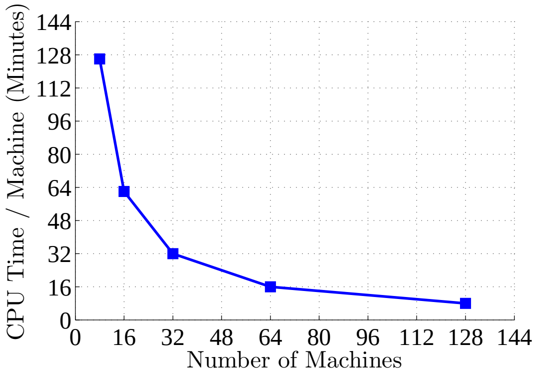
In this section, we analyze various aspects of the DIMO system.

4.6.1 Network Overhead

It is important to minimize the amount of data exchanged over the network in the DIMO system; otherwise DIMO may not be able to process large datasets or run on large clusters. We run different sets of experiments on a fixed-size cluster of 16 machines, and we measure the amount of data exchanged among computing nodes across the network. We measure this network overhead for various reference data sizes as well as for different query sizes, and we plot the results in Figure 4.12. We obtain the amount of data exchanged from Hadoop logs. In Figure 4.12(a), we vary the reference dataset size from 10 to 160 million data points, and we fix the query dataset size at 100,000 data points. The figure shows that the amount of data shuffled across the network is not affected by the increase of the reference dataset size, which indicates that DIMO effectively partitions the reference dataset to minimize the network communication; if otherwise, more data would have been shuffled across the network as the size of the reference dataset increases, which proportionally increases the



(a) Total CPU time across all machines



(b) Average CPU time per machine

Figure 4.13: Analysis of CPU usage by DIMO for different cluster sizes.

size of the distributed index.

In Figure 4.12(b), we vary the query dataset size from 100,000 to 500,000 data points, and we fix the reference dataset size at 10 million data points data points. The results show a linear increase in the total amount of data shuffled across the network as the size of the query dataset increases.

4.6.2 Storage Usage

Our usage of storage is predictable before running the system. We write the data points in binary format, and store them as fixed-length records in flat files. We limit the file size to fit in one distributed file system block (64 MB or 128 MB). This enables fast I/O operations by reading the whole bin using the minimum number of I/O operations. In addition, the distributed file system uses a lazy allocation strategy. That is, if data points inside a partition are less than the the block size, the distributed file system used only the required space without allocating the whole block size. This yields good utilization of the storage system.

In our experiments with image datasets, we use 128-dimension SIFT features as data points. Each data point is stored in 136 bytes: 8 bytes for object ID and point ID, and 128 bytes for the 128 dimensions. Thus, for example, for a 1 million data point, we need 136 MB of storage on the distributed file system for each copy (We store a copy for the dataset for each tree or hash table). We confirmed these calculations by inspecting the Hadoop logs, which showed close numbers. The storage overhead in this case is 8 bytes (for storing IDs for each point) divided by 136 bytes (total size needed for each point), which is less than 6%.

In addition, we store the upper part of the distributed index (the space partitioner) only once on the distributed file system. The size of the space partitioner depends on the number of data points in the reference dataset and build parameters. However, as mentioned in Section 3.2, the space partitioner does not store data points; it only stores meta data. This meta data is a scalar value, which means that each node in the tree or a hash bucket in a hash table needs up to 4 bytes. Therefore, the space partitioner takes a negligible space on the storage system compared to the reference dataset.

4.6.3 Memory Usage

The DIMO system does not require large memory. For our experiments, we set the maximum allowed memory to 512 MB per node for reducer tasks, and 256 MB per node for mapper tasks. Thus, DIMO can run on regular off-the-shelf servers, even for processing very large datasets.

4.6.4 CPU Usage and Load Distribution

Two important aspects of any distributed system are how it can balance the workload among the computing machines, and whether adding more machines introduces more overheads. Load imbalance results in inefficient utilization of resources and increased running times. More overheads ultimately limit the scalability of the system. To analyze the performance of DIMO along these two important issues, we fix the total workload and increase the cluster size from 8 to 128 machines. The workload is composed of processing 100,000 query data points against 40 million reference data points.

For each cluster size, we run the experiment and measure the total CPU time, which is the summation of CPU times on all machines in the cluster. We also measure the average CPU time per machine in each case. The results are shown in Figure 4.13. We obtain these measurements from Hadoop logs. The results in Figure 4.13(a) show that DIMO does not introduce any significant overheads when the cluster size increases, because the total CPU time remains around 1,000 minutes while the number of machines varies from 8 to 128. In addition, Figure 4.13(b) shows that the workload (the 1,000 total CPU time) is equally distributed across all machines. For example, increasing the number of machines from 16 to 32 results in commensurate reduction in the CPU time per machine from 64 to 32 minutes.

In summary, the results in this section indicate that DIMO: (i) does not impose high network overhead, (ii) uses the storage system efficiently, (iii) does not require large main memory even for processing large datasets, and (iv) balances the load across the used computing machines.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we presented a new method to store and index large-scale high-dimensional data points for fast searching and matching. Unlike systems proposed in previous works, our index is general and can be used by multiple applications that require nearest neighbors search in high-dimensional spaces. Our method computes approximate nearest neighbors, where the accuracy of the computed neighbors can be to traded off with the required computing resources. This feature makes our method useful for diverse multimedia applications that have different accuracy requirements and run on computing platforms with various capacities. We implemented our method in a complete system called DIMO using the MapReduce programming model. We presented adapted designs and implementations of two state-of-the-art space-partitioning techniques, which are K-dimensional forest (KD forest) and multi-probe locality-sensitive hashing (MP-LSH). We experimented with our system on clusters of different sizes from the Amazon Elastic MapReduce (EMR) cloud service. We extracted more than 160 million data points from more than 1 million images from the public ImageNet dataset. The data points are SIFT features, where each one has 128 dimensions. We rigorously assessed the performance of the DIMO system and compared it against the closest one in the literature, which is called RankReduce. Our results showed that our system achieves high accuracy of up to 95% compared to the ground-truth nearest neighbors. Our results also showed that DIMO is scalable and elastic in the sense that it can efficiently utilize varying amounts of computing resources, which is a desirable feature given the wide adoption of the on-demand cloud computing model for acquiring computing resources. In addition, our comparison results showed that DIMO outperforms RankReduce in terms of the achieved precision of the computed nearest neighbors, uses less storage, and it is more computationally efficient.

5.2 Future Work

The work in this thesis can be extended in multiple directions. For example, in Section 3.3 we introduced the design of KD forest as a space partitioner. Other types for trees [54] can be adapted as well in our design of DIMO, after we perform our ideas of keeping data points only at leaves, aggregating data points into bins, and storing bins on the distributed file system. Also we note that one of the drawbacks of KD trees and hierarchical space division techniques in general is the performance degradation for very high-dimensional data (1,000s or 1,0000s). As the classical method for dimension reduction and analysis fail to operate on such high dimensions. Nonlinear dimension reduction like kernel PCA [56], and machine learning-based methods like deep belief networks (DBNs) [35] can be used for more effective dimensionality reduction.

A possible way of improving the hashing-based space partitioner is to learn the hash codes from the data instead of the random projections methods. Many techniques are introduced for supervised and semi-supervised learning of hash codes [70]. As an example, we can use the binary reconstructive embeddings (BRE) introduced by Kulis and Darrell [39], where the algorithm learns from the data the hash codes that minimize the reconstruction error. The algorithm is trained from a sample labeled data of similar and dissimilar objects. It then positions the projection hyperplanes to minimize the distance between the similar objects and to maximize the distance between the dissimilar objects in the feature space. Such techniques can easily be adapted and plugged into our DIMO system.

A different direction of extending the system is to use memory-based frameworks for distributed processing like Spark [63] instead of disk-based frameworks like Hadoop [31]. Spark uses in-memory data structures for low-latency and real-time processing.

Bibliography

[1] Ahmed Abdelsadek and Mohamed Hefeeda. Dimo: Distributed index for matching multimedia objects using mapreduce. In *Proceedings of ACM Multimedia Systems Conference (MMSys)*, pages 115–126, 2014.

[2] Mohamed Aly, Mario Munich, and Pietro Perona. Distributed KD-Trees for Retrieval from Very Large Image Collections. In *Proceedings of British Machine Vision Conference (BMVC)*, 2011.

[3] Mohamed Aly, Mario Munich, and Pietro Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *Proceedings of IEEE Workshop on Applications of Computer Vision (WACV)*, pages 418–425, 2011.

[4] Mohamed Aly, Peter Welinder, Mario Munich, and Pietro Perona. Scaling object recognition: Benchmark of current state of the art techniques. In *Proceedings of IEEE Workshop on Emergent Issues in Large Amounts of Visual Data (WS-LAVD)*, pages 2117–2124, 2009.

[5] Amazon web services. `http://aws.amazon.com/`.

[6] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 459–468, 2006.

[7] Sunil Arya and David Mount. Algorithms for fast vector quantization. In *Proceedings of Data Compression Conference (DCC)*, pages 381–390, 1993.

[8] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[9] Bahman Bahmani, Ashish Goel, and Rajendra Shinde. Efficient distributed locality sensitive hashing. In *Proceedings of ACM International Conference on Information and Knowledge Management (CIKM)*, pages 2174–2178, 2012.

[10] Li Baoli, Lu Qin, and Yu Shiwen. An adaptive k-nearest neighbor text categorization strategy. *ACM Transactions on Asian Language Information Processing (TALIP)*, 3(4):215–226, 2004.

[11] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer Vision and Image Understanding (CVIU)*, 110(3):346–359, 2008.

[12] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1000–1006, 1997.

[13] Jon Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[14] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 217–235, 1999.

[15] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 97–104, 2006.

[16] Christian Bohm and Florian Krebs. High performance data mining using the nearest neighbor join. In *Proceedings of IEEE International Conference on Data Mining (ICDM)*, pages 43–50, 2002.

[17] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[18] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of Conference on Very Large Data Bases (VLDB)*, pages 426–435, 1997.

[19] K. Dana, B. Ginneken, S. Nayar, and J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics (TOG)*, 18(1):1–34, January 1999.

[20] Abhinandan Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: Scalable online collaborative filtering. In *Proceedings of International Conference on World Wide Web (WWW)*, pages 271–280, 2007.

[21] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of Symposium on Computational Geometry (SCG)*, pages 253–262, 2004.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Opearting Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[23] Sampath Deegalla and Henrik Bostrom. Reducing high-dimensional data by principal component analysis vs. random projection for nearest neighbor classification. In *Proceedings of the*

International Conference on Machine Learning and Applications (ICMLA), pages 245–250, 2006.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[25] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling lsh for performance tuning. In *Proceedings of ACM Conference on Information and Knowledge Management (CIKM)*, pages 669–678, 2008.

[26] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. A content-addressable network for similarity search in metric spaces. In *Proceedings of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, pages 126–137, 2005.

[27] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[28] A. Fu, P. Chan, Y. Cheung, and Y. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, 9(2):154–173, July 2000.

[29] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 518–529, 1999.

[30] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[31] Apache hadoop. `http://hadoop.apache.org`.

[32] Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology (EDBT)*, pages 744–755, 2009.

[33] Hannaneh Hajishirzi, Wentau Yih, and Aleksander Kolcz. Adaptive near-duplicate detection via similarity learning. In *Proceedings of ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 419–426, 2010.

[34] Alexander Hinneburg, Charu C. Aggarwal, and Daniel A. Keim. What is the nearest neighbor in high dimensional spaces? In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 506–515, 2000.

[35] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[36] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

[37] You Jia, Jingdong Wang, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3392–3399, 2010.

[38] Naghmeh Khodabakhshi and Mohamed Hefeeda. Spider: A system for finding 3d video copies. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 9(1):1–20, 2013.

[39] Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *Proceedings of Advances in Neural Information Processing Systems (NIPS'09)*, pages 1042–1050, 2009.

[40] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pages 2130–2137, 2009.

[41] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Back-propagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, December 1989.

[42] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[43] Fei-fei Li. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. In *In Proceedings of IEEE CVPR Workshop of Generative Model Based Vision (WGMBV)*, 2004.

[44] Haojun Liao, Jizhong Han, and Jinyun Fang. Multi-dimensional index on hadoop distributed file system. In *Proceedings of IEEE International Conference on Networking, Architecture, and Storage (NAS)*, pages 240–249, 2010.

[45] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision (IJCV)*, 60(2):91–110, 2004.

[46] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment (PVLDB)*, 5(10):1016–1027, 2012.

[47] 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 International Conference on Very Large Data Bases (VLDB)*, pages 950–961, 2007.

[48] Brian McFee and Gert RG Lanckriet. Large-scale music similarity search with spatial trees. In *Proceedings of Symposium for Music Information Retrieval (ISMIR)*, pages 55–60, 2011.

[49] James McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 23(9):964–976, 2001.

[50] Marius Muja and David Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proceedings of the International Conference on Computer Vision Theory and Application (VISSAPP)*, pages 331–340, 2009.

[51] Marius Muja and David Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 36, 2014.

[52] David Nistr and Henrik Stewnius. Scalable recognition with a vocabulary tree. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2161–2168, 2006.

[53] M. Patella and P. Ciaccia. Approximate similarity search: A multi-faceted problem. *Journal of Discrete Algorithms*, 7(1):36–48, 2009.

[54] Parikshit Ram and Alexander Gray. Which space partitioning tree to use for search? In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 656–664, 2013.

[55] Frederik Schaffalitzky and Andrew Zisserman. Automated scene matching in movies. In *Proceedings of the International Conference on Image and Video Retrieval (CIVR)*, pages 186–197, 2002.

[56] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Journal of Neural Computation*, 10(5):1299–1319, 1998.

[57] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[58] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, 2008.

[59] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pages 1470–1477, 2003.

[60] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine*, 25(2):128–131, 2008.

[61] Malcolm Slaney, Yury Lifshits, and Junfeng He. Optimal parameters for locality-sensitive hashing. *Proceedings of the IEEE*, 100(9):2604–2623, 2012.

[62] S. J. Smith, M. O. Bourgoin, K. Sims, and H. L. Voorhees. Handwritten character classification using nearest neighbor in large databases. *IEEE Transactions on Pattern Analysis and Machine Intelligence(PAMI)*, 16(9):915–919, 1994.

[63] Apache spark. `http://spark.apache.org`.

[64] Aleksandar Stupar, Sebastian Michel, and Ralf Schenkel. Rankreduce - processing k-nearest neighbor queries on top of mapreduce. In *Proceedings of the International Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, pages 13–18, 2010.

[65] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment (PVLDB)*, 6(14):1930–1941, 2013.

[66] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 563–576, 2009.

[67] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[68] Andrea Vedaldi and Brian Fulkerson. Vlfeat: An open and portable library of computer vision algorithms. In *Proceedings of the ACM International Conference on Multimedia (MM)*, pages 1469–1472, 2010.

[69] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 591–602, 2010.

[70] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. Semi-supervised hashing for large-scale search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 34(12):2393–2406, 2012.

[71] 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 International Conference on Very Large Data Bases (VLDB)*, pages 194–205, 1998.

[72] Peter Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321, 1993.

[73] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 38–49, 2012.

[74] Hao Zhang, Alexander C Berg, Michael Maire, and Jitendra Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2126–2136, 2006.