DIMO: Distributed Index for Matching Multimedia Objects using MapReduce

Ahmed Abdelsadek
School of Computing Science
Simon Fraser University
Surrey, BC, Canada

Mohamed Hefeeda
Qatar Computing Research Institute
Qatar Foundation
Doha, Qatar

ABSTRACT

This paper presents the design and evaluation of DIMO, a distributed system for matching high-dimensional multimedia objects. DIMO provides multimedia applications with the basic function of computing the K 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 a novel method for partitioning, searching, and storing high-dimensional datasets on distributed infrastructures that support the MapReduce programming model. We have implemented DIMO and extensively evaluated it on Amazon clusters with number of machines ranging from 8 to 128. We have experimented with large datasets of sizes up to 160 million data points extracted from images, and each point has 128 dimensions. Our experimental results show that DIMO: (i) results in high precision 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. In addition, DIMO outperforms the closest system in the literature by a large margin (up to 20%) in terms of the achieved average precision of the computed nearest neighbors. Furthermore, DIMO requires at least three orders of magnitudes less storage than the other system, and it is more computationally efficient.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; H.2.4 [**Database Management**]: Systems—*Multimedia databases*

General Terms

Design

Keywords

Object matching, nearest neighbors, multimedia search, high dimensional data, large-scale data

MMSys '14, March 19–21 2014, Singapore

http://dx.doi.org/10.1145/2557642.2557650.

1. INTRODUCTION

The rapidly increasing volume of multimedia content over the Internet creates many research challenges for efficiently storing, processing, and searching such sheer volume. In this paper, we address one of these challenges: matching multimedia objects, which is the problem of finding similar objects to a given multimedia object. We address this problem for large-scale multimedia datasets that are characterized by a large number of high-dimension points. Object matching is an important problem with numerous real-life applications, such as image retrieval, document classification, duplicate removal, video copy detection, among many others. To solve this problem, a method for finding the nearest neighbors of a given data point is needed. This is known as the K-nearest neighbors problem, where $K \geq 1$. After obtaining the nearest neighbors for a given point or group of points, other processing steps may need to be applied for obtaining the final object matching results. These steps depend on the specific application that uses the object matching results. For example, in a video copy detection application [16], individual frames from a suspected query video are first matched against frames from reference videos. Then, the temporal aspects of the matched frames are considered in order to decide whether the suspected video is a copy of one of the reference videos.

We propose a distributed system for matching multimedia objects, which we call DIMO. DIMO provides the primitive function of finding K-nearest neighbors (KNN). It also supports application-specific functions to be applied on the computed nearest neighbors. There are several *centralized* approaches for solving the K-nearest neighbors problem in the literature, including ones that produce exact neighbors [30, 8], and others that compute approximate neighbors [7, 3, 10]. However, these centralized approaches do not scale to the currently-available massive volumes of multimedia objects. Distributed approaches have also been proposed, such as [31, 20, 1]. However, most of these approaches either do not support high (100+) dimensions [31, 20] or they are customized for a specific application [1]. In contrast, the proposed DIMO system offers the following desirable properties:

- **General.** DIMO can be used for different object matching applications. It produces approximate K neighbors, where the accuracy of the neighbors can be traded off with the computational complexity. 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 clusters of different numbers of machines and on large-scale datasets. We used publically-available, large image data sets [11, 2] in our experiments. Our results show that DIMO: (i) yields high precision, (ii) scales almost linearly with data size and number of machines, (iii) outperforms the closest system in the literature, and (iv) does not impose significant storage or network overheads on the distributed cluster. Furthermore, DIMO can run on tens to thousands of machines, because it has no hot-spot central machine or single point of failure.

The rest of this paper is organized as follows. Section 2 surveys the related works in the literature. Section 3 presents the details of the proposed system, and Section 4 presents our evaluation setup and results. Section 5 concludes the paper.

2. RELATED WORK

Nearest neighbors search has hundreds of applications in many fields in computer science. Our focus in this paper is on the challenging high dimensional multimedia datasets. Khodabakhshi and Hefeeda [16] develop a system for copy detection of 3D videos, where they depend on a K nearest neighbors algorithm in the underlying index. Smith et al. [25] use nearest neighbors algorithm in character classification. Zhang et al. [32] combine a K nearest neighbors algorithm with support vector machine (SVM) for different tasks of visual category recognition. They experiment with tasks like handwritten character recognition, texture analysis, and object categorization. Kulis et al. [17] integrate locality-sensitive hashing (LSH) with a kernel machine algorithm for scalable image search and object recognition tasks. Our proposed DIMO system can support such applications.

Techniques for solving the K-nearest neighbors problem can be divided into two main categories: (i) hierarchical space division and (ii) space mapping. In the first category, algorithms are used to hierarchically divide the search space into tree-based structures. Then, branch and bound methods are used to search and manipulate these structures. These techniques can be used either in the Euclidean space, e.g., R tree [14] and KD tree [5], or in the general metric space, e.g., VP tree [30] and M tree [8]. These techniques are mainly designed to return exact neighbors, and are suitable only for low dimensional datasets.

For high-dimensional datasets, exact neighbors are costly to find, because most nearest neighbors search methods will do no better than linear scan of the whole dataset [6]. 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, either by changing the distance used to compare objects or by modifying the dimensions of the object space. Then, they solve the problem on the new (approximate) space, where the search is simpler. Examples of such techniques are those that approximate vectors using a fixed number of bits such as vector approximation (VA-Files) [7]; ones that are based on locality-sensitive hashing (LSH) [13, 3]; or those based on dimensionality reduction using projections [10, 21].

The above 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 [12, 15, 29]. In [12], Flachi et al. introduce M-CAN, which is based on the Content-addressable Network (CAN) P2P architecture [23]. 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. [15] use LSH on top of the Chord P2P architecture [26]. In [29], 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, 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 [12, 15, 29] did not focus on large-scale multimedia datasets that are characterized by high dimensions. These works either used low dimensional data, e.g., five dimensions in [29], or simulation on a single machine in [15].

Another class of works for solving the nearest neighbors problem in a distributed manner relies on distributed processing frameworks such as MapReduce. In [18], Liao et al. build a multi-dimensional index using R-tree on top of the Hadoop distributed file system (HDFS) [24]. 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 [20] and [31] solve the K nearest neighbors over large datasets using MapReduce [9]. Lu et al. [20] 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. [31] split both query and reference datasets into a number of disjoint equal-sized subsets, where operations can be done in parallel. The systems in [20, 31] 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 [27], 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. [1] 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.

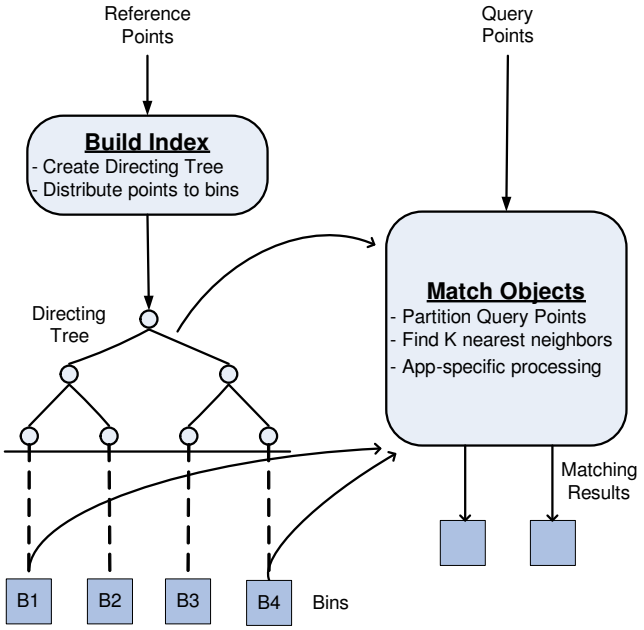


Figure 1: High-level architecture of DIMO . Round boxes are MapReduce jobs. The top part of the index, directing tree, is serialized and used by multiple machines, while bin at the lower part contain data points and are stored on the distributed file system.

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 [1] is tailored to image search, while our system is more general. For example, the authors of [1] 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.

3. PROPOSED SYSTEM

In this section, we start by presenting an overview of the proposed system, which is followed by the details of its main components in separate subsections.

3.1 Overview

DIMO is a general system for matching multimedia objects that are characterized by many features and each feature is of high dimensions. For example, an image can be characterized by 100–200 SIFT descriptors, and each has up to 128 dimensions, and a video object will have even more features extracted from its frames. To achieve this general matching task, DIMO first provides an efficient, distributed, implementation for computing K nearest neighbors for high-dimensional data. 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 a distributed index over R and it then uses it to 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 one 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 partitions the reference points R into bins. These bins are mapped to files and stored on a distributed file system. The bins are searched in parallel against query objects. The core component in our system that enables efficient partitioning, mapping, and searching for objects is the Distributed Index. Figure 1 shows the high-level architecture of the DIMO system. There are two main computational tasks: Build Index and Match Objects. Build Index takes the reference points and creates the Distributed Index. Match Objects computes the nearest neighbors for each query point as well as it performs application-specific object matching functions using the found nearest neighbors.

DIMO is implemented using the MapReduce distributed programming model [9]. The two computational tasks mentioned above are MapReduce jobs that run on multiple machines. 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 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.

The proposed design of the DIMO system achieves the desired properties mentioned in Section 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 3.4, we discuss how DIMO can scale to very large datasets and how it can dynamically add/remove data points to the reference set.

3.2 Building the Distributed Index

We design a scalable, elastic, and distributed index for the high-dimensional object matching problem. As shown in the left part of Figure 1, the index is divided into two parts: (i) *directing tree* and (ii) *bins* . Directing tree is a space partitioning tree [22] that is used to group similar points in the same or close-by bins. It is also used to forward query points to the bins with potential matches. Bins are the leaf nodes of the directing tree, but they are stored as files 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 leaf nodes. Intermediate nodes do not store any data, they only store meta data to guide the search through the tree. This significantly reduces the size of the directing tree 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 directing tree to computing machines to process queries in parallel. Replicating the directing tree 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 failures. The efficiency is improved because there is no central machine or set of machines that other machines need to contact during the computation. The second feature of our index design is the separation of the leaf nodes (bins) 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 al leaves makes updating the index easier as explained in Section 3.4.

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 directing tree and (ii) distributing the reference dataset to bins. The directing tree is created using a sample from the reference dataset and it is done on one machine; details are given in Section 3.2.1. Once created, the directing tree 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 3.2.2.

3.2.1 Constructing the Directing Tree

The directing tree is the top part of the index, which contains all non-leaf nodes. Different types of trees [22] can be used as our directing 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. We chose the KD tree [5] as the base for our directing tree, because of its efficiency and simplicity. A KD tree is binary tree in which every node is a K-dimensional point. Every non-leaf node can be considered as a splitting hyperplane that divides the space into two parts. 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. The hyperplane direction is chosen in a way such that every node in the tree is associated with one of the K dimensions, with the hyperplane perpendicular to that dimension's axis, and it splits the data points around it into two equal-size subsets. The equal-size subsets make the tree balanced.

Figure 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.

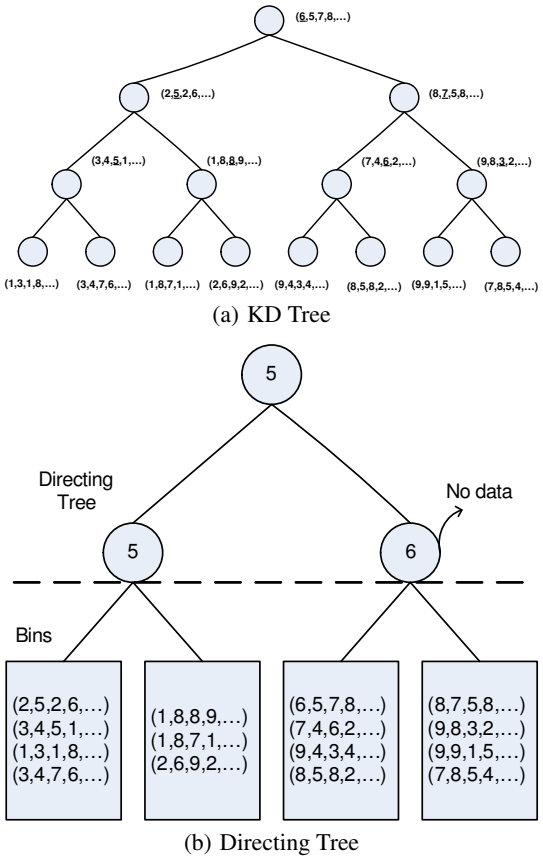


Figure 2: Classical KD Tree and Directing Tree.

We are interested in matching objects with high dimensions. Thus, if we use the traditional KD tree, it will be too deep with too many leaf 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 2(b) shows our directing tree for the same 15 data points used in Figure 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 directing 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).

Since we compress the depth of the tree, we use only a subset of the dimensions of the data points. Multiple methods can be used to choose this subset of dimensions. For example, we may use the dimensions that have the highest variance in the dataset. In our implementation, we use the principal component analysis (PCA) to choose the most representative dimensions to project the dataset on. PCA is a well studied technique for dimension reduction. It finds a hyperplane of the required target dimensionality to project the actual points on, such that the variance among them after projection is maximized. It finds this hyperplane by calculating the singular value decomposition (SVD) of the covariance matrix of the input points.

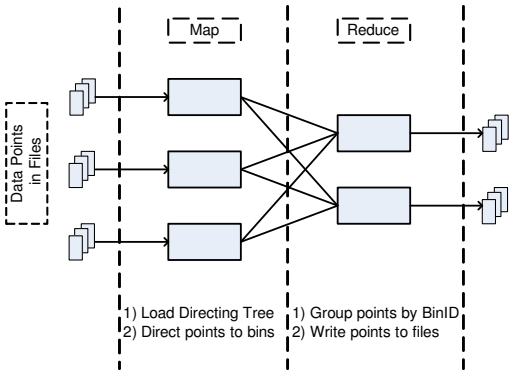


Figure 3: Illustration of the Distribute Data MapReduce job.

In order to construct the directing tree, we use a random sample of the reference dataset, if the reference dataset is large. Note that the directing tree is used in distributing all data points to bins as explained in Section 3.2.2, but the construction of the tree is done from a sample of the reference data points.

The following steps are performed to construct the directing tree.

- Decide on the number of levels $numLevels$ in the directing tree. This is calculated from the size of the reference dataset $refSize$ and the storage block size $bockSize$ 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: $numLevels \geq \lceil \log_2 (refSize/blockSize) \rceil$.

- Decide on the number of dimensions $numDim$ of the data points that will be used. Recall that we partition the dataset based on a different dimension in each level of the tree.

- Compute the splitting value for each level in the tree. First, we apply PCA on a random sample of the reference dataset to estimate $numDim$ principal components.

- Construct the directing 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.

Procedure 1 Distribute Data MapReduce Job

MAPPER

1: **function** SETUP ▷ Loaded once per machine
2: DirectingTree dt = LoadDirectingTree ()
3: **end function**

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

1: **function** MAP(FileID i, File f)
2: **for each** Point p in File f **do**
3: $Bins$ = GetClosestBins (dt, p, 1)
4: **for each** Bin in $Bins$ **do**
5: Emit (Bin.getID(), p)
6: **end for**
7: **end for**
8: **end function**

REDUCER

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

1: **function** REDUCE([(BinID bid, Point p)])
2: Write a file for each bin and and store all points that correspond to that bin in it
3: **end function**

3.2.2 Distributing Data Points

Distribution of data points to bins is achieved in parallel using the Distribute Data MapReduce job, which is illustrated pictorially in Figure 3 and described in pseudo code in Procedure 1.

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 directing tree from the distributed file system. Then, for each data point, the mapper traverses the directing tree to find the closest bin to this point using GetClosestBins() function. After finding the closest bin for each point, the mapper emits key-value pairs in the form of (BinID, point). Pairs having the same BinID are sent to the same reducer by the MapReduce infrastructure. The reducer, in turn, groups all points with the same BinID and writes them to a file on the distributed file system, which is identified by the BinID.

The GetClosetBins() function returns the requested number of closest bins to a given point. Procedure 2 shows the pseudo code of this functions, which is a variant of the best bin first algorithm in [4]. 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 directing tree. This is done by maintaining a priority queue, based on the distance between the query point and nodes in the tree. The queue is initialized by inserting the root of the directing tree in it. Then, while the algorithm traverses down the directing 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 directing 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 directing tree again starting from the head of the priority queue. Otherwise, the algorithm returns the list of closest bins.

3.3 Matching 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

Procedure 2 Finding Closest Bins to Given Data Point

```
1: function GETCLOSETBINS(DirectingTree dt, Point p, int count)
2:     List closestBins = []
3:     PriorityQueue Q = root of dt
4:     DirectingTreeNode 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
```

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 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 partitions the query dataset such that each partition contains a bin and a list of data points that are likely to have neighbors in that bin. 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 and Procedure 1), except for two modifications. The first modification is that the reducer in the Partition Queries task does not store bins on the distributed file system. Rather, it emits bin IDs and lists of query data points; one list for each bin ID. The directing tree is used to create the list of data points that corresponds to each bin, in the same way as described in Section 3.2.2. The second modification is the setting of the number of bins parameter in GetClosetBins() function in line 4 of Procedure 1. While the number of bins is set to one in the Distribute Data task, since each reference data point is stored in only one bin, the number of bins is variable and used to control the accuracy of the computed K nearest neighbors in the Partition Queries task. If the number of bins is set to n, then each query data point is compared against all reference points in n bins, 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 4. The figure shows that these two steps are achieved through one MapReduce job that has one mapper and

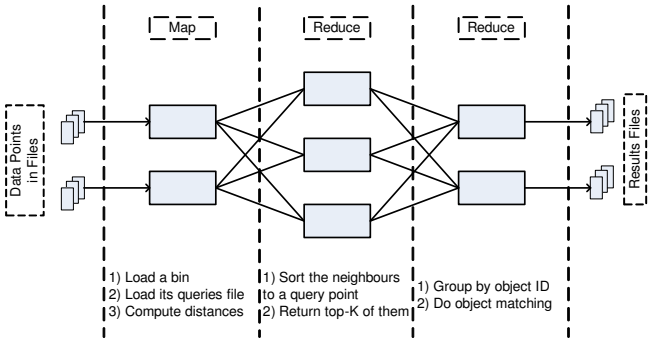


Figure 4: Illustration of the second and third steps of the object matching process in the DIMO system.

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 [16], individual matching of query frames with 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: $(BinID, [p1, p2, \dots])$. It then loads the file corresponding to $BinID$ 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 bins 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 bins, 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 features) 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.

3.4 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 leaf nodes, makes updating the index less complex than other space partitioning data structures that store data in interior nodes. When a data point is added or removed, we only change leaf nodes, and we do not need to manipulate or adjust the internal structure of the tree, which can involve significant computation and commu-

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

Procedure 3 Object Matching MapReduce Job

MAPPER

//Input: BinID and list of query points

//Output: List of $[(pointID, neighbor)]$, where $neighbor = (pointID, distance)$

```
1: function MAP(BinID bid, Point qList[])
2:     Point rList[] = Load points from file corresponding to bid
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 ≤ 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], ..., 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
```

nication overheads in distributed environments. In addition, since leaf nodes are stored as separate files on the distributed file system, accessing and updating them can easily be done in parallel, as concurrent 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.

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

Scaling Index Up/Down. When the number of points in the index increases/decreases significantly, the internal tree structure of the index needs to be scaled up or down to handle the change of the data size. We scale the index by controlling the number of levels $numLevels$ in the directing 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 the tree and hence doubling the total number of points that can be managed by the index. To achieve this, 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 we have used all dimensions already in the tree, we recycle through them again. That is, the dimension for the new level of the tree will be the same as the first level of the tree. Then, as we do for the initial index construction, we project and sort the data points (of the random sample) 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 used 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.

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 probabilistic 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 bins become quite large while others are empty or have few points. This means that large bins 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 bins, 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 bins on the distributed file system. If the bin size distribution significantly deviates from the expected uniform distribution, the index should be re-built from scratch, by repeating the steps in Section 3.2. The deviation from the uniform distribution is quantified by measuring the variance in the bin sizes and if it exceeds a pre-defined threshold, the index re-building process is initiated.

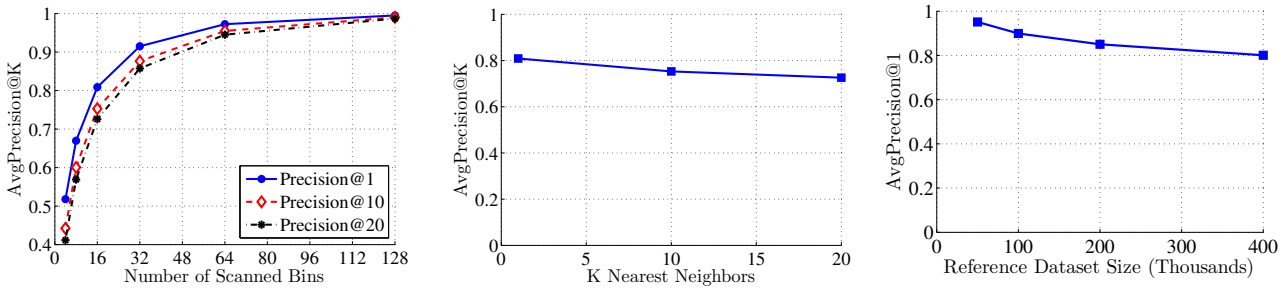
4. EVALUATION

In this section, we evaluate the performance of the proposed DIMO system and 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 [27]. Then, we analyze the elasticity and scalability of DIMO. Finally, we analyze various overhead imposed by DIMO.

4.1 Experimental Setup and Datasets

Platform. We have implemented the DIMO system using Java 1.7 and Apache Hadoop 1.0.3. 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 op-



(a) Varying number of bins. (b) Varying number of nearest neighbors. (c) Varying data size.

Figure 5: Comparing DIMO versus ground truth.

erating system is 64-bit Ubuntu linux 12.04. We used this virtual Hadoop cluster for small-scale experiments.

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 and because they are more available than other data types. We extract SIFT [19] features from images and use each SIFT feature as a data point. We use the VLFeat 0.9.17 [28] 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. This dataset is composed of data points extracted from 2,500 images. The images are obtained from the Caltech Buildings and Game Covers datasets [2]. 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 200 data points from each image. Thus, this dataset contains 500,000 reference data points. We use all points in the construction of the directing tree. The query set contains 10,000 data points randomly selected from the 500,000 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 [11]. 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. To build the directing tree, we randomly select 1 million points out of the 200 millions. The query set contains 100,000 data points randomly selected from the 200 million points.

Performance Metrics. The main goal of the DIMO system is to provide accurate nearest neighbors. The accuracy of the retrieved K nearest neighbors for a point p is assessed using the $Precision@K(p)$ metric, which is given by:

$$Precision@K(p) = \frac{\sum_{i=1}^{K}\{T_i <= K\}}{K}, \quad (1)$$

where T_i is the rank of a true neighbor. $T_i <= K$ equals 1 if a true neighbor is within the retrieved K, and 0 otherwise.

The average precision of the retrieved K nearest neighbors across all points in the query set Q points is:

$$AvgPrecision@K = \frac{\sum_{i=1}^{|Q|}\{Precision@K(i)\}}{|Q|}. \quad (2)$$

We use the $AvgPrecision@K$ metric with different values for K in our experiments.

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

4.2 Comparison Against Ground Truth

We compare the accuracy of the returned nearest neighbors by DIMO against the true nearest neighbors computed by a brute force approach. The true nearest neighbors are computed 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 this experiment, which has 500,000 reference points and 10,000 query points.

We plot the $AvgPrecision@K$ in Figure 5(a) for $K = 1, 10$ and 20. On the x-axis, we vary the number of scanned bins when searching for nearest neighbors, as explained in Section 3.3. We scan different number of bins at values of 4, 8, 16, 32, 64 and 128 bins out of 1024 total bins. The scanned bins represent 0.39%, 0.78%, 1.5%, 3.125%, 6.25%, and 12.5% of the data size, respectively. We measure the average precision of 10,000 query data points against the 500,000 reference data points. The results show that DIMO achieves high precision. For example, when we scan 16 bins out of 1,024 representing only 1.5% of the data size, the precision is more than 80% for $K = 1$ and it is more than 70% for $K = 10$, and 20. In addition, when we set the number of scanned bins to 64, DIMO achieves an average precision of more than 93% for $K = 1, 10$, and 20. This means that, on average, 93% 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 precision achieved and the computing resources needed, which makes DIMO suitable for various applications with different requirements.

In the next experiment, we study the effect of changing the number of the retrieved K nearest neighbors. We measure the average precision at different values of K. We fix the number of scanned bins at 16 representing only 1.5% of the data size. The results

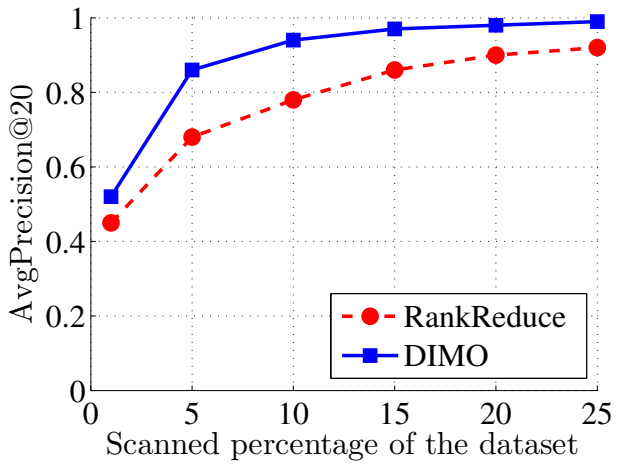


Figure 6: Comparing DIMO versus the closest system in the literature, RankReduce.

are plotted in Figure 5(b), which show that the average precision achieved by DIMO is not significantly impacted by increasing K.

Next, we show the effect of changing the reference dataset size, while keeping the number of scanned bins fixed for each query data point. We take the average precision of 10,000 query data points against different number of reference data points of 50,000, 100,000, 200,000, and 400,000. For each reference dataset, we build a directing tree of an increasing size. We start by a 10-level tree for the 50,000 points having 1,024 leaf nodes (bins). We increase the number of levels in the directing tree as the reference dataset increases, until we reach 13 levels in the tree for the reference dataset of size 400,000 points. In this case, there are 8,192 leaf nodes. We calculate average precision at $K = 1$ (the nearest neighbor) and plot the results in Figure 5(c). At query time, we always scan the same number of bins, which is 64. Hence, the scanned bins represent different percentages of the reference datasets, which are 6.25%, 3.125%, 1.5%, and 0.78% for the 50,000, 100,000, 200,000, and 400,000 datasets, respectively. Note that, since we double the total number of bins when we double the dataset size, each bin will roughly have the same number of points. Figure 5(c) shows that DIMO achieves high average precision of at least 80%, even when we scan a small fraction of less than 1% of the reference dataset. The figure also shows that the average precision is decreased by less than 15% after increasing the reference dataset by 8 folds, while scanning the same number of bins (64). Note that, scanning fixed number of bins, regardless of the data size, yields the same running time. That is, keeping the same response time even when the load increases 8 times.

The experiments in this section show that the DIMO system results in high average precision when compared against the ground truth nearest neighbors. And this high precision is maintained in different settings of increasing the number of nearest neighbors K, number of scanned bins, and size of the reference dataset.

4.3 Comparison Against RankReduce

We compare the proposed DIMO system against the closest one in the literature, which is RankReduce [27]. 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 [27] 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. 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 6. The results show that DIMO consistently outperforms RankReduce. And 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 precision achieved by DIMO is about 84%, while the average precision achieved by RankReduce is less than 65% for the same fraction of scanned data points. For RankReduce to achieve 84% average precision, 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 precision, while RankReduce achieves less than 80% average precision. 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 precision), 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 precision, 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 precision, 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 (RabkReduce [27]) by a large margin in terms of the achieved average precision of the computed nearest neighbors. Furthermore, DIMO requires at least 3 orders of magnitude less storage than RankReduce and it is more computationally efficient.

4.4 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

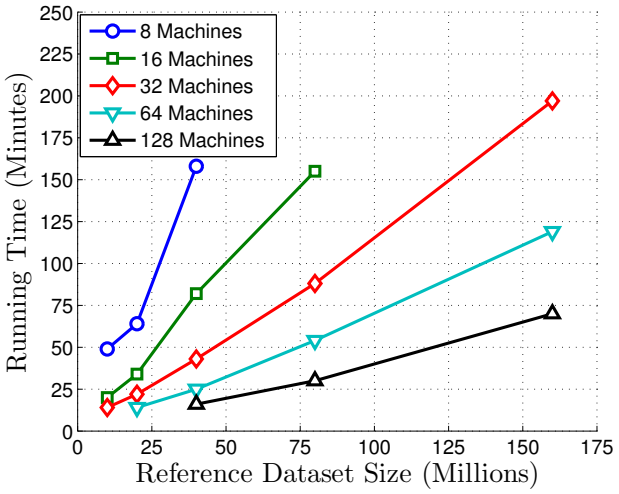


Figure 7: Scalability and elasticity of DIMO: Running times of different dataset sizes on different number of machines.

machines. In all experiments, we compute the $K = 10$ nearest neighbors for a query dataset of size 100,000 data points. We measure the total running time to complete processing all queries, and we plot the results in Figure 7. 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 7 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.

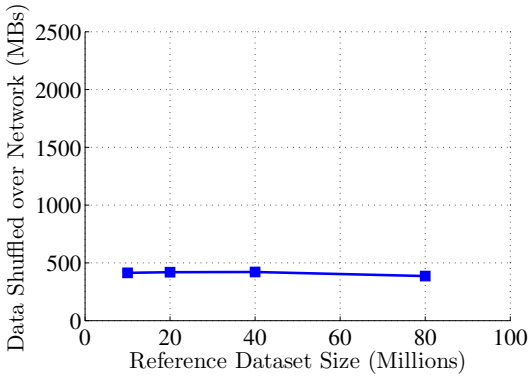
The scalability and elasticity of DIMO are obtained mainly by our design of the distributed index, which partitions the datasets into independent and non-overlapping bins. These bins are allocated independently to computing machines for further processing. This data partitioning and allocation to bins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

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.

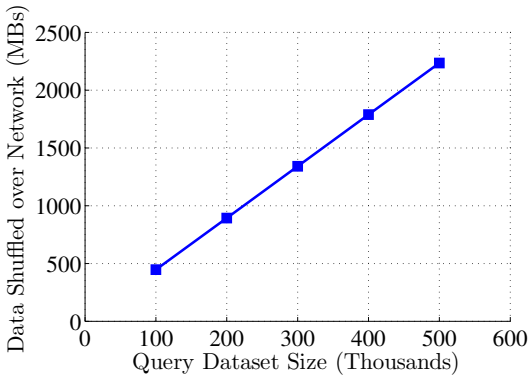
4.5 Overhead Analysis of DIMO

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

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 8. We obtain the



(a) Different sizes of reference datasets



(b) Different sizes of query datasets

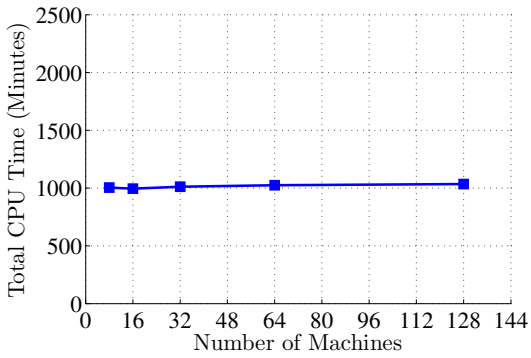
Figure 8: Network overhead imposed by DIMO.

amount of data exchanged from Hadoop logs. In Figure 8(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 size of the distributed index.

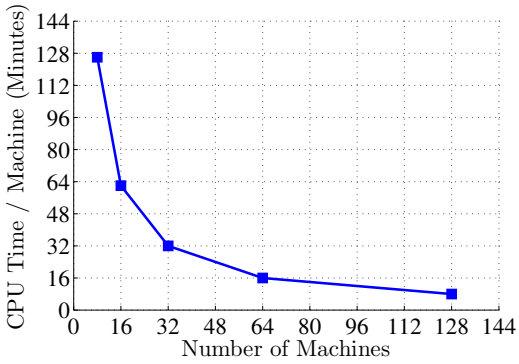
In Figure 8(b), we vary the query dataset size from 100,000 to 500,000 data points, and we fix the reference dataset size at 10 million data points data points. The results show a linear increase in the total amount of data shuffled across the network as the size of the query dataset increases.

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 bin 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



(a) Total CPU time across all machines



(b) Average CPU time per machine

Figure 9: Analysis of CPU usage by DIMO for different cluster sizes.

136 MB of storage on the distributed file system. 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 directing tree) only once on the distributed file system. The size of the directing tree depends on the number of data points in the reference dataset. However, as mentioned in Section 3.2.1, the nodes in the directing tree do not store data points; they only store meta data (splitting values). This meta data is a scalar value, which means that each node in the tree needs up to 4 bytes. Therefore, the directing tree takes a negligible space on the storage system compared to the reference dataset.

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.

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 9. We obtain these measurements from Hadoop logs. The results in Figure 9(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 9(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.

5. CONCLUSIONS

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 installed and 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 [27]. 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 three orders of magnitudes less storage.

Acknowledgments

This work is partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada and the British Columbia Innovation Council (BCIC).

6. REFERENCES

[1] M. Aly, M. Munich, and P. Perona. Distributed Kd-Trees for Retrieval from Very Large Image Collections. In *Proc. of British Machine Vision Conference (BMVC)*, 2011.

[2] M. Aly, P. Welinder, M. Munich, and P. Perona. Scaling Object Recognition: Benchmark of Current State of the Art Techniques. In *Proc. of IEEE Workshop on Emergent Issues in Large Amounts of Visual Data (WS-LAVD)*, pages 2117–2124, 2009.

[3] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proc. of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–468, 2006.

[4] S. Arya and D. Mount. Algorithms for fast vector quantization. In *Proc. of Data Compression Conference (DCC)*, pages 381–390, 1993.

[5] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, pages 509–517, 1975.

[6] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbor meaningful? In *Proc. of Conference on Database Theory (ICDT)*, pages 217–235, 1999.

[7] S. Blott and R. Weber. A simple vector-approximation file for similarity search in high-dimensional vector spaces. *ESPRIT Technical Report TR19, ca*, 1997.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of Conference on Very Large Databases (VLDB)*, 1997.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[10] S. Deegalla and H. Bostrom. Reducing high-dimensional data by principal component analysis vs. random projection for nearest neighbor classification. In *Proc. of Conference on Machine Learning and Applications (ICMLA)*, pages 245–250, 2006.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[12] F. Falchi, C. Gennaro, and P. Zezula. A content-addressable network for similarity search in metric spaces. In *Proc. of conference on Databases, information systems, and peer-to-peer computing (DBISP2P)*, pages 98–110, 2007.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of Conference on Very Large Data Bases (VLDB)*, pages 518–529, 1999.

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

[15] P. Haghani, S. Michel, and K. Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *Proc. of Conference on Extending Database Technology (EDBT)*, pages 744–755, 2009.

[16] N. Khodabakhshi and M. Hefeeda. Spider: A system for finding 3d video copies. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 9(1), 2013.

[17] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *Proc. of IEEE Conference on Computer Vision (ICCV)*, pages 2130–2137, 2009.

[18] H. Liao, J. Han, and J. Fang. Multi-dimensional index on hadoop distributed file system. In *Proc. of IEEE Conference on Networking, Architecture and Storage (NAS)*, pages 240–249, 2010.

[19] D. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, pages 91–110, 2004.

[20] W. Lu, Y. Shen, S. Chen, and B. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment (PVLDB)*, 5(10):1016–1027, 2012.

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

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

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 161–172, 2001.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

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

[26] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 149–160, 2001.

[27] A. Stupar, S. Michel, and R. Schenkel. Rankreduce - processing k-nearest neighbor queries on top of mapreduce. In *Proc. of Workshop on Large-Scale Distributed Systems for Information Retrieval(LSDS-IR)*, pages 13–18, 2010.

[28] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. `http://www.vlfeat.org/`, 2008.

[29] J. Wang, S. Wu, H. Gao, J. Li, and B. Ooi. Indexing multi-dimensional data in a cloud system. In *Proc. of ACM Conference on Management of data (SIGMOD)*, pages 591–602, 2010.

[30] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. of ACM Symposium on Discrete algorithms (SODA)*, pages 311–321, 1993.

[31] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proc. of Conference on Extending Database Technology (EDBT)*, pages 38–49, 2012.

[32] H. Zhang, A. Berg, M. Maire, and J. Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2126–2136, 2006.