Searching Large-Scale Image Collections

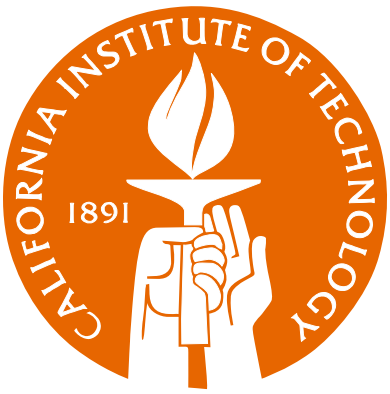
Thesis by

Mohamed Alaa El-Dien Mahmoud Hussein Aly

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2011

(Defended May 23, 2011)

To my family: my mother, my father, my wife and kids, and my sister. They have always helped and supported me.

Acknowledgments

First and foremost, all thanks are due to God, for giving me the strength and persistence to go through six years of graduate work.

Second, I would like to thank my thesis adviser, Prof. Pietro Perona, for being an exceptional adviser, both academically and professionally. I learned a lot from his knowledge, insight, and guidance.

Third, I would like to thank Prof. Yaser Abu-Mostafa. He was my initial co-adviser at Caltech. He gave me invaluable advice and patiently and graciously helped me refine and define my thesis topic.

Fourth, I would like to thank Dr. Mario Munich for hosting me for an internship at Evolution Robotics, and for collaborating with me and providing me with advice during all my thesis work.

Fifth, I would like to thank Dr. Jean-Yves Bouguet for hosting me for one internship at Google, and helping me get a second one. I would also like to thank Dr. Drago Anguelov, Dr. James Philbin, Dr. Hartwig Adam, and Dr. Hartmut Neven for their great help in the second internship at Google, in which I was able to perform the large-scale distributed experiments of Chapter 7.

Sixth, I would like to thank my colleagues at Caltech, both inside and outside the Vi-

sion Lab, who helped make my time possible. Specifically, I would like to thank Peter Welinder for helping me collect some of the datasets, and co-authoring a couple of papers. I would also like to thank Marco Andreetto and Ahmed Elbanna for helpful and insightful discussions.

Finally, I would like to thank my family. My parents have always supported and pushed me, starting from elementary school and up to graduate school. My wife and kids also supported me and have been very patient with me taking time away from them in order to finish my homeworks, exams, and research work.

Abstract

Searching quickly and accurately in a large collection of images has become an increasingly important problem. The ultimate goal is to make visual search possible: allow users to search using images in addition to typing text. The typical approach is to index all the images of interest (e.g., images of landmarks, books, or DVDs) in a database and let users question the system with query images. Such a database can reach billions of images, and this poses challenges in terms of memory and computational requirements and recognition performance. In this work we provide an in depth study of systems used for searching large-scale image collections.

Specifically, we provide a thorough comparison of the two leading image search approaches: Full Representation (FR) vs. Bag of Words (BoW). We derive theoretical estimates of how the memory and computational cost scale with the number of images in the database, and empirically evaluate the performance and run time on four real-world datasets. Our experiments suggest that FR provides better recognition performance than BoW, though it requires more memory. Therefore, we address these shortcomings by presenting novel methods that increase the recognition performance of BoW and decrease the memory requirements of FR. Finally, we present a novel way to parallelize FR on multiple machines and scale up database sizes to 100 million images with interactive run time.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Searching for a specific object in a large-scale collection of images has become an increasingly important problem with numerous applications, especially with the popularity of smart phones. There are currently several applications that allow users to take a photo with the smart phone camera and search a database of stored images, e.g., Google Goggles and Barnes and Noble. The ultimate goal of such applications is to make visual search a reality. In other words, to allow users to search the Internet using images, as it is possible now to search the Internet using text.

Typical scenarios for visual search include searching images of book covers, DVD covers, retail products, and buildings and landmarks. The size of databases involved vary from hundreds of thousands to potentially millions of images, but they could conceivably reach billions. After building an index using these images, users query the database with a probe image containing an object of interest, e.g., an image of a book cover from a certain viewpoint and scale. The system should respond with the identity of that book together with some useful information about it, like links to buy it online. All this process should take only a few seconds, and should work with high precision.

In this thesis we focus on how to successfully build such a system. Our goal is to have a

working image search system that is physically realizable, works with high precision, and is scalable to hundreds of millions of images, all while working interactively with users. Obviously building such a system is not an easy task, and poses a lot of challenges, namely memory requirements, computational cost, and recognition performance. For example, if we consider 1 billion images, and store on average 100 KB per image, we need to store 100 TB of data just for the feature descriptors, and naively searching for the nearest feature in a database of 10^{12} features takes 4 minutes on a supercomputer with 1 TFLOPS. This is clearly not satisfactory: most applications of interest are interactive and require fast response time on the order of a few seconds. In turn, this implies the need to store image descriptors in storage that is very close to the processor, e.g., RAM (with top-of-the-line machines nowadays having around 50 GB of RAM).

To this end, we start with a comprehensive comparison of the two leading image search approaches, and study how their properties scale with large number of images both theoretically and experimentally. Both approaches are based on extracting local features from the images (see Chapter 2 for details), and then indexing these features or some information extracted therefrom. This information is then used to quickly find the best match of a probe image in the database images. The first approach, the *Full Representation* (FR) approach, stores the features of the database image in exact or compressed form, and efficiently indexes each feature of the probe image into this database. The second approach, the *Bag of Words* (BoW), is based on quantizing features into visual words and representing each image with a histogram of visual words. Our experiments suggest that that FR methods have better recognition performance with larger memory requirements, while BoW have

better memory usage with worse performance (see Chapter 4).

Based on this comparison, we then explore ways to remedy the shortcomings of both approaches. Specifically, we explore ways to reduce the memory usage of FR methods, specially with Kd-Trees (Chapter 5). We present Compact Kd-Trees, which are able to achieve an order of magnitude less memory usage by compressing the local features, while achieving comparable recognition performance. We also explore ways to boost the recognition performance of BoW methods (Chapter 6). We present Multiple Dictionaries for BoW, which is able to significantly boost the recognition performance of BoW approach, albeit with more computational and memory requirements. Finally, we focus on the parallelization of FR methods, specially Kd-Trees, so that the system can handle millions of images (Chapter 7). We present Distributed Kd-Trees, which provide excellent recognition performance running on a database with 100 million images while processing input images in a fraction of a second.

This thesis makes a number of contributions:

1. We provide a comprehensive comparison of the two leading image indexing approaches: Full Representation and Bag of Words. In particular, we provide:

   (a) Theoretical estimates of the memory requirements, computational cost, and parallelizability of these methods as a function of the number of images.

   (b) Experimental evaluation of these different methods on four real world datasets. We report the recognition performance and run time as the number of images grows.

2. We challenge the conventional wisdom in image indexing methods. We argue that

the FR approach is the way to go, since, although it requires an order of magnitude more storage, it provides superior recognition performance to BoW, especially with large datasets.

3. We present novel methods to remedy some of the shortcomings of these two methods:

   (a) Compact Kd-Trees that are able to cut the memory usage and run time of FR methods by an order of magnitude while achieving comparable recognition performance.

   (b) Multiple Dictionaries for BoW that are able to significantly boost the recognition performance of BoW methods to levels comparable to FR methods, at the expense of increased memory and computational costs.

4. We present a novel way of parallelizing Kd-Trees, Distributed Kd-Trees, and run experiments on thousands of machines with 100 million images. The system outperforms the state-of-the-art in both recognition performance and throughput, and can process a query image in a fraction of a second.

The thesis is organized as follows: in Chapter 2 we give an overview of the approaches explored in the rest of the thesis and the basic image search algorithm considered. Chapter 3 details the theoretical estimates of the different properties of these approaches and how they scale up with billions of images. Chapter 4 presents the experimental evaluation of the different methods on four datasets. Chapter 5 explains a novel method, Compact Kd-Tree, to reduce the memory usage of the FR approach with comparable recognition performance. Chapter 6 presents a novel method, Multiple Dictionaries for BoW, to boost the recogni-

tion performance of the BoW approach. Chapter 7 gives the implementation details of Distributed Kd-Trees, and presents experiments on thousands of machines with millions of images. Finally, Chapter 8 details the conclusions of this thesis and discusses future work.

Chapter 2

Methods Overview

2.1 Introduction

In this chapter we give an overview of the image search problem considered in this thesis. We then explain the basic image search algorithm, and its relation to the two leading approaches: Full Representation and Bag of Words. Finally, we finish by describing the different variants of these two approaches that we consider later in the comparison. Section 2.2 describes the image search problem we consider in this thesis. Section 2.3 describes the different image representations. Section 2.4 gives an overview of the basic image search algorithm. Finally, Section 2.5 describes the Full Representation approach, followed by the Bag of Words approach in Section 2.6.

2.2 Image Search Problem

The basic problem we consider in this thesis is image search. We have a database that stores images of objects of interest, for example DVD covers, book covers, and landmarks (see Figure 2.1). The system allows users to take images of different objects, for example

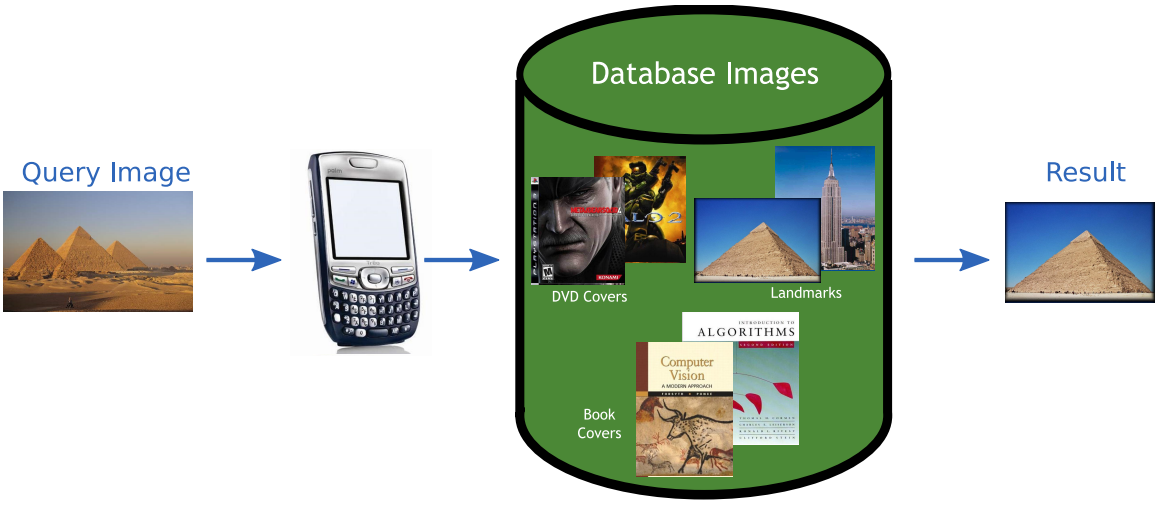


Figure 2.1: **Basic Image Search Problem**. The system contains a database that indexes images of objects of interest. Users can query the system using images taken with their cell phones. The system searches its database and replies with the identity of the query object together with some useful information.

an image of the pyramids, and to query the database using that query image. The system then searches its database and responds with the identity of the object depicted in the query image, presenting its stored canonical image and any additional information about that object.

The image search can be done in one of two ways: Object Category Recognition or Specific Object Recognition (see Figure 2.2). In category recognition, the database images and the query images do not necessarily represent objects of the same identity, but of the same category. For example, the query images for object category "sedan car", can represent a Honda Accord, while there might not be an image of such object in the database. The goal of such systems is to identify the category of the object, e.g., a car vs. a bird, rather than identifying the identity of the object. In specific object recognition, on the other hand, the goal is to retrieve the correct identity of the object, in this case the make and

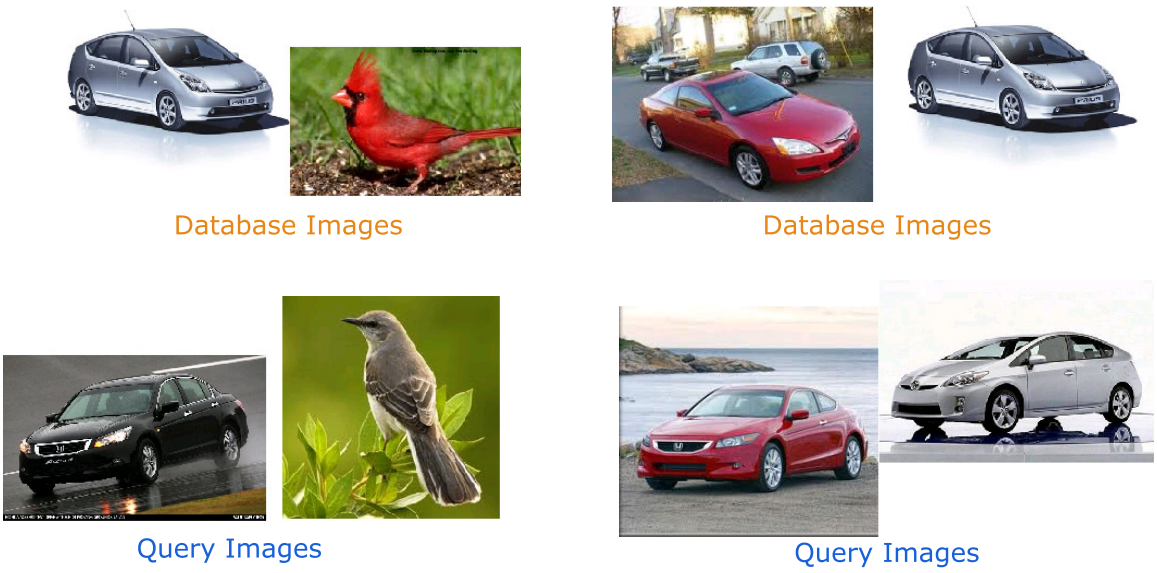


Figure 2.2: **Object Category Recognition** (*left*) **Vs. Specific Object Recognition** (*right*). In this thesis, we focus on specific object recognition.

model of the car. In this thesis we focus only on specific object recognition rather than category recognition. The systems we are interested in should return the identity of the specific object depicted in the query image.

Specific object recognition can in turn be divided into three cases (see Figure 2.3): (a) Scene to Object: where the database image depicts a canonical cropped version of the object (for example Eiffel Tower), while the query image contains the object of interest in addition to other objects. (b) Object to Scene: the opposite of case (a), where the database image contains different objects of interest, while the query image contains only one object. (c) Object to Object: where both the query and database images contain clean cropped versions of the object of interest. In this thesis, we focus on the third case.

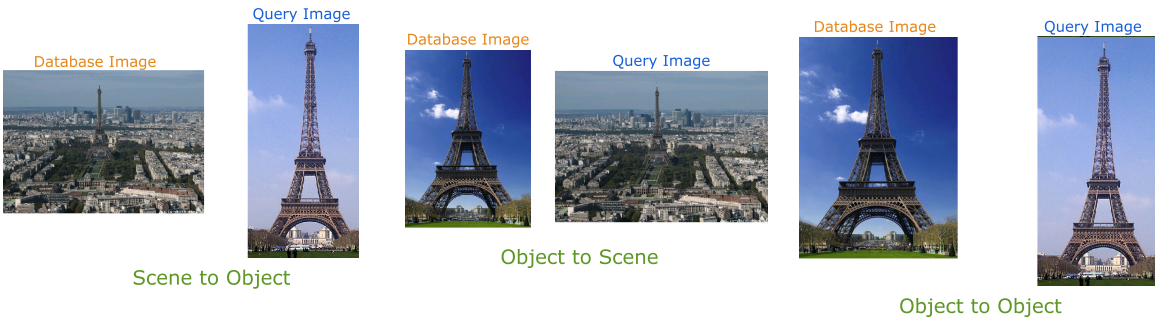


Figure 2.3: **Object Recognition Flavors**. (a) Scene to Object: database image depicts cropped version of objects while query image contains different objects. (b) Object to Scene: opposite of (a). (c) Object to Object: both query and database images contain only one clean cropped version of the object. In this thesis, we focus on (c).

2.3 Image Representation

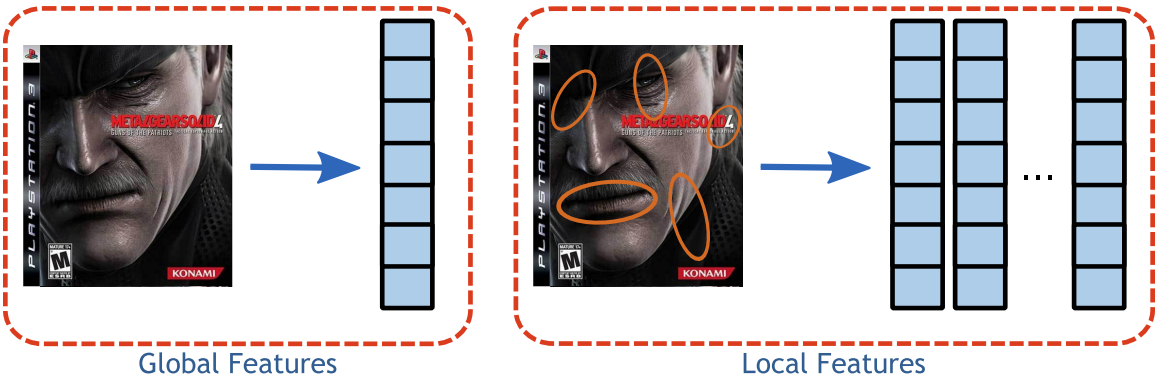


Figure 2.4: **Image Representations**. Global features (*left*) represent an image by one multi-dimensional feature descriptor, whereas local features (*right*) represent an image by a set of features extracted from local regions in the image.

For any image processing operation, we need to represent an image by features extracted therefrom. The raw image is perfect for the human eye to extract all information from, however that is not the case with computer algorithms. There are generally two ways to

represent images in computer vision (see Figure 2.4):

1. **Global features**: where the image is represented by one multi-dimensional feature, describing the information in the image. The information can be color histograms [18], edge magnitude or orientation histograms [15], or a specific descriptor extracted from some filters applied to the image, such as GIST features [28]. The advantages of global features are that they are fast and easy to compute and generally require small amounts of memory. However, they have been shown to perform worse than the other type, local features [17].

2. **Local features**: where the image is represented by a set of local feature descriptors extracted from a set of regions around the image. There are generally two components for local features: a feature detector and a feature descriptor [18, 25]. A feature detector detects interesting locations in the image, for example corners and edges. A feature descriptor describes the image patch around that interest point, usually by histograms of gradients or orientation. There are different kinds of feature detectors, but among the most common ones are Difference of Gaussians (DoG) [24], Hessian-Affine, and Harris Affine [25]. Similarly there are various feature descriptors, but by far SIFT [24] and HOG [15] are the most widely used. The main advantage of local features is their superior performance, however they usually have much larger memory usage, as a typical image can have hundreds of local features [17].

In this thesis we focus on using local features for large-scale image search, since they have much higher performance than global features provide.

Algorithm 2.1 Basic Image Search Algorithm

Image Index Construction

1. Extract local features $\{\mathbf{f}_{ij}\}_j$ from every database image i.
2. Store some function of these features $g(\mathbf{f}_{ij})$ and build data structure $d(g)$ based on $g()$. In the data structure each feature $g(\mathbf{f}_{ij})$ is associated to one (or more) database images that contain that feature.

Image Index Search

1. Given the probe image, extract its local features $\mathbf{f}_{qj}$ and compute $g(\mathbf{f}_{qj})$.
2. Search through the data structure $d(g)$ for "nearest neighbors".
3. Every nearest neighbor votes for one (or more) database image(s) i that it is associated to.
4. Sort the database images based on their score s_i.
5. Post-process the sorted list of images to enforce some geometric consistency and obtain a final list of sorted images s'_i. The geometric consistency check is done using a RANSAC algorithm to fit an affine transformation between the positions of the matched features in the query image q and the database image i.

2.4 Basic Image Search Algorithm

The basic local features image search algorithm is depicted in Figure 2.5 and detailed in Algorithm 2.1. The two main steps are:

1. **Training phase**: where the image index is constructed. Local features are extracted from the input images, and are inserted (optionally after some post-processing) into a data structure that allows for fast approximate nearest-neighbor search. Several types of these data structures, or indices, will be discussed in Sections 2.5 and 2.6.

2. **Query phase**: where the image index is searched for a query image. Every local feature in the query image is used to search the database features for nearest neighbors. The score of every database image that has such nearest neighbors is increased. The

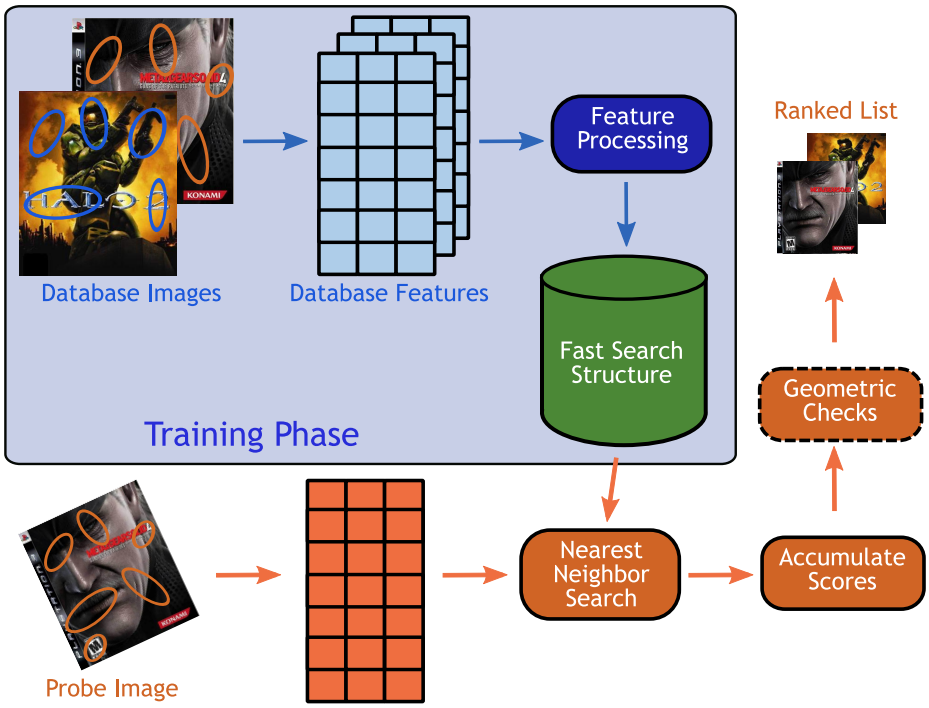


Figure 2.5: **Basic Image Search Algorithm**. In the training phase, local features are extracted from the database images, optionally processed, and then inserted into a fast search data structure. At query time, local features are extracted from the query image, and for each such feature the index is searched for nearest neighbors. Scores are accumulated for each database image, and a final ranked list of result images is presented to the user. See Algorithm 2.1.

system can then perform post-processing to verify the geometric consistency of the feature locations in the image. The final result is a ranked list of database images, with high ranked images being more likely to correspond to the query image.

There are two leading approaches for building such large-scale image indexing systems:

1. **Full Representation (FR)**: where the feature space is searched directly for nearest neighbors using the full features [24, 4, 3] (see Figure 2.6).

2. **Bag of Words (BoW)**: where some quantization of the feature space is used as a proxy for searching for approximate nearest neighbors [32, 30, 12, 3] (see Figure

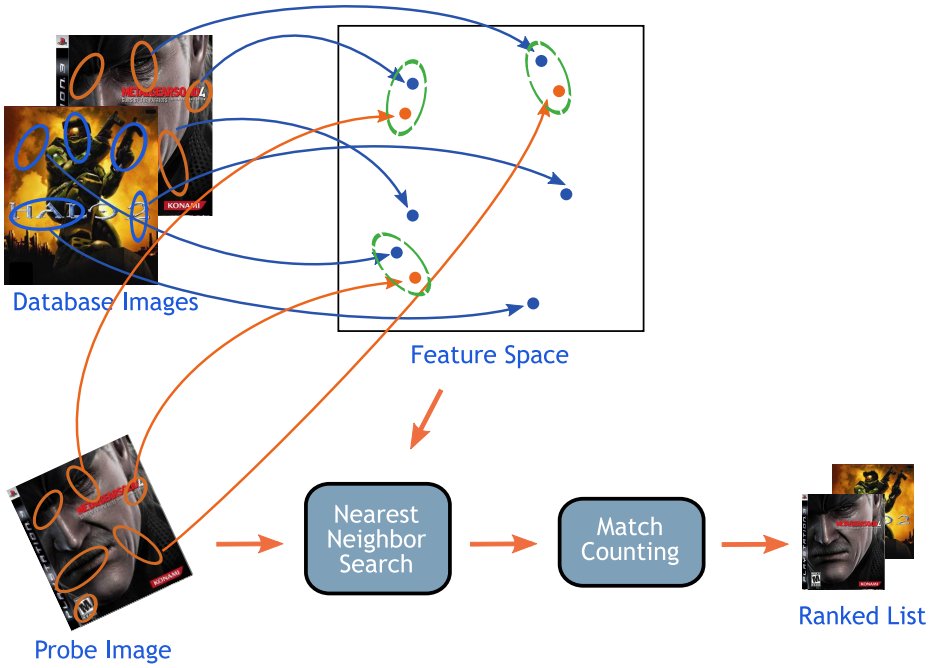


Figure 2.6: **Full Representation Image Search**. The nearest-neighbor search is done directly in the feature space, as opposed to BoW search which uses quantization in the feature space (see Figure 2.7).

2.7).

Each of these two approaches have a number of variants that use different techniques to perform the actual search. These techniques have a lot of parameters that affect their recognition performance, run time, and memory requirements. In the following we will briefly describe these techniques and list their parameters, and we will explore their effect experimentally in Chapter 4.

2.5 Full Representation (FR) Image Search

In FR image search (see Figure 2.6), the feature space is searched directly for nearest neighbors using a variety of specific data structures that provide fast nearest-neighbor search [3].

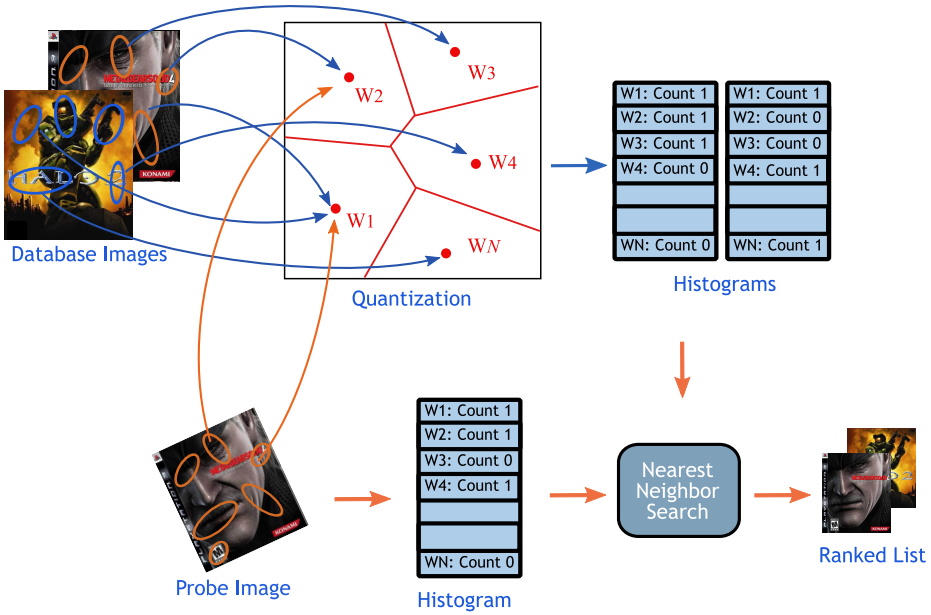


Figure 2.7: **Bag of Words Image Search**. The nearest-neighbor search is done indirectly via vector quantization of the feature space, as opposed to directly searching the feature space as in FR search (see Figure 2.6).

The specifics from Algorithm 2.1 are:

1. The function $g(\mathbf{f}_{ij}) = \mathbf{f}_{ij}$ is the identity function, i.e., the full features are stored

2. The data structure $d(\{\mathbf{f}_{ij}\}_{ij})$ is designed for efficient nearest neighbor lookup at the expense of some additional storage.

We consider three general methods for performing nearest neighbor search in the feature space: Kd-Trees, Locality Sensitive Hashing, and Hierarchical K-Means.

2.5.1 Kd-Trees (Kdt)

One (or more) randomized Kd-trees [6, 24] are built for the database features $\{\mathbf{f}_{ij}\}_{ij}$ to allow for approximate nearest-neighbor matching in logarithmic time [6] (see Figure 2.8).

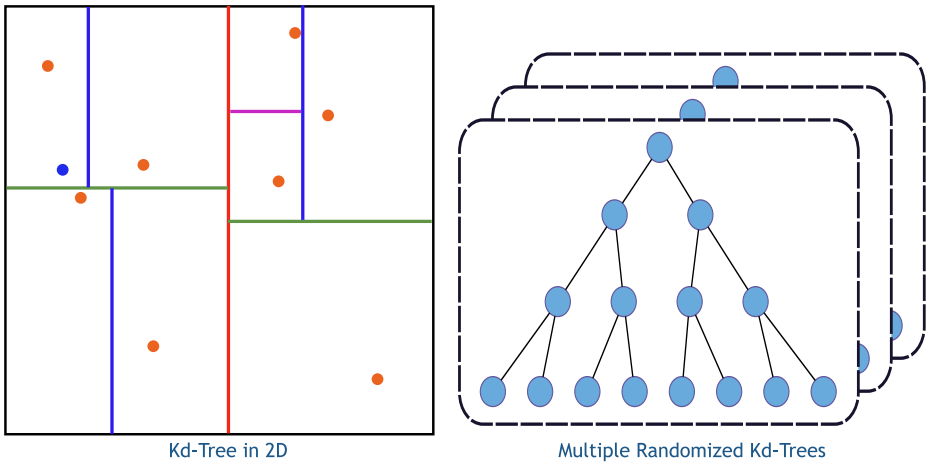


Figure 2.8: **Kd-Trees (Kdt)**. (*Left*) A Kd-Tree in two dimensions. (*Right*) A set of randomized Kd-Trees. See Section 2.5.1.

Algorithm 2.2 explains the Kd-Tree construction procedure, while Algorithm 2.3 details the search procedure.

Kd-Trees have two parameters that affect their recognition performance, storage, and run time:

1. T: the number of trees, where having more trees increases accuracy without much affecting speed, due to the single search queue. However, this increases the storage required due to the need to store the additional trees.

2. B: the number of backtracking steps, which poses a trade off between accuracy and speed. Having more steps will give more accuracy but takes more time.

2.5.2 Locality Sensitive Hashing (LSH)

A number of locality sensitive hash functions [5, 23] are extracted from the database features $\{\mathbf{f}_{ij}\}_{ij}$, and are arranged in a set of tables (see Figure 2.9). Each table has a set of

Algorithm 2.2 Randomized Kd-Trees Construction

Input: A set of vectors $\{x_i\} \in \mathbb{R}^N$
Output: A set of binary Kd-Trees $\{T_t\}$. Each internal node has a split (dim, val) pair where dim is the dimension to split on and val is the threshold such that all points with $x_i[dim] \leq val$ belong to the left child and the rest belong to the right child. The leaf nodes have a list of indices to the features that ended up in that node.
Operation: For each tree T_t:

1. Assign all the points $\{x_i\}$ to the root node.
2. For very *node* in the tree visited in breadth-first order, compute the split as follows:
   (a) For each dimension $d = 1 \ldots N$, compute its mean $mean(d)$ and variance $var(d)$ from the points in that node.
   (b) Choose a dimension d_r at random from the variances within 80% of the maximum variance.
   (c) Choose the split value as the mean of that dimension $mean(d_r)$.
   (d) For all points that belong to this node: if $x[d_r] \leq mean(d_r)$ assign x to $left[node]$, otherwise assign x to $right[node]$.

Algorithm 2.3 Randomized Kd-Trees Search

Input: A set of Kd-Trees $\{T_t\}$, a set of vectors $\{x_i\} \in \mathbb{R}^N$ used to build the trees, a query vector $q \in \mathbb{R}^N$, maximum number of backtracking steps B
Output: A set of k nearest neighbors $\{n_k\}$ with their distances $\{d_k\}$ to the query vector q.
Operation:

1. Initialize a priority queue Q with the root nodes of the t trees by adding $branch = (t, node, val)$ with $val = 0$. The queue is indexed by $val[branch]$, i.e., it returns the branch with smallest val.
2. $count = 0$. $list = []$.
3. While $count \leq B$:
   (a) Retrieve the top *branch* from Q.
   (b) Descend the tree defined by *branch* till $leaf$, adding unexplored branches on the way to Q.
   (c) Add the points in $leaf$ to $list$.
4. Find the k nearest neighbors to q in $list$ and return the sorted list $\{n_k\}$ and their distances $\{d_k\}$.

hash functions, which are then concatenated to get the index of the bucket within the table where the feature should go. All features with the same hash value go to the same bucket. At query time, the hash value is computed for the query feature. Only features in buckets with this value need to be further processed for nearest neighbors. Three different hash functions are considered:

1. **L2**: this approximates the Euclidean distance [5], where the hash function is $h(x) = \left\lfloor \frac{<x,r>+b}{w} \right\rfloor$ where $< .,. >$ is the dot product, r is a random unit vector, b is a random offset, and w is the bin width. For normalized feature vectors, it basically projects the feature onto a random direction and then returns the bin number where the projection lies.

2. **Spherical-Simplex**: this approximates distances on the hypersphere [33], where the hash function is $h(x) = \text{argmin}_i \langle x, y_i \rangle$, where y_i are the vertices of a random simplex inscribed in the unit hypersphere. The hash value is the index of the nearest vertex of the simplex.

3. **Spherical-Orthoplex**: this approximates distances on the hypersphere [33], where the hash function is $h(x) = \text{argmin}_i \langle x, y_i \rangle$, where y_i are the vertices of a random orthoplex inscribed in the unit hypersphere. The hash value is the index of the nearest vertex of the orthoplex.

There are generally two parameters for any LSH method:

1. T: the number of tables. Generally, having more tables improves performance but increases run time as well.

2. H: the number of hash functions. Having more functions increases run time, but makes collisions, and hence bucket size, less.

In addition, L2 LSH includes one more parameter, which is w, the bin size for the projection.

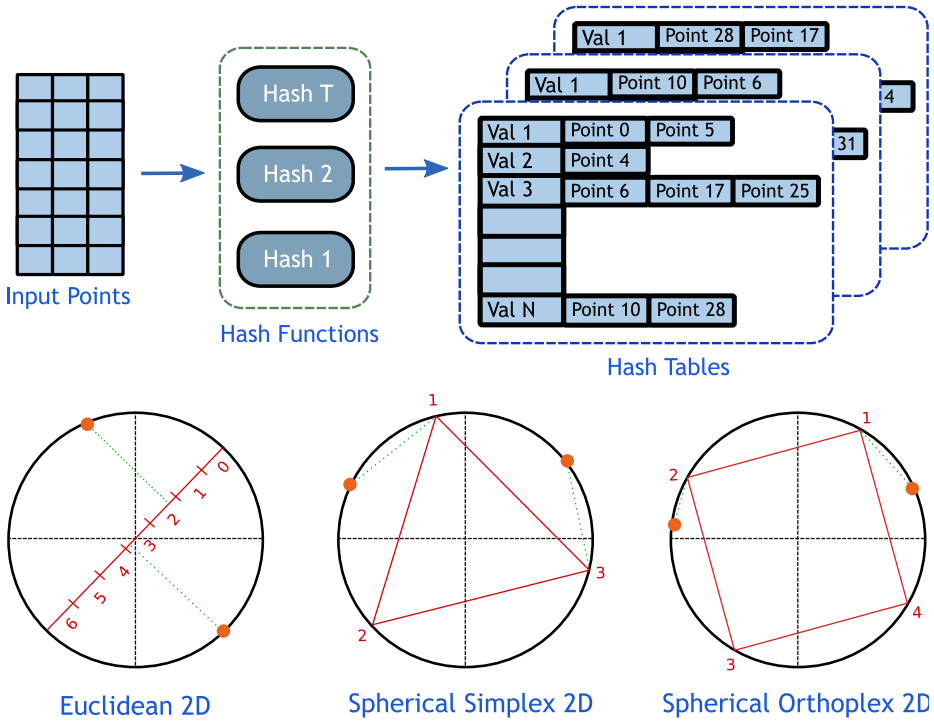


Figure 2.9: **Locality Sensitive Hashing (LSH)**. (*Top*) A schematic of LSH search. A set of hash functions are extracted for each feature, and features with the same hash values go to the same bucket in each hash table. At query time, only buckets with the same hash value as the query feature need to be searched. (*Bottom*) Three different hash functions in two dimensions: L2, Spherical Simplex, and Spherical Orthoplex. See Section 2.5.2.

2.5.3 Hierarchical K-Means (HKM)

A hierarchical decomposition [27] is built from the database features $\{\mathbf{f}_{ij}\}_{ij}$. At each level of the tree, the features that are associated with the current node are clustered using K-means, and the process is repeated recursively until the maximum depth allowed is reached

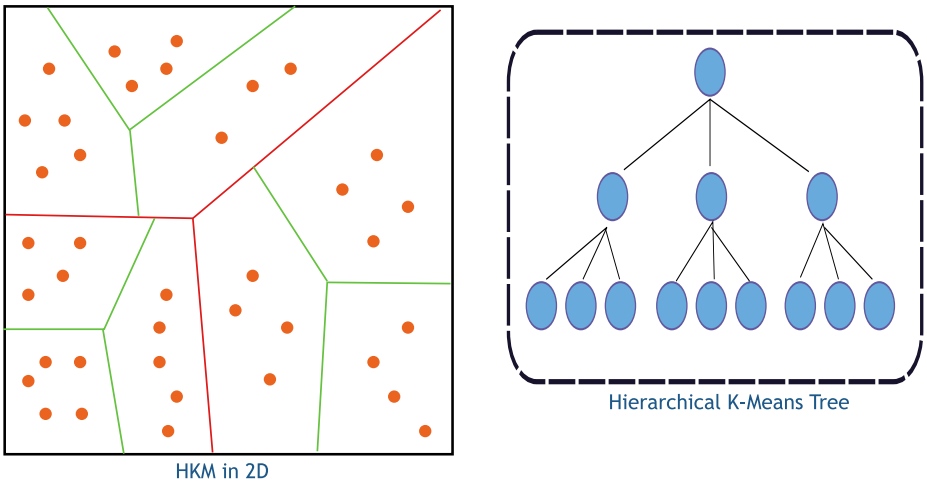


Figure 2.10: **Hierarchical K-Means (HKM)**. (*Left*) HKM in two dimensions. (*Right*) A schematic of the HKM search tree. See Section 2.5.3.

(see Figure 2.10). At query time, the tree is traversed using the query feature, until it reaches a leaf, in which case those features in the leaf are processed for the nearest neighbor. Backtracking can also be performed on the tree [26], but is not considered in this work.

There are two parameters that affect the amount of storage and running time of HKM:

1. D: the maximum depth of the tree. A deeper tree requires more storage, but reduces the run time (for a fixed number of features), as the number of features in the leaves will be smaller.

2. k: the branching factor of the tree. A larger k also requires more storage, since we will need to store more cluster centers in the internal nodes. However, for a fixed number of features, it reduces the run time since it creates more leaf nodes in the bottom of the tree.

2.6 Bag of Words (BoW) Image Search

In BoW image search (see Figure 2.7), the nearest-neighbor search is done using quantization in the feature space. Two features are considered matched if they belong to the same cluster, or "visual word", in the feature space [32, 12, 30, 3]. This is inspired by the text search literature where documents are represented by the frequency of the words they contain [36]. The specifics from Algorithm 2.1 are:

1. The function $g(\mathbf{f}_{ij})$ represents a vector quantization of the input features. The "dictionary" used for quantization is built from the database images by clustering features into representative "visual words". Then, each image is represented by a histogram of occurrences of these visual words $\{\mathbf{h}_i\}_i$, and the original local feature vectors are thrown away.

2. The data structure $d(\{\mathbf{f}_{ij}\}_{ij})$ is designed for efficient nearest-neighbor lookup for histograms of visual words $\{\mathbf{h}_i\}_i$.

In the training phase, the image features are "quantized", i.e., each feature is assigned to the closed visual word. Then, the images are represented by histograms of these visual words, which are then stored in a data structure that allows fast histogram search (see Sections 2.6.1 and 2.6.2). At query time, the histogram of the query image is computed, and used to query the data structure for the image with the closest histogram. We consider two data structures that try to perform faster search through the database histograms: Inverted File and Min-Hash.

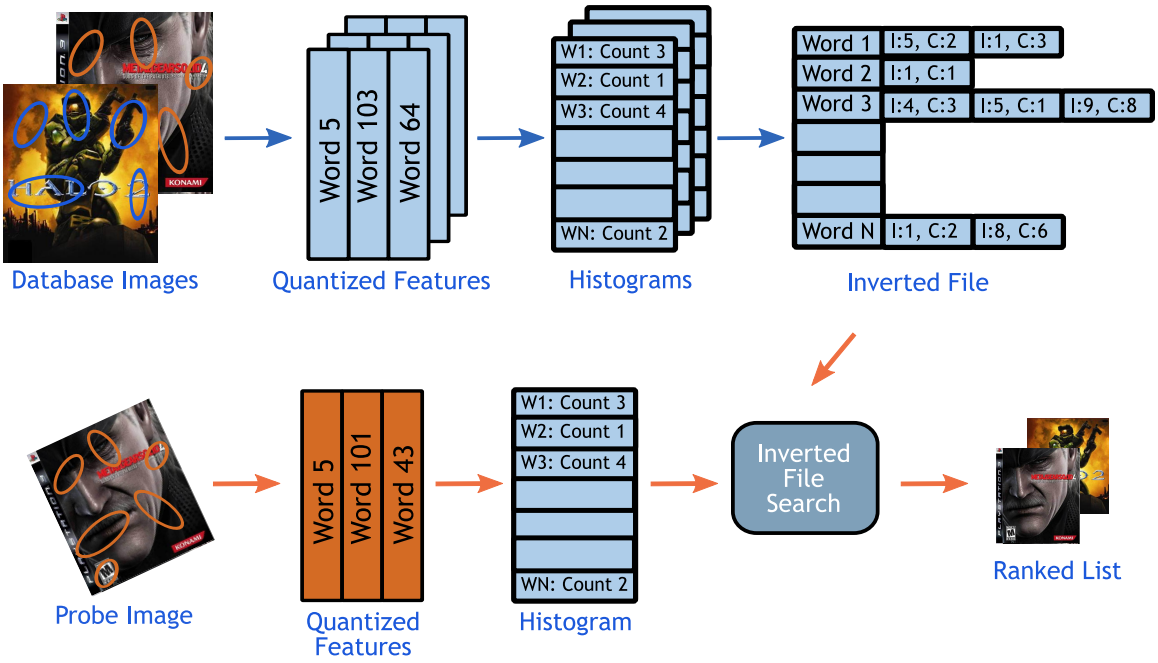


Figure 2.11: **BoW Inverted File (IF) Search**. The image features are extracted, quantized, and then converted into histograms of visual words. The histograms are stored in an inverted file, which is then searched with the histogram of the query image for the nearest neighbor. See Section 2.6.1.

2.6.1 Inverted File (IF)

The idea is to store for every visual word the list of images that contain it (see Figure 2.11). At run time, only images with overlapping words are processed, and this saves a lot of time and provides exact search results [7, 36].

Inverted files have several parameters that affect performance and storage requirements, including the number of visual words, histogram weighting, normalizations, dictionary generation method, and distance functions:

- **W**: the number of visual words. Typically, this is in the order of hundreds of thousands to a million [30, 20, 2]. Using fewer visual words usually results in worse performance, while using a much larger number of visual words also hurts perfor-

mance, except when done in a special way (see Chapter 6). Also using more visual words increases the search speed due to the lower probability of overlap between image histograms.

- **Weighting**: whether to use the raw histogram, or modify the histogram counts

  1. **none**: use the raw histogram
  2. **binary**: binarize the histogram, i.e., just record whether the image has the visual word or not
  3. **tf-idf**: weight the counts in such a way to decrease the influence of more common words and increase the influence of more distinctive words

- **Normalizations**: how to normalize the histograms

  1. l_1: normalize so that they sum to one, i.e., $\sum_i |h_i| = 1$
  2. l_2: normalize so they have unit length, i.e., $\sum_i h_i^2 = 1$

- **Distance**: what distance function to use between histograms

  1. l_1: use the sum of absolute differences, i.e., $d_{l1}(h,g) = \sum_i |h_i - g_i|$
  2. l_2: use the sum of squared differences, i.e., $d_{l2}(h,g) = \sum_i (h_i - g_i)^2$
  3. cos: use the dot product, i.e., $d_{\cos}(h,g) = 2 - \sum_i h_i \times g_i$. (Note that for l_2-normalized histograms, this is equivalent to l_2 distance since $||h||_2 = ||g||_2 = 1$ and $d_{l2} = ||h-g||_2^2 = ||h||_2^2 + ||g||_2^2 - 2\sum_i h_i g_i = 2 - 2\sum_i h_i g_i$, i.e., they give the same ordering of the histograms).

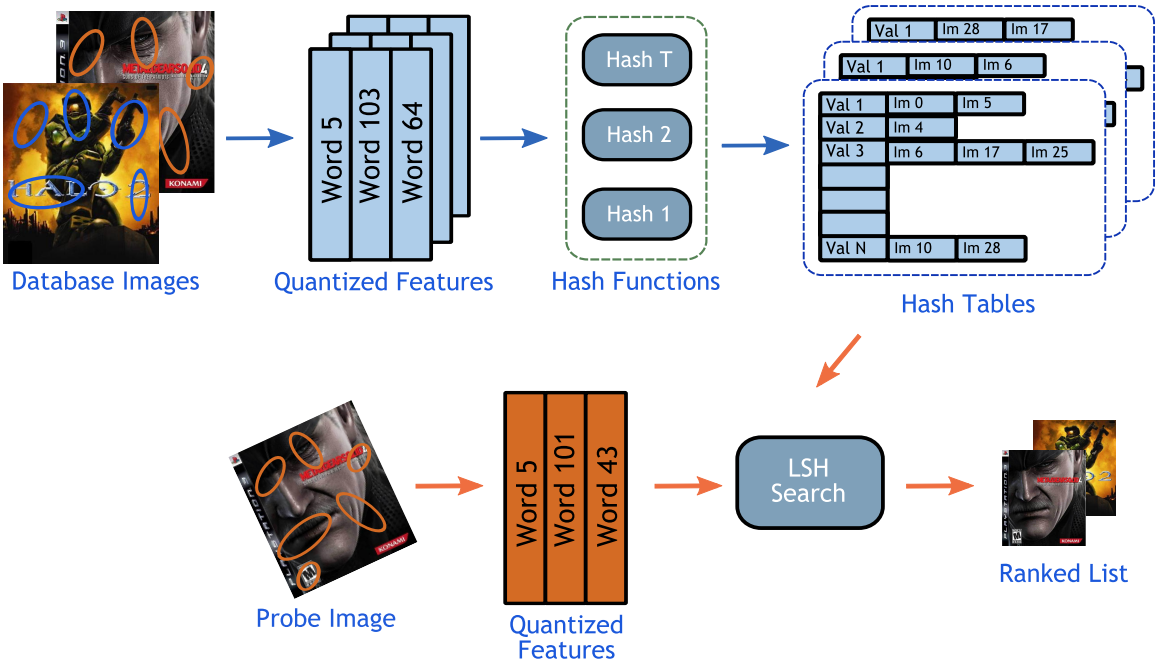


Figure 2.12: **BoW Min-Hash (MH) Search**. The image features are extracted, quantized, and then converted into histograms of visual words. The histograms are binarized, and used to build a set of Min-Hash tables. The histogram of the query image is then used to search for the nearest neighbor in Min-Hash tables. See Section 2.6.2.

- **Dictionary Generation**: what method to use for generating the visual words (feature space quantization)

  1. **Approximate K-Means (AKM)**: which approximates the nearest-neighbor search within K-Means using a set of randomized Kd-Trees [30].

  2. **Hierarchical K-Means (HKM)**: which builds a vocabulary tree by applying K-Means recursively [27] at each node in the tree.

2.6.2 Min-Hash (MH)

A number of locality sensitive hash functions [9, 10, 8, 12] are extracted from the database histograms $\{\mathbf{h}_i\}_i$, and are arranged in a set of tables (see Figure 2.12). The histograms are

binarized (counts are ignored), and each image is represented as a "set" of visual words $\{\mathbf{b}_i\}_i$. The hash function is defined as $h(\mathbf{b}) = \min \pi(\mathbf{b})$ where π is a random permutation of the numbers $\{1,...,W\}$, where W is the number of words in the dictionary.

There are three parameters that affect the performance of MH image search:

1. W: the number of visual words in the dictionary. Generally, the larger the dictionary, the better the performance and the faster the actual search, because the number of overlapping images is smaller.

2. T: the number of hash tables.

3. H: the number of hash functions.

2.7 Summary

Full name	Abbreviation	Section
Full Representation	FR	2.5
Exhaustive	exhaustive	2.5
Kd-Trees	Kdt	2.5.1
Locality Sensitive Hashing	LSH	2.5.2
LSH with L2	LSH-L2	2.5.2
Spherical LSH with Simplex	LSH-Sim	2.5.2
Spherical LSH with Orthoplex	LSH-Orth	2.5.2
Hierarchical K-Means	HKM	2.5.3
Bag of Words	BoW	2.6
Inverted File	invf or IF	2.6.1
Min-Hash	minhash or MH	2.6.2

Table 2.1: **Search Methods Full Names and Abbreviations**.

In this chapter we introduced the basic image search algorithm that is based on local features. We introduced the two leading approaches for performing large-scale image

search: Full Representation and Bag of Words. We briefly described the different variants of these two approaches, together with their design parameters. For convenience, Table 2.1 lists all the methods and their abbreviations that are used throughout the thesis. In the next chapter we will detail the theoretical analysis of the properties of these methods and how they scale with very large numbers of images.

Chapter 3

Theoretical Comparison

3.1 Introduction

Chapter 2 introduced the different methods used to build large-scale image search systems. These methods have different characteristics, and their properties behave differently when the size of the database increases. We are specifically interested in three essential properties, and how they scale up with the number of images:

1. Memory/Storage: How much memory/storage is needed for the method to work properly?

2. Computational Cost: How much time or how many computational steps does it need to search for a query image?

3. Recognition Performance: What is the precision of the search results obtained?

In this chapter we provide theoretical estimates of how the storage and computational cost of these methods scale with the number of images in the database. We need these theoretical estimates since, we cannot physically run experiments involving billions of images, but we are still interested in exploring how the methods behave with such large databases. Section

3.2 summarizes the theoretical estimates of these properties while Section 3.3 provides the comparison and discussion of these estimates. Finally, Section 3.4 details the derivations of these formulas.

3.2 Theoretical Estimates

We estimate formulas for the storage and computational cost as a function of the number of images. We consider two cases for the computational cost: using a single machine with infinite memory, and using multiple machines with a set amount of memory. The former is unrealistic, but gives a sense of how many computations are needed if we had such a supercomputer. The latter gives a more realistic estimate of the practical situation where multiple machines are used in parallel.

We present the definitions of the methods parameters in Table 3.1 and summary of the results in Table 3.2, with details of the derivations in Section 3.4. We note that these calculations are based on minimum theoretical storage and average matching cost scenarios. We also note that we compute the distance between vectors using the dot product, which is equivalent to the Euclidean distance, since we assume the feature vectors are normalized. We do not consider any compression technique that might decrease storage (e.g., run-length encoding).

Parameter	Description	Typical Value
I	no. of images	10^9
s	bytes/feature dim	1
d	feature dimension	128
F	#features/image	1,000
M	# machines	varies
C	main memory/machine	50GB
T_{kdt}	# kd-trees	4
L	length of ranked lists	100
B_{kdt}	# backtracking	250
T_{lsh}	# lsh tables	4
H_{l2}	# hash fun LSH-L2	50
B_{lsh}	# buckets	10^6
H_{sph}	# funcs for LSH-Spherical	5
D	depth of HKM tree	7
k	branching factor of HKM	10
W	# words for BoW	10^6
T_{mh}	# tables for Min-Hash	50
H_{mh}	# hash funs for Min-Hash	1
B_{mh}	# buckets in Min-Hash	10^6

Table 3.1: **Methods Parameter Definitions and Typical Values**. See Section 3.4 and Chapter 2 for more details.

3.3 Theoretical Comparison

3.3.1 Memory and Run Time

Figure 3.1 shows how storage requirements and run time scale with the number of images in the database, assuming they are implemented on one machine with infinite storage and 1 GFLOPS processor. We note the following:

- BoW methods take one order of magnitude less storage than FR methods, due to the fact that we don't need to store the feature vectors.

- Run time for Kd-Trees and Min-Hash grows very slowly with the database size.

Method	Storage (bytes)	Ex. (TB)
Exhaustive	$(sd+4)IF$	132
Kd-Tree	$IF(sd+4+2T_{kdt}+T_{kdt}\frac{\log_2 IF}{8})$	160
LSH-L2	$IF(sd+4+T_{lsh}\frac{\log_2 IF}{8})$	152
LSH-Sim	$IF(sd+4+T_{lsh}\frac{\log_2 IF}{8})$	152
LSH-Orth	$IF(sd+4+T_{lsh}\frac{\log_2 IF}{8})$	152
HKM	$IF(sd+4)+\frac{k^D-1}{k-1}ksd$	132
Inverted File	$Wsd+FI(5+\frac{\log_2 I}{8})$	9
Min-Hash	$Wsd+FI(4+\frac{\log_2 W}{8})+T_{mh}\frac{\log_2 I}{8}$	7

(a) Theoretical storage requirement *(Storage)*

Method	Comp. (FLOP/im)	Ex. (GFLOP/im)
Exhaustive	$F^2I(2d+1)$	256×10^6
Kd-Tree	$B_{kdt}F(2d+1+\log_2 FI)$	0.074
LSH-L2	$FT_{lsh}(H_{l2}(2d+2)+\frac{FI}{B_{lsh}}(2d+1))$	1028
LSH-Sim	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{B_{lsh}}(2d+1))$	1028
LSH-Orth	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{B_{lsh}}(2d+1))$	1028
HKM	$FD(2d+k)+\frac{F^2I}{k^D}\times(2d+1))$	25.7
Inverted File	$FB_{kdt}(2d+1+\log_2 W)+F(2+I)$	1
Min-Hash	$FB_{kdt}(2d+1+\log_2 W)+4FT_{mh}H_{mh}+\frac{T_{mh}I}{MB_{mh}}$	0.07

(b) Theoretical computational cost on a single machine with infinite memory *(Comp.)*

	Parallel (FLOP/im)	Ex. (GFLOP/im)
Exhaustive	$\frac{F^2I}{M}(2d+1)+L(M-1)$	128×10^3
Kd-Tree (IKdt)	$FB_{kdt}(2d+1+\log_2\frac{FI}{M})+L(M-1)$	0.071
Kd-Tree (DKdt)	$FB\log_2\frac{C}{4T_{kdt}}+\frac{FB}{M}(2dB_{kdt}+B_{kdt})+F\log_2\frac{4FIT}{C}+L(\min(FB_{kdt},M)-1)$	0.012
LSH-L2	$FT_{lsh}(H_{l2}(2d+2)+\frac{FI}{MB_{lsh}}(2d+1))+L(M-1)$	85
LSH-Sim	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{MB_{lsh}}(2d+1))+L(M-1)$	85
LSH-Orth	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{MB_{lsh}}(2d+1))+L(M-1)$	85
HKM	$FD(2d+k)+\frac{F^2I}{Mk^D}\times(2d+1)+L(M-1)$	0.021
Inverted File	$FB_{kdt}(2d+1+\log_2 W)+F(2+\frac{IF}{MW})$	0.075
Min-Hash	$FB_{kdt}(2d+1+\log_2 W)+4FT_{mh}H_{mh}+\frac{T_{mh}I}{B_{mh}}$	0.07

(c) Theoretical parallel computational cost with minimum required number of machines M (see Figure 3.2).

Table 3.2: **Theoretical Scaling Properties**. Refer to figure 3.1. Please check Sections 3.2 and Chapter 2.

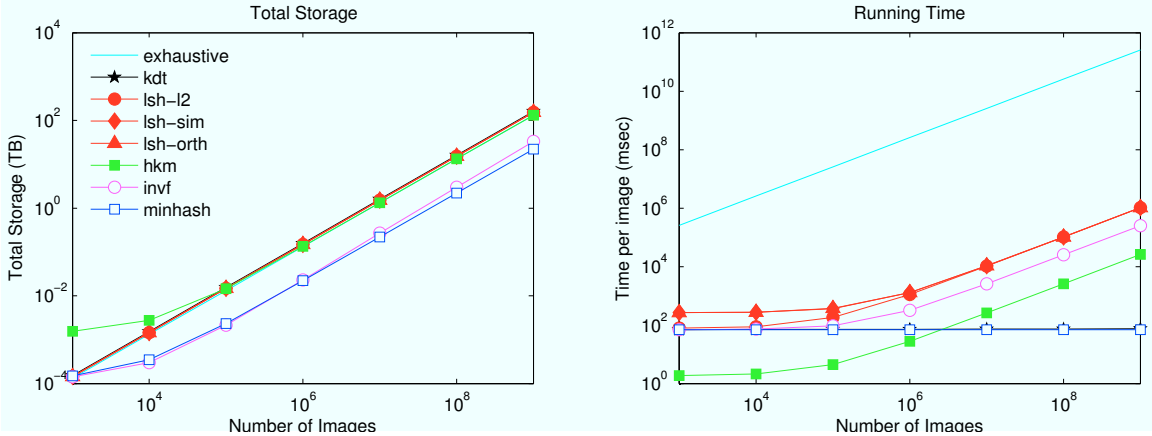


Figure 3.1: **Theoretical Scaling Properties**. Theoretical storage vs. size *(left)*, and run time vs. size *(right)*, assuming single machine with infinite memory. We note that BoW methods (*star* symbols) take an order of magnitude less storage than FR methods. Refer to Tables 3.1-3.2

- Inverted file and LSH methods have asymptotically similar run time. After staying almost constant up to ~ 10^6 images, the theoretical run time increases linearly with the number of images.

3.3.2 Parallelization

The parallelization considered here is the simplest: for every method, we determine the minimum number, M, of machines that can fit the storage required in their main memory, assuming machines with 50 GB of memory. Then we split the images evenly across these machines and each will take a copy of the probe image and search its own share of images. Finally, all the machines will communicate their ranked list of images (of length L) and produce a final list of candidate matches that is further geometrically checked. We call this simple scheme Independent Kd-Trees (IKdt), since the Kd-Trees on the different machines are built independently.

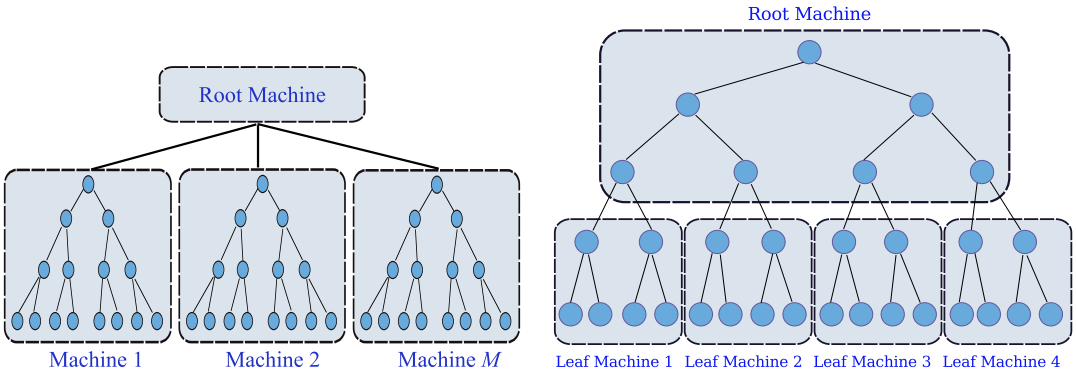


Figure 3.2: **Kd-Tree Parallelizations**. (*Left*) *Independent Kd-Trees (IKdt)*. The images are divided onto the machines, which build independent Kd-Trees for the images they contain. At query time, all the machines are searched in parallel. (*Right*) *Distributed Kd-Trees (DKdt)*. The *root* machine stores the top of the tree, while the *leaf* machines store the leaves of the tree. At query time, the root machine directs features to a small subset of the leaf machines, which leads to higher throughput. See Section 3.3.2

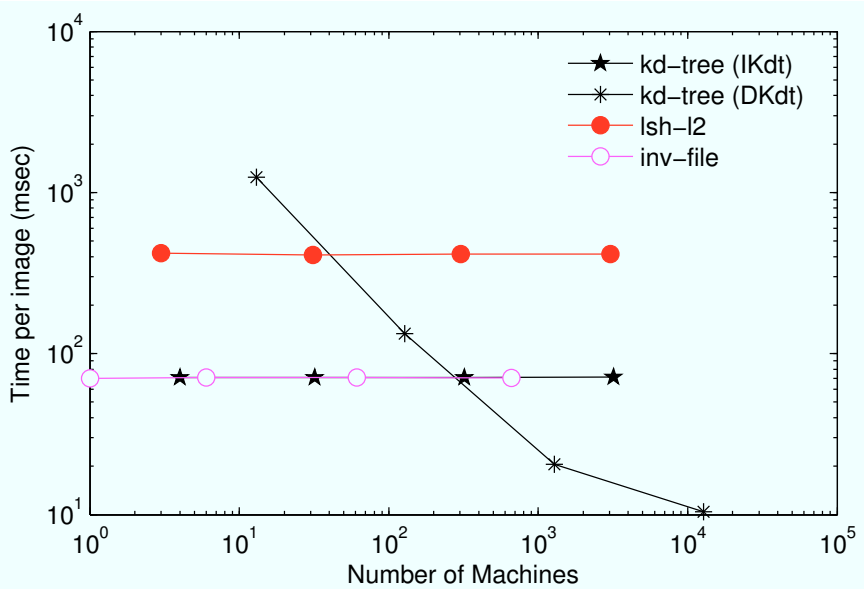


Figure 3.3: **Parallelization Run Time**. Time per image vs. min. no. of machines required M for different dataset sizes. Each point represents a dataset size from 10^6 to 10^9 images, going left to right. IKdt is the Independent Kd-Tree, while DKdt is the Distributed Kd-Tree parallelization. See Section 3.3.2.

More sophisticated parallelization techniques are possible, that can take advantage of the properties of the method at hand. For example, in the case of Kd-trees, one such advanced approach is to store the top part of the Kd-tree on one machine, and divide the bottom subtrees across other machines (see Figure 3.2). We call this approach Distributed Kd-Trees (DKdt), since different parts of the same Kd-Tree are distributed among different machines. For 10^{12} features (10^9 images), we have 40 levels in the Kd-tree, and so we can store up to 30 levels in one *root* machine, and split the bottom 10 levels evenly across the *leaf* machines (see Section 3.4). Given a probe image, we first query the root machine and get the list of branches (and hence subtrees) that we need to search with the backtracking. Then the appropriate leaf machines will process these query features and update their ranked list of images.

The motivations for this distinction between root and leaf machines are:

- Most of the storage for the tree is in the leaves. Therefore we can store most of the top of the tree in a single machine, which will then dispatch jobs to the leaf machines.

- The time spent searching the top of the tree is smaller than that spent searching the lower parts of the tree, so we can gain more speed up by splitting the bottom part of the tree instead of splitting the whole tree.

- Searching the tree will generally not activate all of the leaf machines, i.e., the list of explored nodes will not span all the node machines. Therefore, the root machine can process more features, and the leaf machines can interleave computations of different features simultaneously. This offers more speed up over the first parallelization approach above. So, even though the search time per feature is similar to that in the

first case, interleaving computations will make this much smaller.

This approach has the advantage of significant speed up and better utilization of the machines, since not all the machines will be working on the same query feature at the same time, rather they will be processing different features from the probe image concurrently (see figure 3.3). However, a drawback with this approach is the extra storage requirements, because the lower parts of the trees will generally not store the same features for different trees, and therefore each machine will have to store the features in the subtrees it is holding, and as an upper bound the extra storage will be proportional to the number of Kd-Trees, i.e., we will need to store each feature K times, where K is the number of trees. We provide more details on the implementations of Independent and Distributed Kd-Trees in Chapter 7.

Figure 3.2 shows a schematic of the Distributed Kd-Trees compared to the Independent Kd-Trees, while Figure 3.3 shows the time per image vs. the minimum number of machines M for different dataset sizes from 10^6 to 10^9. We note the following:

- Distributed Kd-Trees provide significant speed ups starting at 10^8 images. It might seem paradoxical that increasing the dataset size decreases the run time, however it makes sense when we notice that adding more machines allows us to interleave processing of more features concurrently, and thus allows faster processing. This however comes at a cost of more storage used (see Figure 3.3).

- We also note that many of the FR methods have parameters that affect the trade off between run time and accuracy, e.g., number of hash functions or bin size for LSH. These parameters have to be tuned for every database size under consideration,

and we should not use the same settings when enlarging the database. This poses a problem for these methods, which have to be re-tuned as the database size changes, as opposed to Kd-Trees which adapt to the current database size and need very little tuning.

3.4 Theoretical Derivations

3.4.1 Exhaustive Search

Description

- Here we do the nearest-neighbor search using exhaustive linear search through the set of database features $\{\{\mathbf{f}_{ij}\}_j\}_i$, i.e., for every feature of the query image $\mathbf{f}_{qj}$, we find the feature that minimizes the Euclidean distance: $n_{qj} = \text{argmin}_{ij}\mathbf{f}_{ij}$, where n_{qj} is the index of the feature and i is the image in the database containing this feature.

- For every database image, we count the number of such nearest neighbors s_i, and for all images with a pre-set amount of nearest neighbors (10 in our experiments), we do the RANSAC affine step to obtain s'_i where s'_i =# inliers of the affine transformation.

Storage

The storage requirements for this method are just the full feature vectors and feature information:

- Feature vectors:

$$S_{fv} = sdIF$$

where s is the number of bytes per feature dimension, d is the dimension of the feature vector, I is the number of images in the database, and F is the average number of features per image.

- Feature information:

$$S_{fi} = 4IF$$

where for every feature we need to store its location (x and y position in the image) in addition to its scale and orientation. We assume we can discretize this information and fit it into 1 byte each.

- Total:

$$S_{exhaustive} = (sd+4)IF$$

Speed

- The main bottleneck is the nearest neighbor search. For every feature, we have to search through the whole feature database to find the nearest feature:

$$T_{nn,ex} = F(2FId + FI) = F^2I(2d+1)$$

where for every query feature we need to perform d multiplications and d additions per database feature, in addition to finding the minimum value among FI values.

Parallelization

- Divide the data required evenly among M machines. Search the M machines simul-

taneously.

- Every machine processes the images it is storing, and produces a final sorted list of geometrically checked images. This list is then broadcast or sent to a central machine, which will merge them into one list and returns the result.

- Speed up: Having more machines here corresponds to linear speed up of the search process (in the number of images). Consider having M machines, then each machine will perform

$$T^{p}_{nn,ex} = \frac{F^2 I}{M}(2d+1) + L(M-1)$$

where the second term is the time to merge M sorted lists of length L.

3.4.2 Kd-Trees

Storage

- We still need to store all the features and their information:

$$S_{ex} = (sd+4)IF$$

- In addition, there is storage for the trees. For a tree with n leaves, there are $2n-1$ total leaves, i.e., n-1 internal nodes and n leaves. Each tree here has IF leaves (features).

Storage for each tree is

$$\begin{aligned} S_{tr} &= 2IF + IF\frac{\lceil \log_2 IF \rceil}{8} \\ S_{tr} &= IF(2 + \frac{\lceil \log_2 IF \rceil}{8}) \end{aligned}$$

where the first term is for the internal nodes where we need to store the dimension and the threshold values (assumed to be discretized to single bytes), and the second term is for the leaves where we need to store the index of the feature it contains

- Total:

$$\begin{aligned} S_{kdt} &= S_{ex} + TS_{tr} \\ S_{kdt} &= IF(sd + 4 + 2T + T\frac{\lceil \log_2 IF \rceil}{8}) \end{aligned}$$

Computational Cost

- Searching through the Kd-Trees involves comparison operations at each level of the tree, in addition to exhaustive search with the final list of features, typically ~ 250, corresponding to the number of backtracking steps:

$$T_{nn,kdt} = FB(2d + 1 + \log_2 FI)$$

where the second term is for finding the minimum among the B distance values, while the third term is the number of comparisons performed.

Parallelization

There are two ways to parallelize Kd-Tree operations. The first is a straightforward extension of the exhaustive search case:

- Divide the data into M machines. For every query feature, search the machines simultaneously.
- Combine the search results as for the exhaustive case, by having a local sorted list of images, and then broadcast these lists to get a global sorted list, which is then geometrically verified.
- Speed up: having more machines here corresponds to logarithmic speed up of the search process. Consider having M machines, then each machine will perform

$$T^{p}_{nn,kdt} = F(2dB + B + B\log_2 \frac{FI}{M}) + L(M-1)$$

where L is the size of the list on each machine, and the second term is the time required to merge M sorted lists of size L.

The second is:

- Instead of having independent Kd-Trees in each machine, there will be a single (or multiple) *root* machine with copies of the top part of the Kd-Trees. For example, with 50 GB of memory available, we can fit 25 billion internal nodes. Therefore, for storing 4 trees, we need to store ~ 6 billion nodes per tree (i.e. 3 billion leaves), which corresponds to storing the top $\log_2 3e9$=30 levels for each of the 4 trees.

- The rest of the trees are then divided to the rest of the M machines (called *leaf* machines) that will also store the feature vectors and information.

- The root machine(s) will get query features and compute an initial list of nodes to explore based on the top of the tree. Then, it will dispatch that feature to all leaf machines that have that part of the tree. Once a leaf machine gets a feature, it will handle backtracking within its own subtree, and updating its ranked list of images.

- Speed up: We have two parts for the running time, one in the root machine and one in the leaf machines:

$$\begin{aligned} T^p_{nn,kdt,r} &= FB\log_2\frac{C}{2\times 2T} \\ T^p_{nn,kdt,l} &= \frac{FB}{M}(2dB+B)+F\log_2\frac{4FIT}{C}+L(\min(FB,M)-1) \end{aligned}$$

where C is the memory capacity of each machine, $T^p_{nn,kdt,r}$ is the time to search the root machine and $T^p_{nn,kdt,l}$ is the time for the leaf machines. The first term is the time taken to search all the F image features concurrently where each feature will be sent to at most B machines (and so we need FB machines to process them in the worst case, where each backtracking step goes to a different leaf machine), the second term is the time to get the minimum, the third term is the time to go down the subtree excluding the top part in the root machine, and the last term is the time to merge the $\min(M,FB)$ sorted lists (equals the number of machines processing the image).

3.4.3 Locality Sensitive Hashing (LSH)

Storage

- We still need to store all the features and their information:

$$S_{ex} = (sd+4)IF$$

- In addition, there is storage for the tables. In each table, we need to store the hash functions, in addition to the indices of all the features available:

$$S_{ta} = IF\frac{\log_2 IF}{8} + S_h$$

where $S_h$is the storage for the hash functions, which depends on the specific hash function used and is negligible compared to the first term.

- Total:

$$\begin{aligned} S_{lsh} &= S_{ex} + TS_{ta} \\ S_{lsh} &= IF(sd+4+T\frac{\lceil \log_2 IF \rceil}{8}) \end{aligned}$$

Computational Cost

Searching through the LSH index involves computing the hash function values and then exhaustively checking the features that share the same hash value. The time can be repre-

sented as

$$T_{nn,lsh} = F(T_h + (2d+1)\frac{FI}{B}T)$$

where the first term is the time to compute the hash functions:

- **L2**: $T_{h,l2} = TH(2d+2)$ where we have H hash functions, and for each we need to compute dot product (one addition and multiplication per dimension) plus another addition and division.
- **Spherical-Simplex**: $T_{h,sim} = TH(2d(d+1)+d) = TH(2d^2+3d)$ where we need to compute product of a $d+1 \times d$ matrix and a vector of d dimensions, and then need to get the closest vertex.
- **Spherical-Orthoplex**: $T_{h,orth} = TH(2d^2+2d) = 2TH(d^2+d)$ where we need to multiply a $d \times d$ matrix with a d vector, and then choose the nearest vertex from the $2d$ choices, where we know that one vertex is just the opposite sign of the other.

The second two terms above are the time to compute exhaustive nearest neighbors, and depends on the size of the LSH bucket, which is not constant, and grows linearly with the number of total features, in addition to it varying greatly for the same number of features. We assume we have a preset number of unique bucket values B, and then the average bucket size would be $\frac{FI}{B}$.

Parallelization

There are two ways to parallelize LSH operations. The first is a straightforward extension of the exhaustive search case:

- Divide the data into M machines. For every query feature, search the machines simultaneously.

- Combine the search results as for the exhaustive case.

- Speed up: having more machines here corresponds to linear speed up. Consider having M machines, then each machine will perform

$$T^{p}_{nn,lsh} \approx F\frac{FI}{MB}T(2d+1)+L(M-1)$$

The second is similar to Kd-Trees:

- Instead of having independent LSH indexes in different machines, there will be a *central* machine that computes all the hash functions, and has a list of which machines store which buckets.

- The rest of the machines will store features that belong to a subset of buckets.

- The central machine(s) will get query features, compute the hash values (buckets), and then dispatch that feature to machines containing that bucket.

- Speed up: We have two parts for the running time, one in the central machine and one in the node machines:

$$\begin{aligned} S_{m2,c} &= FT_{nn,lsh,x} \\ S_{m2,n} &= \frac{F}{T}((2d+1)\frac{FI}{B} \end{aligned}$$

where $S_{m,c}$ is the time to compute the hash values on the central machine, while $S_{m,n}$ is the time taken for the node machines to search that bucket. This method will work if $S_{m2,c}$ is much smaller compared to $S_{m2,n}$. On the other hand, it will be a bottleneck if the two times are comparable.

3.4.4 Hierarchical K-Means (HKM)

Storage

- We still need to store all the features and their information:

$$S_{ex} = (sd+4)IF$$

- In addition, there is storage for the clustering tree. For a tree with depth D and branching factor k, there are k^D leaf nodes, which will store the actual features (in subsets of k). The number of internal nodes is

$$n = \sum_{i=0}^{D-1} k^i = \frac{k^D - 1}{k-1}$$

and for each internal node we need to store the k cluster centers, therefore the total storage is

$$S_{hkm} = \frac{k^D - 1}{k-1} \times ksd + (sd+4)IF$$

Computational Cost

- At each level of the clustering tree, we need to find the nearest cluster, in addition to finding the nearest feature in the leaf node:

$$T_{nn,hkm} = F(D \times (2d+k) + \frac{FI}{k^D} \times (2d+1))$$

where the second term comes from searching the features in the leaf node, assuming each leaf node will have the same share of features.

Parallelization

There are two ways to parallelize HKM operations. The first is a straightforward extension of the exhaustive search case:

- Divide the data into M machines. For every query feature, search the machines simultaneously.
- Combine the search results as for the exhaustive case.
- Speed up: Having more machines here corresponds to less search time. Consider having M machines, then the each machine will perform

$$T^p_{nn,hkm} = FD(2d+k) + \frac{F^2 I}{Mk^D} \times (2d+1) + L(M-1)$$

The second is similar to Kd-trees:

- Instead of having independent HKM trees in different machines, there will be a *central* machine that stores the upper part of the HKM tree (internal nodes).

- The rest of the machines, *node* machines, will store subsets of the leaf nodes and their features.

- The central machine(s) will get query features, go through the top of the tree, and then dispatch features to the appropriate node machines.

- Speed up: We have two parts for the running time, one in the central machine and one in the node machines:

$$\begin{aligned} S_{m2,c} &= D(2d+k) \\ S_{m2,n} &= 2d \times \frac{FI}{k^D} \end{aligned}$$

where $S_{m,c}$ is the time to go through the top of the HKM tree on the central machine, while $S_{m,n}$ is the time taken for the node machines to search the leaf nodes.

3.4.5 Inverted File (IF)

Storage

- We need to store the feature information (e.g., location) to allow the spatial consistency step afterwards:

$$S_{fi} = 4FI$$

- We also need storage for the inverted file:

$$S_{if} = Wsd + FI(\frac{\log_2 I}{8} + 1)$$

where the first term is the storage for the W words, and the second term is storage for the lists of images. We have a total of FI features and for each we need to store the image id and the feature count.

- For large dictionaries of visual words, ~ 1 M visual words, image histograms tend to be binary (i.e. there is one-to-one mapping between features and visual words) so we do not need to store counts:

$$S_{if,bin} = Wsd + FI\frac{\log_2 I}{8}$$

- Total:

$$\begin{aligned} S_t &= S_{if} + S_{fi} \\ &= 4FI + Wsd + FI(\frac{\log_2 I}{8} + 1) \\ &= Wsd + FI(5 + \frac{\log_2 I}{8}) \\ S_{t,bin} &= Wsd + FI(4 + \frac{\log_2 I}{8}) \end{aligned}$$

Computational Cost

There are two components for the run time

- Time for computing the visual words for every feature, which is usually done with

either Kd-Trees or with HKM:

$$\begin{aligned} T_{vw,kdt} &= 2dB + B + B\log_2 W = B(2d + 1 + \log_2 W) \\ T_{vw,hkm} &= D(2d + k) \end{aligned}$$

- Time for searching the inverted file, which is hard to estimate precisely because it depends on the amount of overlap between the query image and the database images. Assuming image features are distributed evenly among the visual words, each query feature will encounter roughly $\frac{IF}{W}$ images per word, and so the time would be

$$T_{if} = 2 + \min(\frac{IF}{W}, I)$$

where the first term is for the normalization of the histogram, and the second is for computing the distance function (e.g., L2 distance) between these two values.

- Total:

$$\begin{aligned} T_{bow,if,kdt} &= F(B(2d + 1 + \log_2 W) + 2 + \min(\frac{IF}{W}, I)) \\ T_{bow,if,hkm} &= F(D(2d + k) + 2 + \min(\frac{IF}{W}, I)) \end{aligned}$$

Parallelization

Parallelization is straightforward:

- Divide the storage among M machines.

- Given a query image, all machines are searched simultaneously to get a ranked list of images in that machine.
- Broadcast the top image per machine, and each machine will compile a list of 100 images.
- The top 100 images are searched for spatial consistency by the machines that are storing them, and the results are again broadcast to get a final list of 100 ranked images
- The results are returned to the user.
- Speed up: The time to search in parallel in the different machines will be smaller, but not as much as the theoretical number above suggests. This is because these calculations assume the worst case run time, which is not usually the case:

$$\begin{aligned} T_{bow,if,kdt} &= F(B(2d+1+\log_2 W)+2+\min(\frac{IF}{MW},\frac{I}{M})) \\ T_{bow,if,hkm} &= F(D(2d+k)+2+\min(\frac{IF}{MW},\frac{I}{M})) \end{aligned}$$

3.4.6 Min-Hash (MH)

Storage

- We need to store the feature information (e.g., location) to allow the spatial consistency step afterwards:

$$S_{fi} = 4FI$$

- We also need storage for LSH tables:

$$S_{lsh,mh} = Wsd + FI\frac{\log_2 W}{8} + T \times (\frac{\log_2 I}{8} + S_h)$$

where the first term is the storage for the W words and the second is for storing the sparse histograms. The third term is storage for the T LSH tables, where for each we need to store indices to the images in the different buckets, plus the negligible S_h to store the hash functions themselves.

- Total:

$$\begin{aligned} S_t &= S_{lsh,mh} + S_{fi} \\ &= Wsd + IF\frac{\log_2 W}{8} + T \times (\frac{\log_2 I}{8} + S_h) + 2FI \\ &= Wsd + FI(4 + \frac{\log_2 W}{8}) + T\frac{\log_2 I}{8} \end{aligned}$$

Computational Cost

There are two components for the run time:

- Time for computing the visual words for every feature, which is usually done with either Kd-Trees or with HKM:

$$\begin{aligned} T_{vw,kdt} &= F(2dB + B + B\log_2 W) = FB(2d + 1 + \log_2 W) \\ T_{vw,hkm} &= FD(2d + k) \end{aligned}$$

- Time for computing the hash functions and the exhaustive search for images within the bucket:

$$T_{mh} = FTH \times (3+1) + T\frac{I}{B}$$

where the first term is for computing the hash function (the permutation is computed as $\pi(x) = ax + b \bmod W$, and so requires three operations), and the second term is for computing the distance for images within that bucket, which is assumed evenly distributed $\frac{I}{B}$ with B unique buckets.

- Total:

$$\begin{aligned} T_{bow,mh,kdt} &= FB(2d+1+\log_2 W) + 4FTH + \frac{TI}{B} \\ T_{bow,mh,hkm} &= FD(2d+k) + 4FTH + \frac{TI}{B} \end{aligned}$$

Parallelization

Parallelization is straightforward:

- Divide the storage among M machines.
- Given a query image, all machines are searched simultaneously to get a ranked list of images in that machine
- Broadcast the top image per machine, and each machine will compile a list of 100 images
- The top 100 images are searched for spatial consistency by the machines that are storing them, and the results are again broadcast to get a final list of 100 ranked

images

- The results are returned to the user
- Speed up: Computations will be faster when using more machines, as the number of images per bucket will go down

$$T_{mh} = TH \times (3+1) + T\frac{I}{BMF}$$

3.5 Summary

This chapter introduced the theoretical comparison of the different methods described in Chapter 2. We provided theoretical estimates of the memory requirements and computational cost of these methods, both on a single machine and on multiple machines. Furthermore, we also described approaches for parallelizing these methods, including a novel way of parallelizing Kd-Trees, Distributed Kd-Trees, which provides much higher throughput. The main result of the comparison here is that Kd-Tree has competitive run time with BoW methods and also has advanced parallelization schemes that provide higher throughput, while requiring an order of magnitude more storage. In the next chapter, we will describe the experimental comparison of these methods, where we measure their run time and recognition performance. Chapter 7 will explore the implementation details for the two parallelizations of Kd-Trees discussed here.

Chapter 4

Experimental Comparison

4.1 Introduction

Chapter 3 detailed the theoretical comparison between the methods introduced in Chapter 2, providing estimates of the storage requirements and the computational cost of these methods for up to 1 billion images. In this chapter we present the experimental comparison between these methods on four real-world datasets, for up to 400 thousand images. In Chapter 7 we run large-scale experiments with Kd-Trees with up to 100 million images.

Section 4.2 describes the datasets used in this thesis. Section 4.3 presents the details of the experimental setup. Section 4.4 provides details and discussions of the experimental comparison. Finally, Section 4.5 gives details of the parameter tuning for the different methods.

4.2 Datasets

We use four different datasets in our experiments, each with different types of statistics. Each dataset contains two types of images (see Figure 4.1):

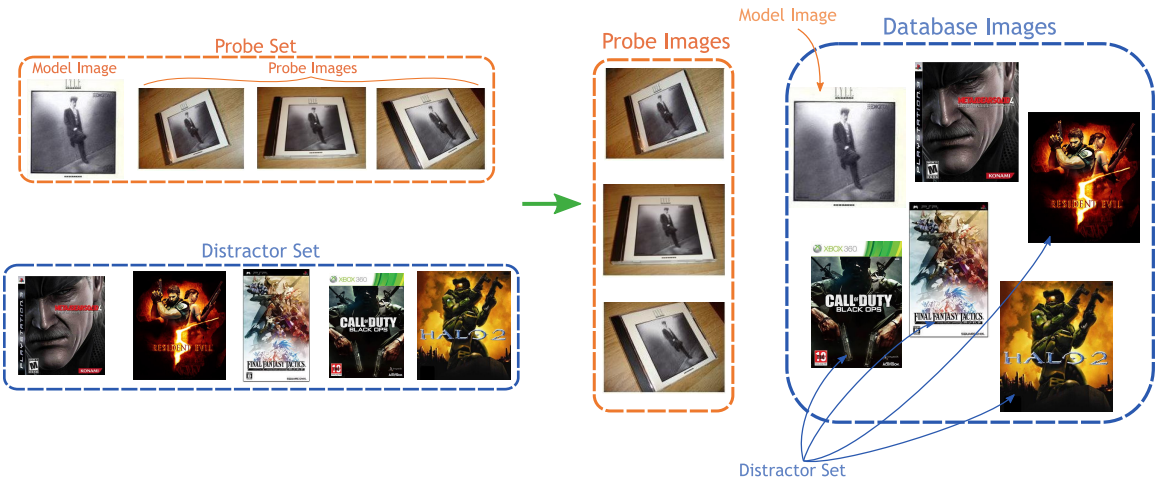


Figure 4.1: **Probe and Distractor Sets**. See Figures 4.2–4.3 and Section 4.2.

1. **Probe**: A set of labeled images used for benchmarking purposes. This has two types of images per object:

   (a) **Model Image**: represents the ground truth image to be retrieved for that object

   (b) **Probe Images**: used for querying the database, representing the object in the model image from different view points, lighting conditions, scales, etc.

2. **Distractors**: A set of images that constitute the bulk of the database to be searched. They are unrelated to the images used for probing, albeit with similar statistics. It represents the clutter, or distractor images that the algorithm must go through in order to find the right image. In the actual setting, this would include all the objects of interest, e.g., book covers, CD covers, etc.

4.2.1 Probe Sets

- **P1: CD Covers.** A set of 5×97=485 images of CD/DVD covers. The model images come from *freecovers.net* while the probe images come from the dataset used in [27]. This was also used in [4].

- **P2: Pasadena-Buildings.** A set of 6×125=750 images of buildings around Pasadena, CA from [4]. The images were taken on two different days, at different times, from different viewpoints. The model image is "image2" (frontal view of the building in the afternoon).

- **P3: ALOI.** A set of 9×80=640 3D objects images from the ALOI collection [19] with different illuminations and view points. We use the first 80 objects, with the frontal view of each object as the model image, and four orientations and four illuminations as the probe images.

- **P4: INRIA-Holidays.** A set of 957 images, which forms a subset of images from [20], with groups of at least 3 images. There are 233 model images and 724 probe images. The first image in each group is the model image, and the rest are the probe images. We used only this subset so that we have at least two probe images per object.

Figure 4.2 shows some examples of images from these distractor sets. Table 4.1 summarizes the properties of these probe sets.

	total	#model	#probe
P1	485	97	388
P2	750	125	525
P3	720	80	640
P4	957	233	724

Table 4.1: **Probe Sets**. Each row depicts the number of model images, the number of probe images, and total number of images. See Section 4.2.

4.2.2 Distractor Datasets

- **D1: Caltech-Covers.** A set of ~ 100 K images of CD/DVD covers used in [4].
- **D2: Flickr-Buildings.** A set of ~ 1 M images of buildings collected from www.flickr.com
- **D3: Image-net.** A set of ~ 400 K images of "objects" collected from www.image-net.org, specifically images under synsets: instrument, furniture, and tools.
- **D4: Flickr-Geo.** A set of ~ 1 M geo-tagged images collected from www.flickr.com

Figure 4.3 shows some examples of images from these distractor sets.

4.3 Experimental Details

4.3.1 Setup

We used four different evaluation scenarios, where in each we use a specific distractor/probe set pair. Table 4.2 displays a list of scenarios used. Evaluation was done by increasing the size of the dataset from 100 to 1 K, 10 K, 50 K, 100 K, up to 400 K. For each such size, we include all the model images to the specified number of distractor images e.g. for 1k

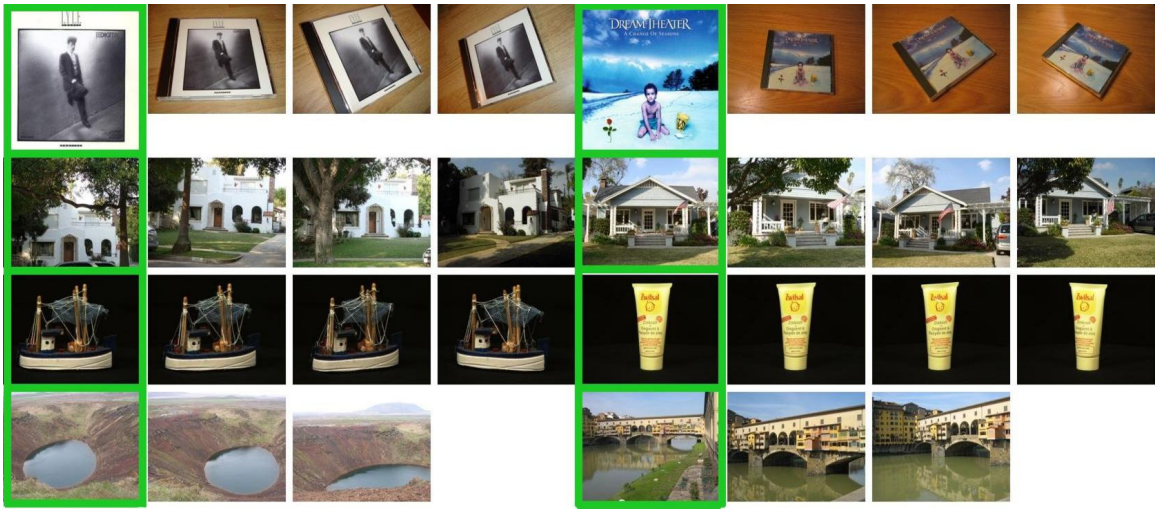


Figure 4.2: **Example Probe Images**. Each row depicts a different set: P1, P2, P3, and P4, respectively. Each row shows two model images and 2 or 3 of its probe images. Details in Table 4.1. See Section 4.2 and Figure 4.1.

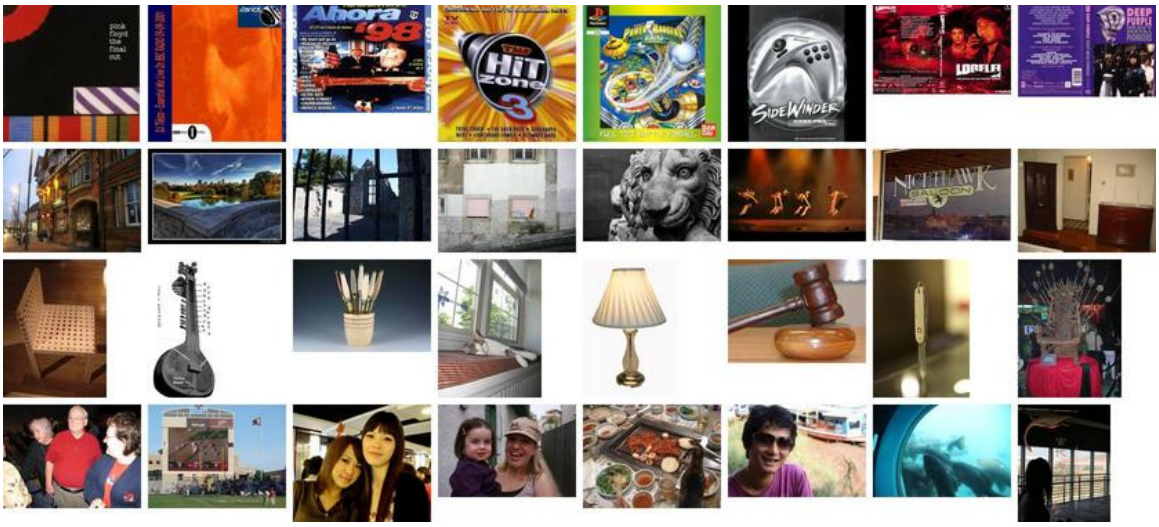


Figure 4.3: **Example Distractor Images**. Each row depicts a different set: D1, D2, D3, and D4, respectively. See Section 4.2 and Figure 4.1.

images, we have 1000 images from the distractor set in addition to all images in the probe set.

Performance is measured as the percentage of probe images correctly matched to their ground truth model image, i.e., whether the correct model image is the first ranked image returned (precision @1). In addition, we measure performance before and after the geometric consistency check, which is done using the RANSAC algorithm [18] to fit an affine transformation between the probe and the model images. Ranking before the geometric check is based on the number of nearest features in the image, while ranking after is based on the number of inliers of the affine transform. We only consider the first ranked image because we want to make sure that the algorithm returns the correct result in the top spot as often as possible. That is also similar to Google Goggles' operation, which returns only one result.

We want to emphasize the difference between the setup used here and the setup used in other "image retrieval"-like systems [20, 13, 30]. In our setup, we have only ONE correct ground truth image to be retrieved and several probe images, while in the other setting there are a number of images that are considered correct retrievals. Our setup is motivated by the application under consideration, for example indexing in a database of book covers. This database would have one canonical image for each book, and the goal is to correctly identify the book in the probe image.

We use SIFT [24] feature descriptors with Hessian affine [25] feature detectors. We used the binary available from tinyurl.com/vgg123. All experiments were performed on machines with an Intel dual Quad-Core Xeon E5420 2.5 GHz processor and 32 GB of

Scenario	Probe	Distractor
1	P1	D1
2	P2	D2
3	P3	D3
4	P4	D4

Table 4.2: **Evaluation Scenarios**. Each evaluation scenario uses one distractor set and one probe set from the sets defined in Section 4.2.

RAM. We implemented all the algorithms using a mix of Matlab and Mex/C++ scripts. All the software is publicly available online at http://vision.caltech.edu/malaa/software/.

4.3.2 Parameter Tuning

All of the methods we compare have different parameters that affect their run time, storage, and recognition performance. We performed parameter tuning in two steps: First, using a subset of the probe images from Scenario 1, we narrowed down the parameter space into set of promising parameters; Second, we ran experiments using this set of parameters for the four scenarios with sizes from 100 up to 10 K images. Based on these results, we chose the settings that made most sense in terms of their recognition performance, run time, and the available memory resources. Specifically, for Kd-Tree, we could run experiments on 100 K images using only 1 tree, although using more trees with more backtracking steps yielded better recognition results (see Figure 4.9 in Section 4.5.1). Please see Section 4.5 for more details. Table 4.3 summarizes the parameters chosen for the full benchmark.

	Parameters
Kd-Trees	T= 1 tree, B =100 backtracking steps
LSH-L2	T =4 tables, H =25 hash functions, bin size w =0.25
LSH-Sim	T =4 tables, H =5 hash functions
LSH-Orth	T =4 tables, H =5 hash functions
HKM	tree depth $D = 5$, branching factor $k = 10$
Inverted File	tf-idf weighting, l_2 normalization, cos distance
	raw histograms, $l_1$normalization, l_1 distance
	binary histograms, l_2 normalization, cos distance
Min-Hash	$T = 100$ tables, $H = 1$ hash function

Table 4.3: **Experimental Comparison Parameter Settings**. The chosen methods parameters used for the full experiments. See Sections 4.3 and 4.5 for details.

4.4 Experimental Results and Discussion

Figure 4.7 shows the recognition performance for different dataset sizes, before and after the geometric consistency check. Figure 4.8 shows recognition performance vs. time for one dataset size (10 K images), before and after geometric checks. We note the following:

- Scenario 2 (Pasadena homes in the probe and Flickr buildings as distractors) is the most challenging. Other scenarios are easier. Across scenarios we find broadly consistent rankings of the different algorithms. The rankings are easier to see in Scenario 2 and therefore we report results for Scenario 2 in Figures 4.4 and 4.5 for more convenient evaluation of the algorithms. Full results are in Figures 4.7 and 4.8.

- Full Representation (FR) methods based on (approximately) matching the local features provide superior recognition performance to Bag of Words (BoW) methods. See Figure 4.4 (left).

- BoW methods provide nearly constant search time compared to linear increase in case of most FR methods, with the exception of Kd-Trees (before exhausting all

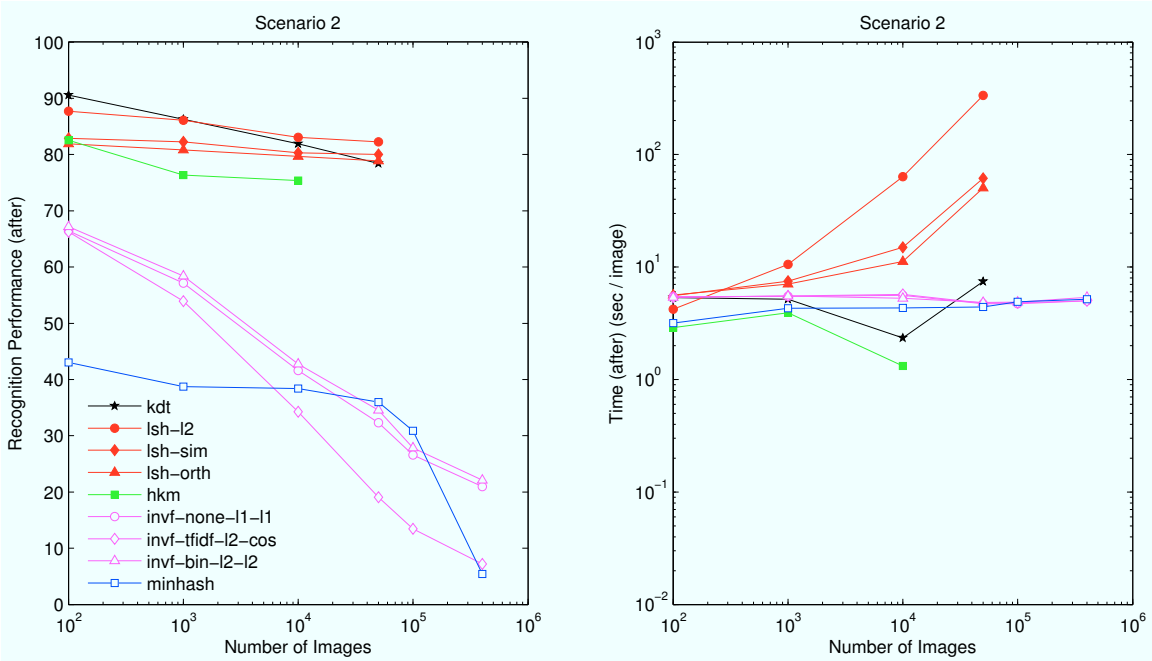


Figure 4.4: **Recognition Performance and Time Vs. Dataset Size**. This figure shows the recognition performance (after geometric consistency checks) and total run time per image as a function of the distractor set size. It only shows results for Scenario 2 (see Section 4.2 and Table 4.2). For more detailed results, please check Figure 4.8.

the main memory, see the big jump in Figure 4.7 for 50 K images) which provide logarithmic increase in search time. See Figure 4.7 (third row).

- FR methods take an order of magnitude more memory than BoW methods, e.g., we can easily fit up to 400 K images for BoW while for some scenarios we can only fit up to 50 K images for some FR methods. See Figure 3.1.

- FR methods pose a trade off between recognition rate and search time. In particular, for Scenarios 2 and 4, we notice that LSH methods are generally better than Kd-Trees in terms of recognition rate but inferior in terms of search time, which rises sharply with database size. This suggests that we can trade off extra search time for better recognition rate. See Figures 4.4 and 4.7.

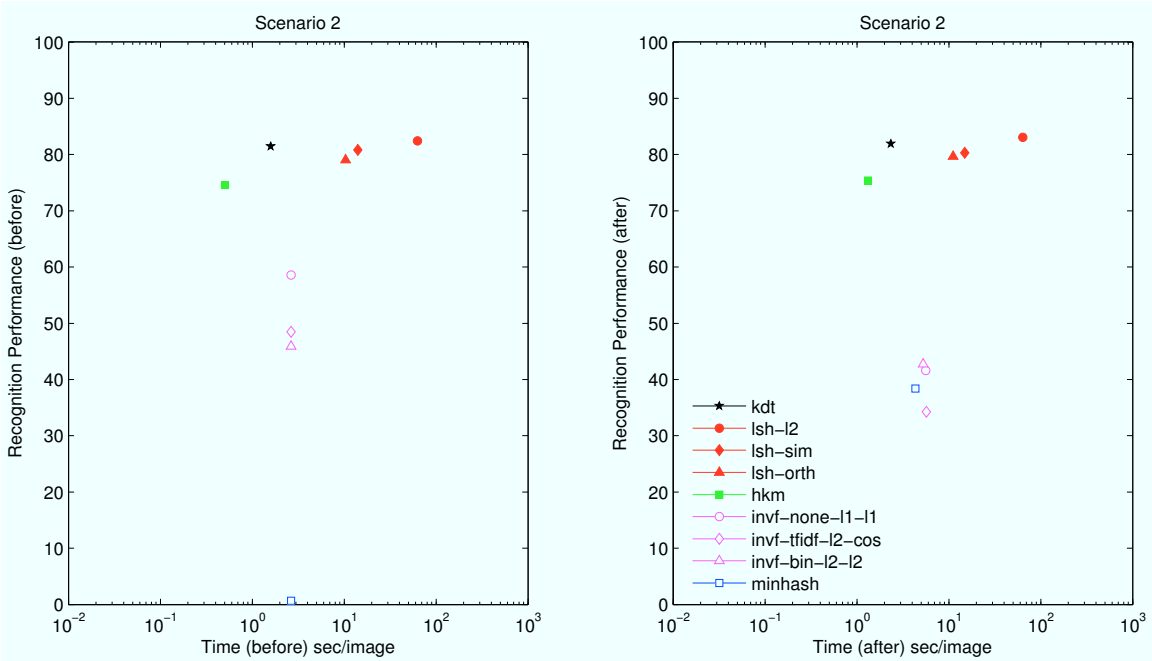


Figure 4.5: **Recognition Performance Vs. Time**. Left column shows recognition performance vs. total matching time per image before the geometric step, while right column shows performance vs. time after the geometric step for dataset size of 10 K images. It only shows results for Scenario 2; see Table 4.2. For more detailed results, please check Figure 4.8.

- Spherical LSH methods provide comparable recognition rate to LSH-L2 methods, but they provide better search time. See Figures 4.4 and 4.7.

- If we focus on BoW methods, using the combination of l_1 normalization with l_1 distance in the inverted file method provides better performance than the standard way of using tf-idf weighting with l_2 normalization and dot product. Using binary histograms also outperforms the standard tf-idf scheme, as reported in [21]. See Figure 4.4.

- Kd-Trees provide the best overall trade off between recognition performance and computational cost. We notice that its run time grows very slowly with the number

of images while giving excellent performance. Moreover, with smart and more complicated parallelization schemes, like the one described in Section 3.3.2 and Figure 3.3, we can have significant speed up when running on multiple machines. The only drawback, which is shared with all FR methods, is the larger storage requirements. We also note that although the benchmark results were obtained using only 1 tree with 100 backtracking steps, the results were significantly better than BoW methods. Using more trees with more backtracking steps can yield even better results at the expense of more memory and computational costs (see Figure 4.9).

Figure 4.6 shows the theoretical running time (from Figure 3.1) vs. the experimental running time for Scenario 3 (from Figure 4.7). We note that the experimental results generally follow the theoretical estimates. The notable exception is that Spherical LSH experimentally has less run time than LSH-L2. The reason for that is the actual distribution of the features on the LSH buckets. In the theoretical estimates we assumed a uniform distribution of features over all buckets, which is not the case practically. For example, the Spherical LSH had a mean bucket size of 20 features with ~ 1 M full buckets, compared to 50 features per bucket with ~ 500 K full buckets for the LSH-L2. We also note that the experimental times are about 4–5 times larger than the theoretical estimates. This is expected, as the estimates do not factor in code inefficiencies in addition to other overhead like function calls, OS calls, memory allocation, cache effects, etc.

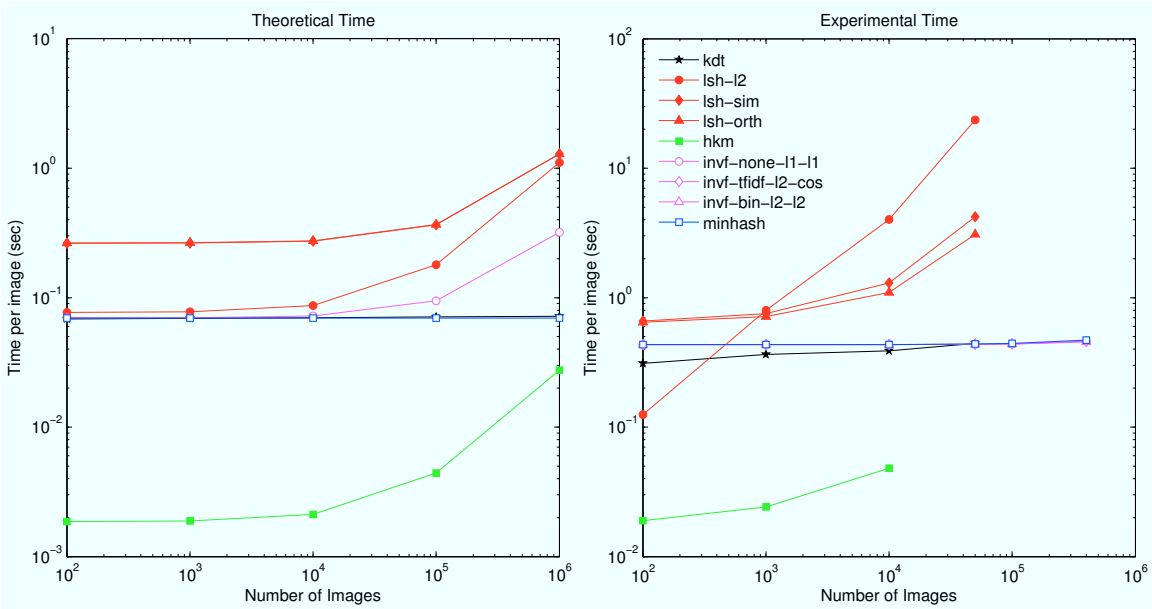


Figure 4.6: **Run Time: Theory Vs. Practice**. The left plot shows theoretical running time estimates (see Fig. 3.1) while the right shows experimental running times before geometric verification step for Scenario 3 (see Figure 4.7).

4.5 Parameter Tuning Details

Some of the algorithms have a large number of parameter combinations that greatly affect performance in terms of trading off speed and recognition performance. In order to get a quick feel for which combinations of parameters to try out, we performed some quick experiments to estimate the performance of various combinations, in terms of speed and recognition rate. The quick tuning is done with a random subset of 100 probe images (out of 388 probe images) from Scenario 1. The matching rate is the percentage of nearest neighbor features that are actually in the ground truth model image. Based on these quick assessments, we chose certain combinations and performed further experiments on all the scenarios with all the probe images.

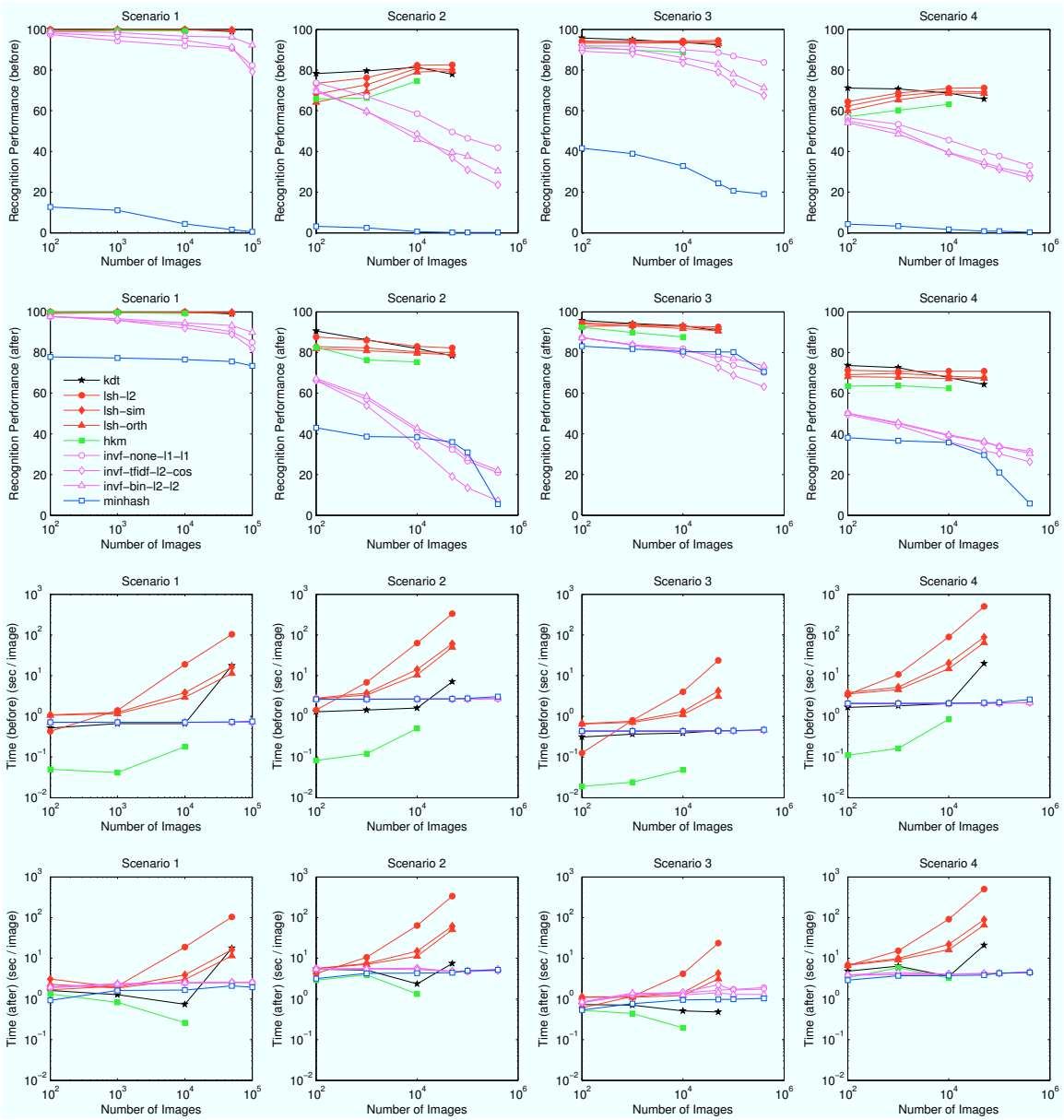


Figure 4.7: **Recognition Performance and Time Vs. Dataset Size** (Full). First two rows show recognition performance before and after the geometric step. Lower two rows show total processing time per image before and after the geometric step. Every column represents a different experimental scenario, see Tables 4.2 and 4.3.

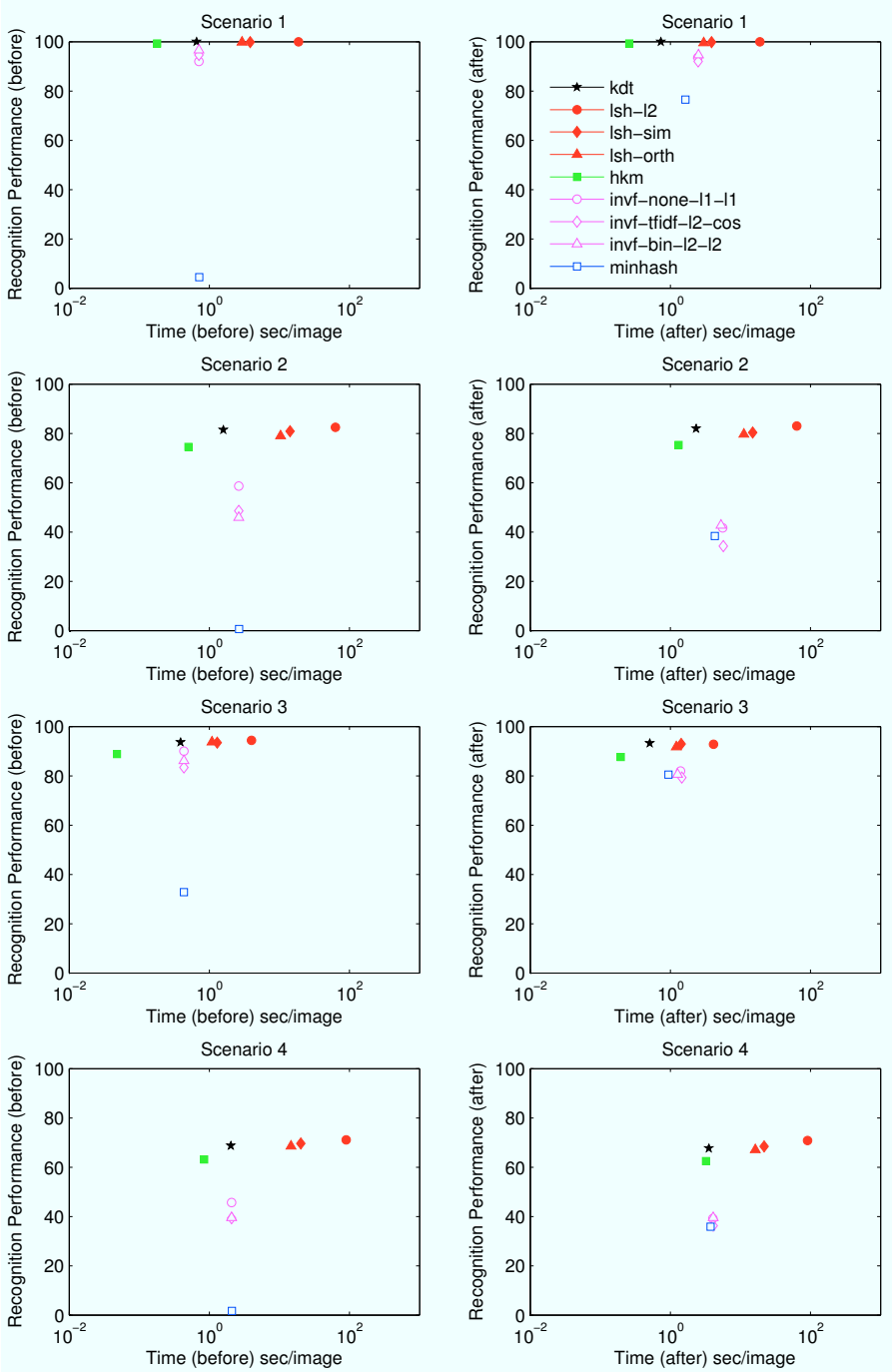


Figure 4.8: **Recognition Performance Vs. Time** (Full). Left column shows recognition performance vs. total matching time per image before the geometric step, while right column shows performance vs. time after the geometric step for dataset size of 10 K images. Each row corresponds to a different scenario (see Table 4.2).

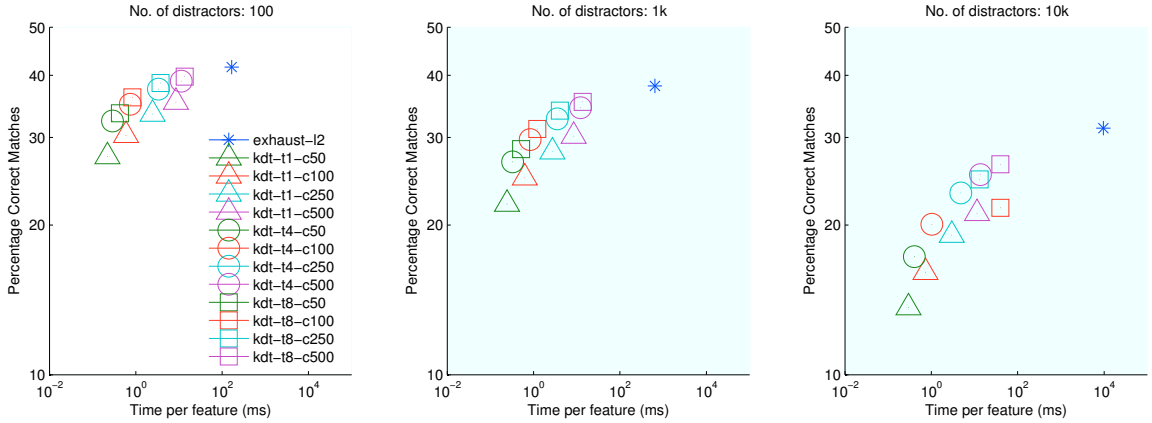
4.5.1 Kd-Tree

For Kd-Trees we have two parameters: T the number of trees and B the number of backtracking steps. Figure 4.9(a) shows results for trying 1, 4, and 8 trees with 50, 100, 250, and 500 backtracking steps. Using 8 trees takes much more memory with minimal gain in matching rate, while using 500 backtracking steps takes too much time. Based on the plot, we decided to go forward with using 1 and 4 trees with 100 and 250 backtracking steps. Figure 4.9(b) shows results for running these parameters on the four scenarios. We chose the combination of $\{T,B\} = \{1,100\}$ as the representative from this method in later comparisons. We note that using more trees with more backtracking steps will have better recognition performance, at the expense of more memory and computation (see Figure 4.9(b)). This represents a trade off that should be decided based on the memory and computational resources available.

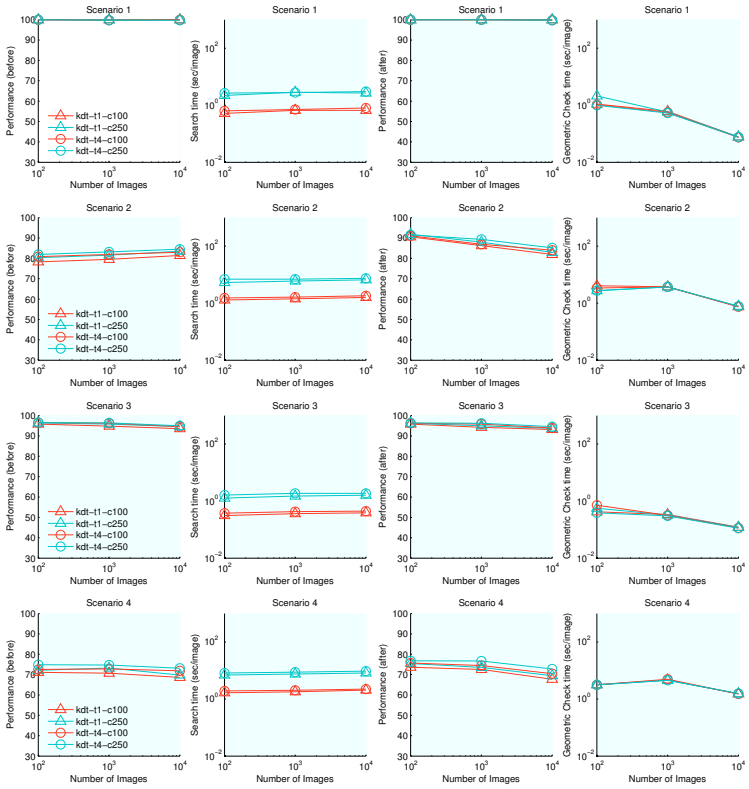
4.5.2 Locality Sensitive Hashing

4.5.2.1 LSH-L2

Here we have three parameters: T the number of tables, H the number of hash functions, and w the bin size. We tried different combinations of using 1, 4, or 8 tables, 5, 10, 25, or 100 hash functions, and bin size of 0.1, 0.25, 0.5, or 1. Based on results in Figure 4.10(a), we decided to use $(T,H,w) = \{4,10,0.1\}$, $\{4,25,0.25\}$, $\{4,50,0.5\}$, and $\{4,100,1\}$ for the rest of the experiments, which are the points at the knee of the plot, and represent the most promising combinations. Those results are shown in Figure 4.10(b). We notice that they provide very similar recognition performance and searching time. We chose the



(a) **Quick Tuning for Kd-Tree**. The plot shows the percentage of correctly matched features as a function of time for different dataset sizes for different combinations of Kd-Trees. Based on these results, we chose $T = 1$ and 4 trees with B =100 and 250 backtracking steps. See (b) below.



(b) **Full Tuning for Kd-Tree**. Results for selected Kdt parameters from (a) above on the 4 scenarios. Based on these results, we chose $T = 1$ and $B = 100$.

Figure 4.9: **Kdt Parameter Tuning.**

combination $(T, H, w) = \{4, 25, 0.25\}$ as the representative from this method in later comparisons.

4.5.2.2 LSH Spherical Simplex

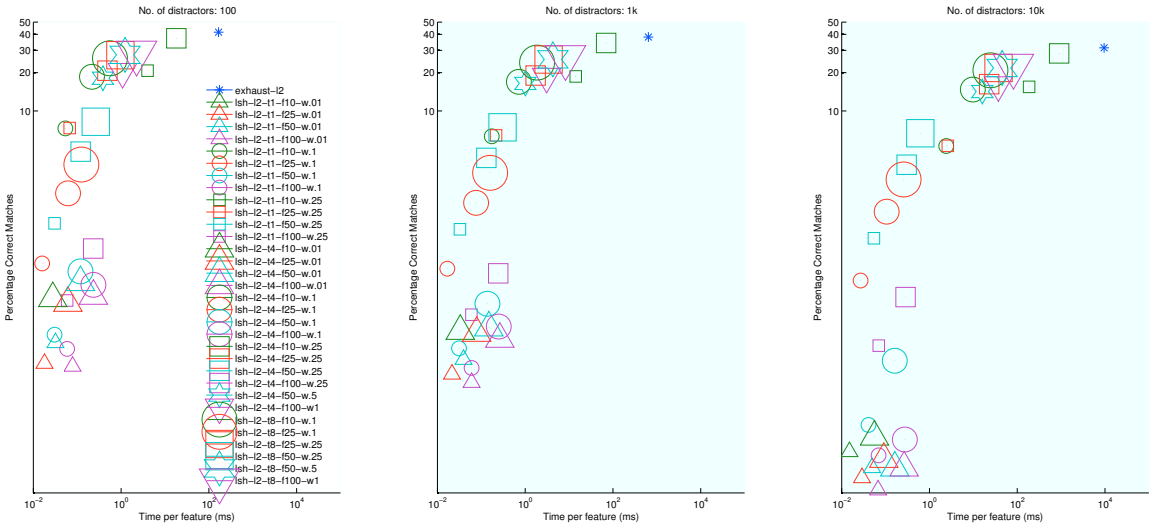
Here we have two parameters: T the number of tables, and H the number of hash functions. We tried using 1 and 4 tables with 3, 5, or 7 hash functions. Results are in Figure 4.11(a). We then decided to use 4 tables with 5 and 7 hash functions, whose results are in Figure 4.11(b). We notice that the recognition performance of using 5 functions is consistently better, while having similar search time. We hence chose the combination $\{T, H\} = \{4, 5\}$ as the representative from this method in later comparisons.

4.5.2.3 LSH Spherical Orthoplex

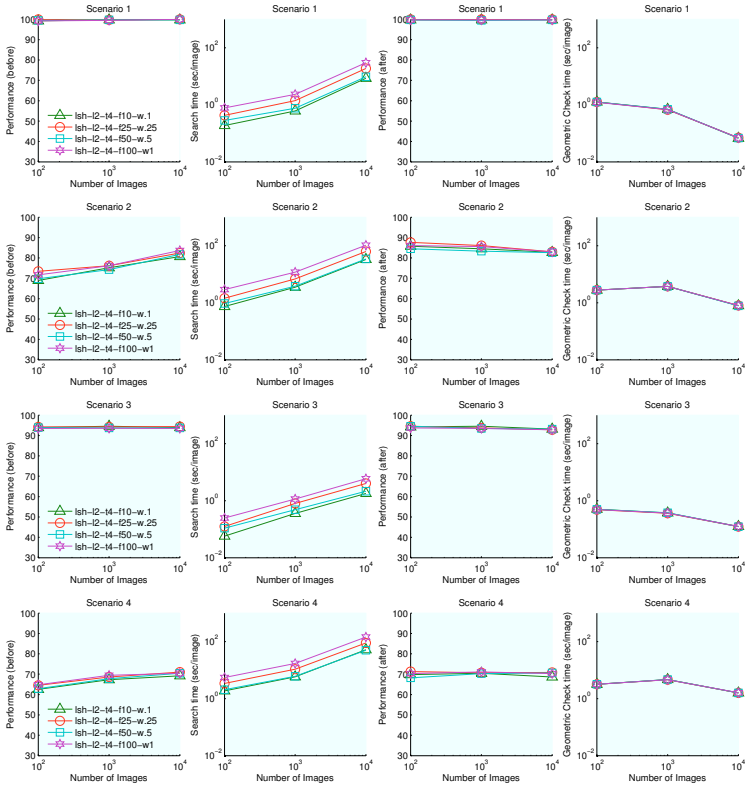
Here we have two parameters: T the number of tables, and H the number of hash functions. We tried using 1 and 4 tables with 3, 5, or 7 hash functions. Results are in Figure 4.12(a). We then decided to use 4 tables with 5 and 7 hash functions, whose results are in Figure 4.12(b). Again, we notice that the recognition performance of using 5 functions is consistently better, while having similar search time. We hence chose the combination $\{T, H\} = \{4, 5\}$ as the representative from this method in later comparisons.

4.5.3 Hierarchical K-Means

HKM has two parameters: D the depth of the tree and k the branching factor. We tried depths of 5, 6, and 7 with branching factor of 10. Results for quick tuning are in Figure 4.13(a), and full results are in Figure 4.13(b). From the quick tuning, all the combinations

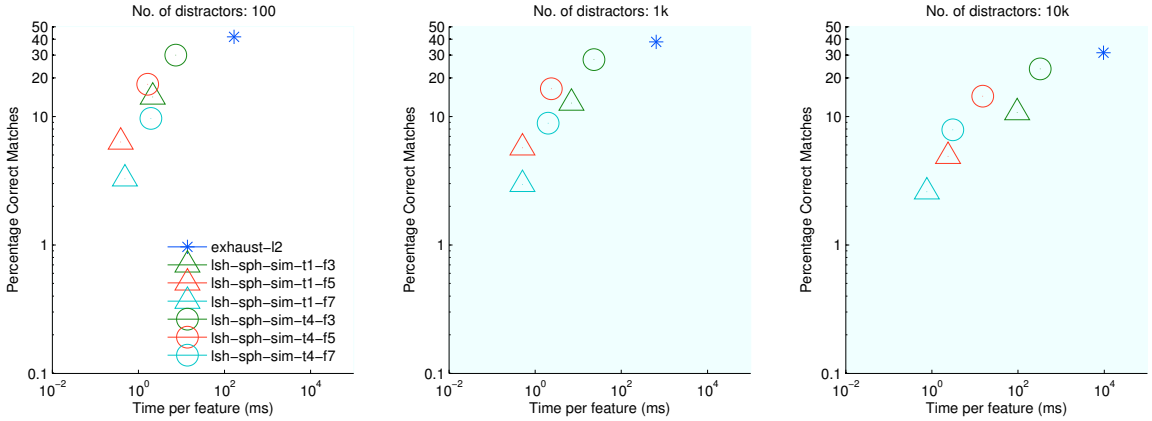


(a) **Quick Tuning for LSH-L2**. The plot shows the percentage of correctly matched features as a function of time for different dataset sizes for different combinations of LSH-L2. The parameters are T =1, 4, and 8 tables, H =10, 25, 50, 100 hash functions, and w =0.1, 0.25, 0.5, and 1 bin size. Based on these results, we used $(T,H,w) = \{4,10,0.1\}$, $\{4,25,0.25\}$, $\{4,50,0.5\}$, and $\{4,100,1\}$ for the full tuning in (b) below.

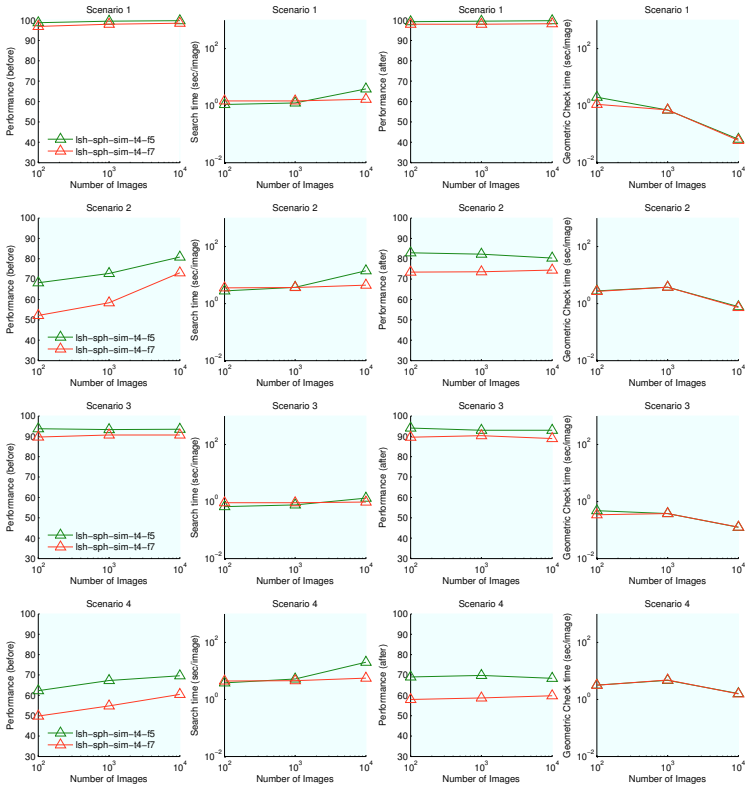


(b) **Full Tuning for LSH-L2**. The plot shows tuning results on the four scenarios for the parameters chosen from (a) above. Based on these results, we chose $(T,H,w) = \{4,25,0.25\}$ as the representative combination.

Figure 4.10: **LSH-L2 Parameter Tuning.**

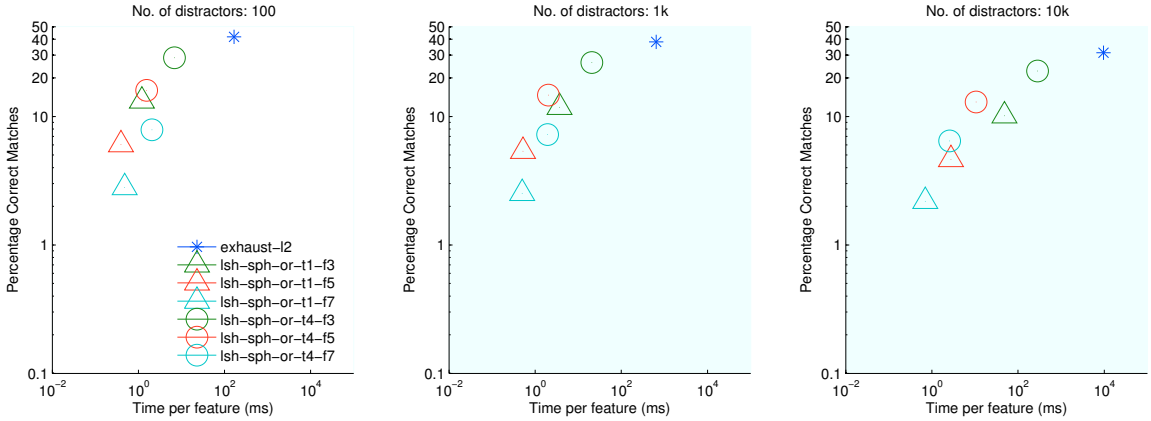


(a) **Quick Tuning for LSH-Sim.** The plot shows the percentage of correctly matched features as a function of time for different dataset sizes for different combinations of LSH-Sim. The parameters are T =1 and 4 tables, and H =3, 5, and 7 hash functions. Based on these results, we used $(T,H) = \{4,5\}$ and $\{4,7\}$, for the full tuning in (b) below.

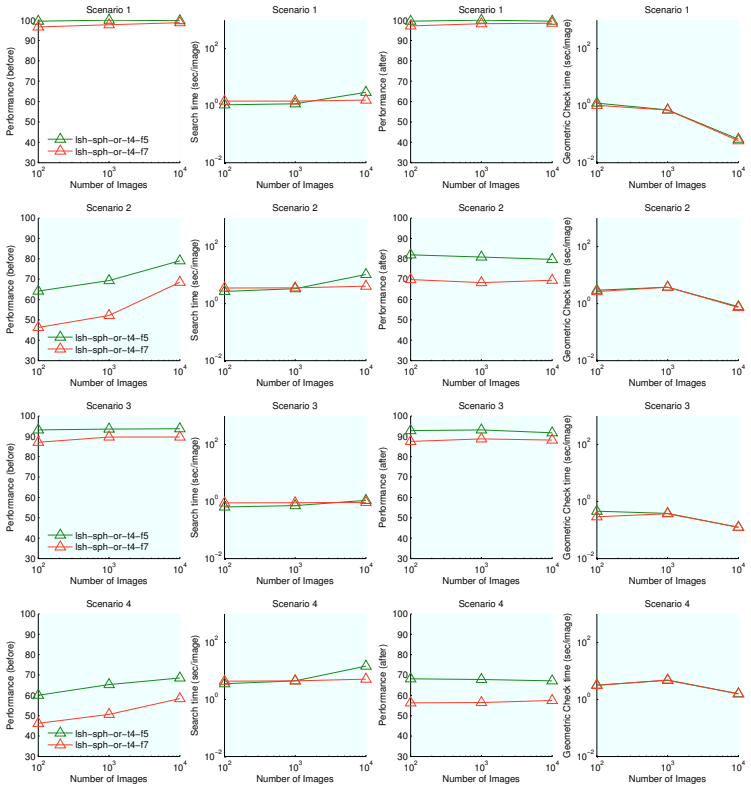


(b) **Full Tuning for LSH-Sim**. The plot shows tuning results on the four scenarios for the parameters chosen from (a) above. Based on the results, we chose $T = 4$ tables and $H = 5$ hash functions per table.

Figure 4.11: **LSH-Sim Parameter Tuning**



(a) **Quick Tuning for LSH-Orth.** The plot shows the percentage of correctly matched features as a function of time for different dataset sizes for different combinations of LSH-Orth. The parameters are T =1 and 4 tables, and H =3, 5, and 7 hash functions. Based on these results, we used $(T,H) = \{4,5\}$ and $\{4,7\}$, for the full tuning in (b) below.



(b) **Full Tuning for LSH-Orth**. The plot shows tuning results on the four scenarios for the parameters chosen from (a) above. Based on the results, we chose $T = 4$ tables and $H = 5$ hash functions per table.

Figure 4.12: **LSH-Orth Parameter Tuning**

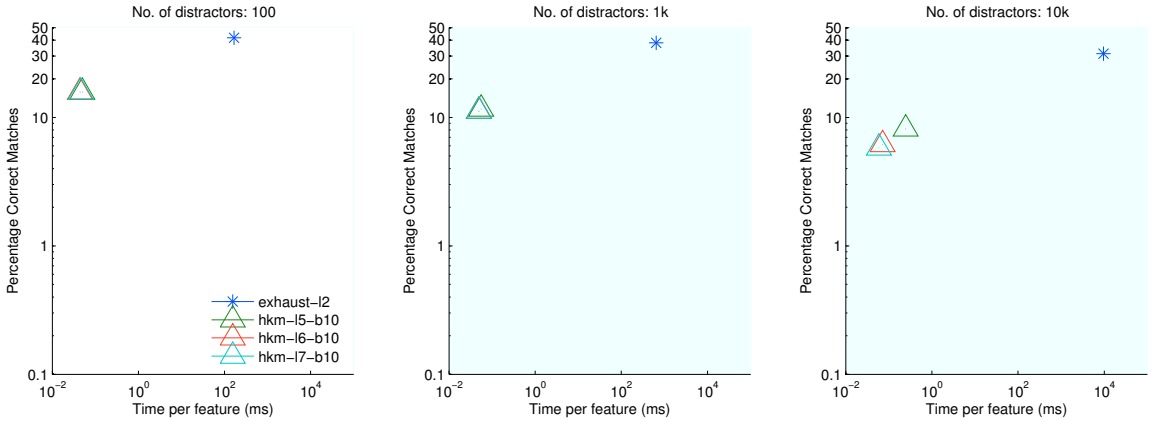
are quite similar, so we ran those on the four scenarios. We notice that they have similar performance, with using a depth of 5 yielding slightly better performance. We chose this combination $\{D,k\} = \{5,10\}$ as the representative of the method in later comparisons.

4.5.4 Inverted File

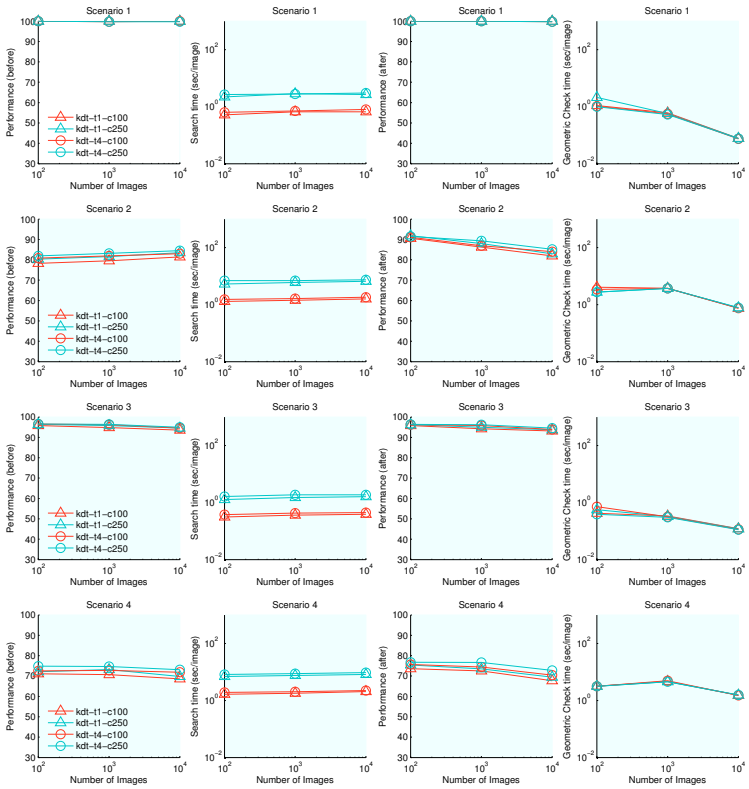
We have several parameters:

- The number of words in the dictionary. We try using 10^4, 10^5, and 10^6 words.
- The method of generating the dictionary, and we compare two popular ways: a vocabulary tree using Hierarchical K-Means procedure (HKM) [27], and a flat dictionary built using Approximate K-Means (AKM) employing a set of random Kd-trees to perform the nearest-neighbor step [30]. The dictionaries are built using features from 10^5 images from each distractor set.
- Weighting the histogram, normalization of the histogram, and distance function. We used the following combinations: $\{w,n,d\}$={none,$l1$,$l1$}, {none,$l2$,$l2$}, {tf-idf,$l2$,cos}, {bin,$l1$,$l1$}, and {bin,$l2$,$l2$}.

Results of tuning are shown in Figure 4.14. From these results, we decided to go ahead with using three combinations: {tf-idf,$l2$,cos}, {bin,$l2$,$l2$}, and {none,$l1$,$l1$}. The first is the standard way in the literature to use inverted file search. The second has also been shown to work competitively with larger dictionaries [21], while the third is a novel combination. We note that increasing the number of words in the dictionary enhances performance, and that HKM is a bit worse than AKM, so we go ahead and use the AKM dictionary with 1



(a) **Quick Tuning for HKM**. The plot shows the percentage of correctly matched features as a function of time for different dataset sizes for different combinations of HKM. The parameters are D =5, 6, and 7 levels, and k =10 branches. We used the three settings for the full tuning in (b) below.



(b) **Full Tuning for HKM**. The plot shows tuning results on the four scenarios for the parameters chosen from (a) above. Based on the results, we chose $D = 5$ levels and $k = 10$ branches.

Figure 4.13: **Hierarchical K-Means Parameter Tuning**

M words. Figure 4.15 shows results of these chosen parameters on the four scenarios. We notice:

- Using the combination {none,$l1$,$l1$} is comparable or outperforms the other two combinations on most scenarios, especially before the geometric step.
- Using the binary histograms generally outperforms the standard way of tf-idf weighting the histograms.

4.5.5 Min-Hash

Here we have similar parameters pertaining to the dictionary: number of words and how the dictionary is generated. In addition, we have two more parameters: T the number of hash tables and H the number of hash functions. We try 1, 5, 25, and 100 tables with 1, 2, and 3 hash functions. Based on the results in Figure 4.16, which show these combinations with different dictionary sizes and dictionary generation, we chose to go ahead with AKM dictionary with 1 M words and using 25 and 100 tables with 1 hash function. Figure 4.17 shows the results of these combinations on the four scenarios, after which we chose $(T,H) = (100,1)$ for the benchmark. We notice the following:

- Min-Hash performance is generally worse than Inverted File. This is expected, as it is just an approximation that works best for near identical images, and not for images with large distortions [12, 13].
- Geometric consistency checks are crucial for Min-Hash, as the first step provides very poor performance.

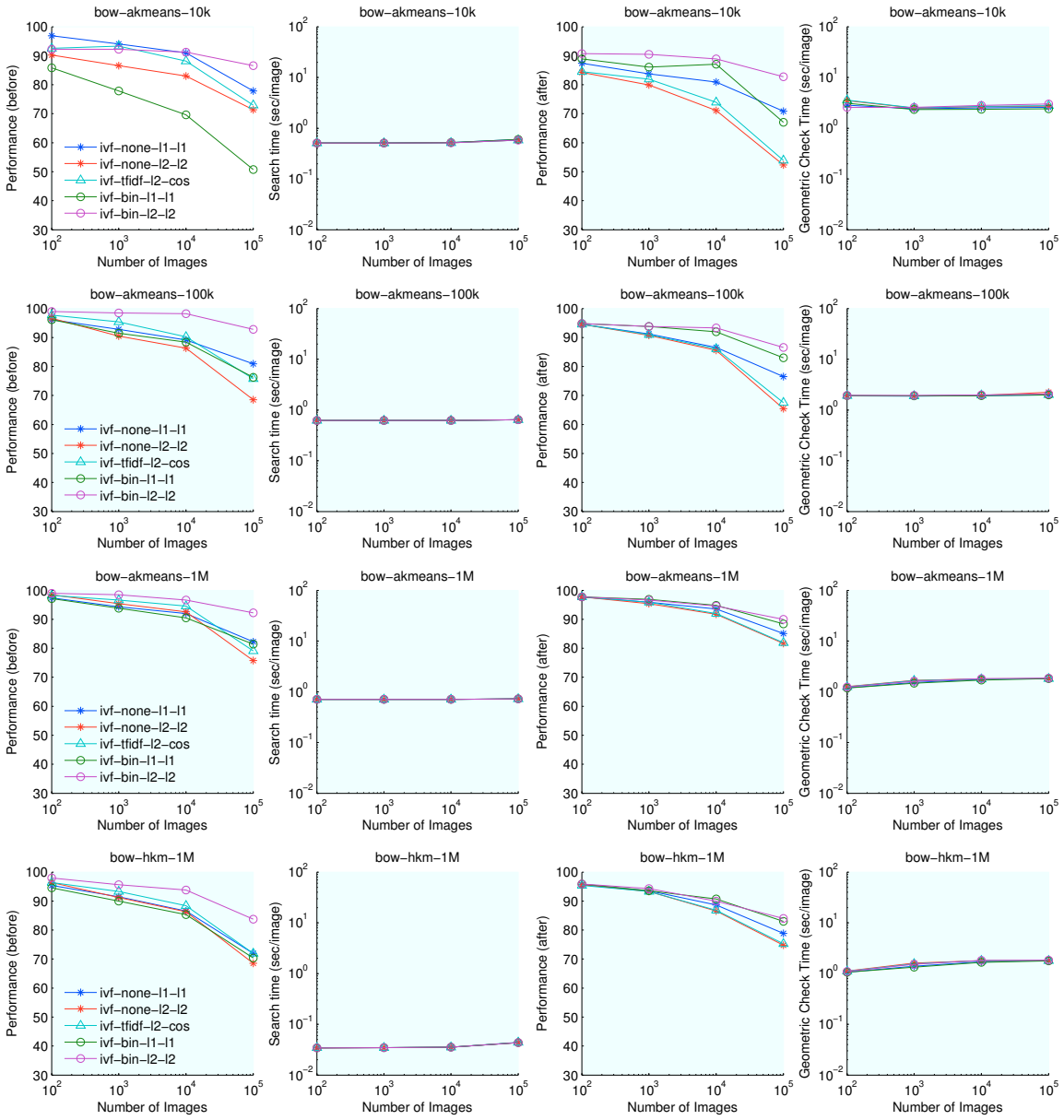


Figure 4.14: **Quick Tuning for Inverted File**. Rows correspond to dictionaries: akm-10K, akm-100K, 1km-1M, and hkm-1M, where the number corresponds to the number of words. First column depicts the recognition performance before geometric checks, second column shows the search time through the inverted file, third column shows the recognition performance after geometric step, while the fourth column shows the geometric check time. Based on the results, we chose AKM dictionaries with 1 M visual words and the three combinations {tf-idf,$l2$,cos}, {bin,$l2$,$l2$}, and {none,$l1$,$l1$} for the full tuning in Figure 4.15.

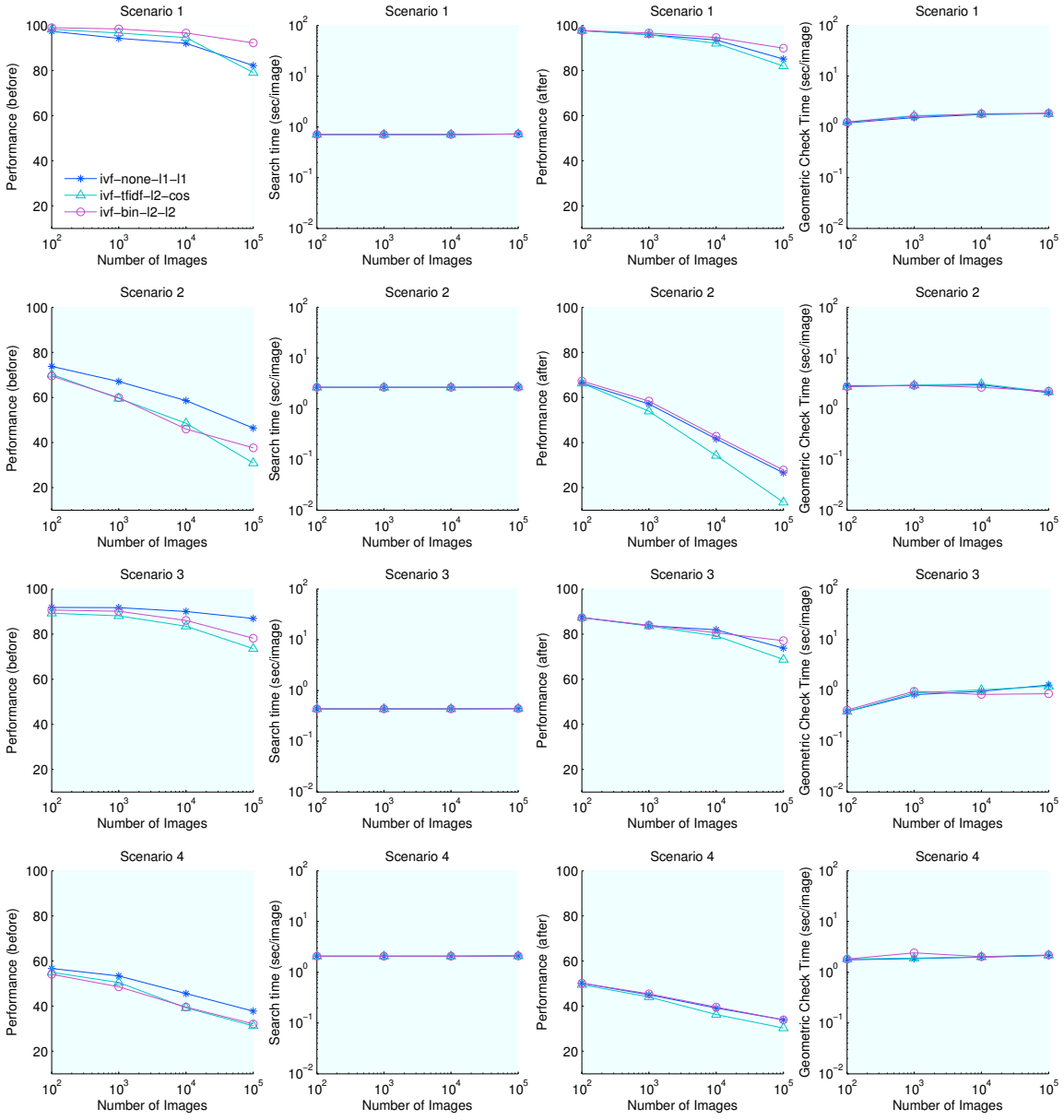


Figure 4.15: **Full Tuning for Inverted File.** Plots the results for the parameters chosen based on Figure 4.14.

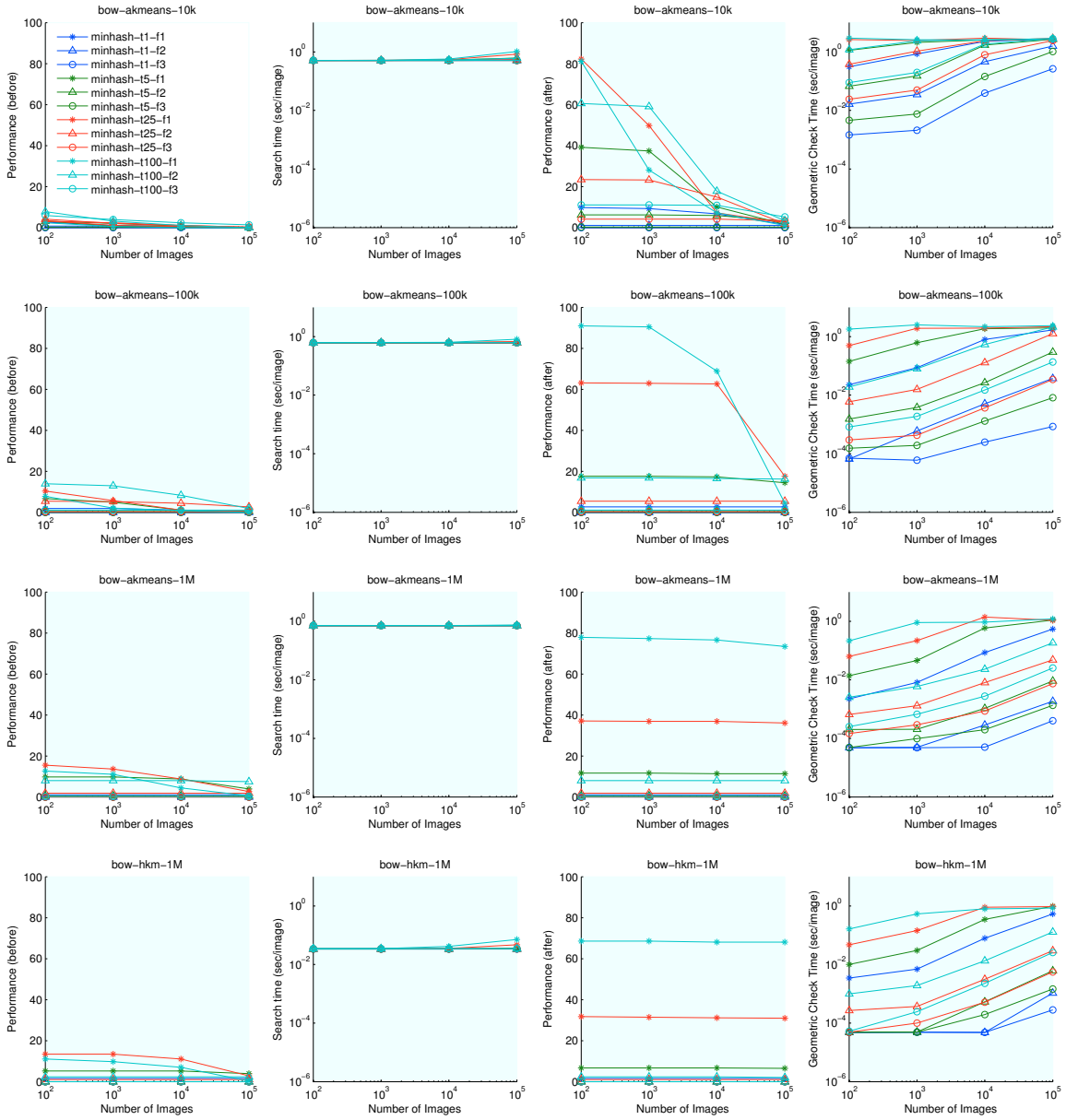


Figure 4.16: **Quick Tuning for Min-Hash**. Rows correspond to dictionaries: akm-10K, akm-100K, akm-1M, and hkm-1M, where the number corresponds to the number of words. First column depicts the recognition performance before geometric checks, second column shows the search time through the inverted file, third column shows the recognition performance after geometric step, while the fourth column shows the geometric check time. We tried T =1, 5, 25, and 100 tables with H =1, 2, and 3 hash functions. Based on these results, we chose AKM dictionaries with 1 M visual words, $T = 25$ and 100 tables with $H = 1$ for full tuning in Figure 4.17.

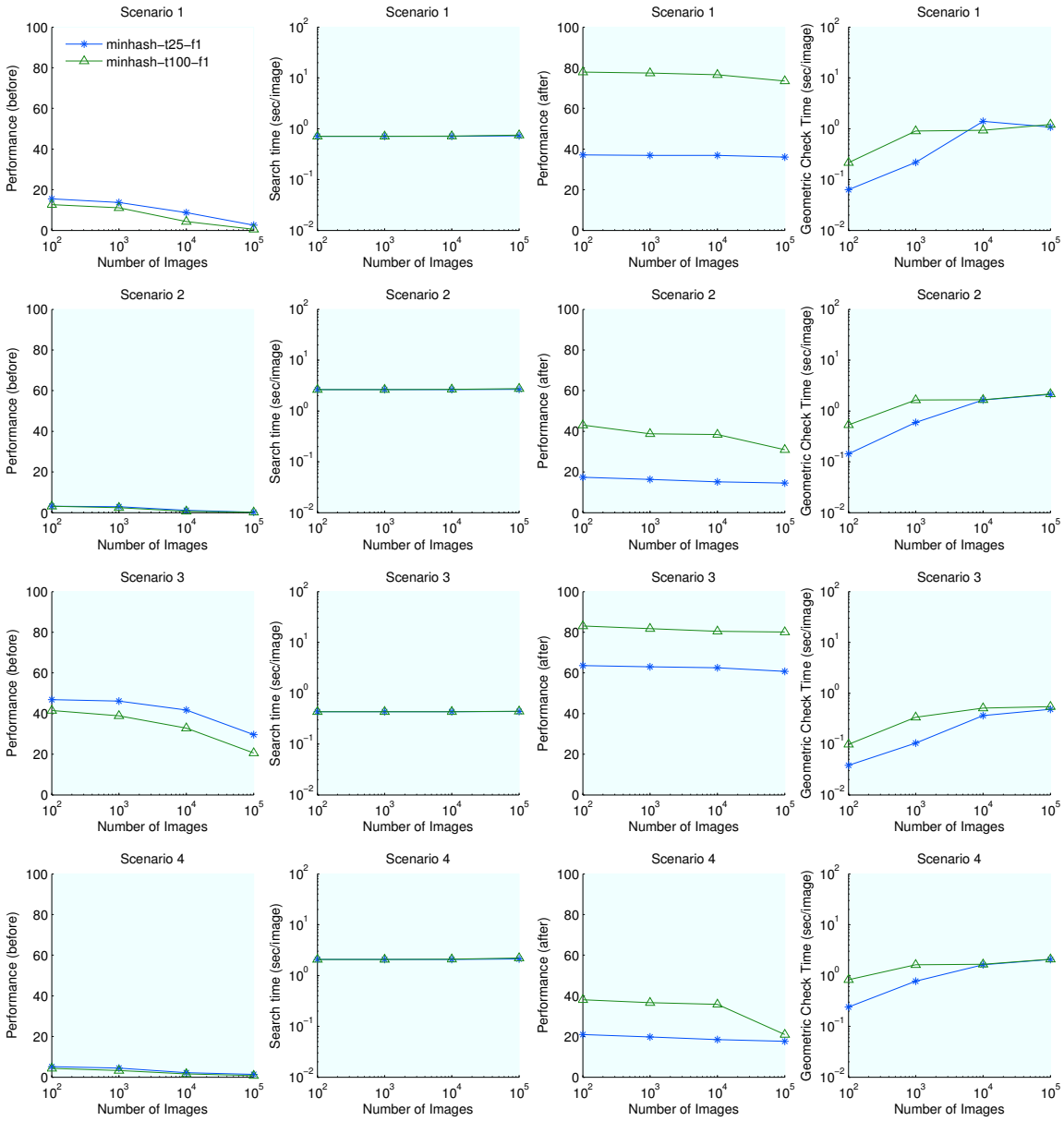


Figure 4.17: **Full Tuning for Min-Hash**. Plots results for the four scenarios for the parameters chosen in Figure 4.16. Based on these results, we chose $T = 100$ tables with $H = 1$ hash function.

4.6 Summary

This chapter provided thorough experimental comparison between Full Representation and Bag of Words methods. From the experiments, we found that FR with Kd-Trees provides better recognition performance than Inverted File BoW with competitive run time, however it requires an order of magnitude more storage (see Chapter 3). We also found that to be able to scale up the image search system to handle millions of images, we need to parallelize the database over hundreds or thousands of machines. Chapter 3 provided two ways to parallelize Kd-Trees. Therefore, in the next chapters we address these challenges:

1. Devising ways to reduce run time and storage requirements for FR methods. This includes developing better features that take less storage, finding ways to store fewer features in the database, and compressing information of features using projection methods (e.g., PCA, Random Projection, etc.). We explore this in Chapter 5 when we introduce Compact Kd-Trees.

2. Devising ways to improve the recognition performance of BoW methods. This includes finding better ways to generate the visual word dictionaries and encoding geometric information in the BoW representation. We explore this in Chapter 6 when we present Multiple Dictionaries for Bag of Words.

3. Devising ways to efficiently parallelize Kd-Trees over multiple machines. We explore this in Chapter 7 when we present implementations for Distributed Kd-Trees.

Chapter 5

Compact Kd-Trees

5.1 Introduction

As concluded from Chapters 3–4, Kd-Trees provide the best trade off between run time and recognition performance. The only drawback, which is shared with all the methods of Full Representation, is the much higher memory requirement. Most of the storage required by the FR approach is for storing the individual features. Therefore, reducing the storage required by the feature descriptors is highly desirable. This can be done in at least two ways: (a) reducing the number of features to be stored by discarding unstable useless features [34], and (b) reducing the memory footprint of the individual feature descriptors.

In this chapter, we focus on representing the features in as few bits as possible while still retaining good performance. In particular, we focus on using compact binary signatures [11, 35, 31] to compress local SIFT features [25, 24] while using Kd-Trees for fast nearest-neighbor search. We compare standard PCA dimensionality reduction to three methods for obtaining compact binary signatures from vectors: Spectral Hashing (SH) [35], Locality Sensitive Hashing (LSH) [11, 5], and Locality Sensitive Binary Codes (LSBC) [31]. Furthermore, we present a novel method, Compact Kd-Trees, that uses an order of magnitude

Parameter	Description	Typical Value
I	# images	100K
b	# bytes/feature dim	1
d	feature dimension	128
F	# features/image	1,000
T	# kd-trees	1
t	# backtracking steps	150
L	depth of the tree	24
B	# bits / signature	64

Table 5.1: **Kd-Tree Parameter Definitions**. See Section 5.2.

less storage by making use of both the full features and the binary codes while retaining comparable performance. Finally, we compare our method to the state-of-the-art Hamming Embedding BoW method [20] and report better performance with equivalent or less storage.

Section 5.2 presents the different methods of generating binary signatures for SIFT features and reviews the Kd-Tree algorithm. Section 5.3 describes the novel algorithm Compact Kd-Tree. Section 5.4 details the experimental results comparing the different binary signature generation methods, CompactKdt, and BoW methods.

5.2 Compact Binary Signatures

In this chapter, we focus on using randomized Kd-Trees, which have been shown to provide excellent recognition performance and run time [4, 26]. We use the Best-Bin-First variant of Kd-Trees [6, 24] which utilizes a priority queue [14] to search through all the trees simultaneously. Algorithm 2.2 outlines the process of constructing a set of randomized Kd-Trees and Algorithm 2.3 outlines the search process.

We can express the storage required for Kd-Tree algorithm as:

$$S_{Kdt} \approx IF(bd + T\left\lceil \frac{\log_2 IF}{8} \right\rceil) + 2^{L+1}T$$

where the parameters are defined in Table 5.1. This formula is a bit different from that in Table 3.2 since here we consider only *one* Kd-Tree, and hence we can reduce the storage even further. The first term is for storing the feature descriptors. A Kd-Tree with L levels has $2^L - 1$ internal nodes. For every internal node per tree, we need to store 2 bytes, the dimension to split on and split value (last term). For every leaf, we need to store the indices of the features that belong to that leaf (second term), the total of which is IF. The tree can be stored in memory as an array, and so we do not need any pointers [14]. Given the typical values in Table 5.1, the storage would take $S_{Kdt} \approx 13.2$ GB, out of which it takes 12.8 GB just to store the feature descriptors (i.e., ~96% of the total storage is taken by the features!).

The number of operations per image can be expressed as

$$T_{Kdt} \approx FLt + Ft\frac{FI}{2^L} \times (2d+1)$$

where t is the number of backtracking steps. The first term is for traversing the tree to the leaves, the second is for computing the distance to the candidate nearest neighbors, and the third is for finding the minimum among the candidates. For the typical values in Table 5.1, it takes $T_{Kdt} \approx 232$ MFLOP (FLoating Point Operation) per image.

We focus on methods for producing compact binary signatures for the features. These methods take in N-dimensional vectors $x \in \mathbb{R}^N$ and produce B-dimensional binary signa-

tures $b^x \in \{0,1\}^B$. The basic requirement is such that the hamming distance $d_H(b^x, b^y) = \sum_i \{b_i^x \neq b_i^y\}$ between the binary signatures b^x and b^y of two features x and y provides a good approximation to the Euclidean distance $d_E(x,y) = ||x-y||_2$.

Using binary signatures has two advantages:

1. Storage reduction: By using, for example, 64-bit signatures instead of 128-bytes features, we can reduce the storage for Kd-Trees to 1.2 GB instead of 13.2 GB, an order of magnitude less (see Table 5.2).

2. Run time Speed up: In the search phase of the Kd-Tree (Algorithm 2.3), we need to search the list of candidate nearest neighbors to get the k closest ones. Instead of computing the Euclidean distance on 128 dimensions (which includes 128 additions + 128 multiplications), for 64-bit signatures we do an XOR and bit count (one XOR + 8 table lookups and additions), almost an order of magnitude fewer operations. In addition, we need to compute the binary signatures themselves for every query feature. The search time is modified as follows:

$$T_{Kdt-Bin} \approx FtL + Ft\frac{FI}{2^L}(2\left\lceil\frac{B}{8}\right\rceil + 1) + FB \times (2d+1)$$

where the middle term is for computing the hamming distance of two B-bit vectors with lookup tables for every byte, and the last term is for computing the binary signature, assuming it is computed by projecting the $d-$dimensional feature over B vectors. For 64-bit signatures, this reduces the operations required from 232 MFLOP down to ~ 35 MFLOP per image.

# bits / feature	Total Storage (GB)	Features Compression	Total Compression
1024 (Full)	13.2	1	1
512	6.8	2	1.9
256	3.6	4	3.6
128	2	8	6.5
64	1.2	16	10.7
32	0.8	32	15.8

Table 5.2: **Storage Savings for Using Binary Signatures with Kd-Trees**. The second column depicts the storage used when compressing the SIFT features with binary signatures of different lengths, using FM Kd-Trees with typical parameter values in Table 5.1. The third column shows the compression factor for storing the features relative to the first row (full features), while the fourth column shows the total compression factor (features + Kd-Tree) achieved. See Section 5.2.

We consider three state-of-the-art methods:

- **Spectral Hashing** (SH) aims at producing balanced binary codes that minimize the mean hamming distance between similar codes [35]. This is shown to be equivalent to a graph partitioning problem which is *NP*-hard. However, by relaxing the constraints it can be solved efficiently by finding the principal components of a training set, and approximating and thresholding the eigenvectors of the graph Laplacian along the principal components [35].

- **Locality Sensitive Hashing** (LSH) uses projections on random hyperplanes such that the output binary signature approximates the dot product [11]. Specifically, the i^{th} bit is defined as: $b_i = \text{sign}\left\langle x, a^i \right\rangle$ where $a^i \in \mathbb{R}^N$ is a random unit vector $a^i \sim N(\mathbf{0}, I)$. Unlike SH, LSH is data independent and does not require any training.

- **Locality Sensitive Binary Codes** (LSBC) was derived as a data-independent variant of SH while having strong theoretical guarantees [31]. It generates binary codes

Algorithm 5.1 Compact Kd-Trees (CompactKdt)

1. For every feature in the training set $\{f_{ij}\} \in \mathbb{R}^N$, compute its binary signature $\{b_{ij}\} \in \{0,1\}^B$.
2. Build the set of Kd-Trees $\{T_t(f_{ij})\}$ using the full feature set $\{f_{ij}\}$, then only store the set of binary signatures $\{b_{ij}\}$
3. For every query feature f_{qj} compute its binary signature b_{qj}.
4. Use the full query feature f_{qj} to retrieve a short list of candidate nearest neighbor features $\{n^l_{qj}\}$.
5. Search the short list $\{n^l_{qj}\}$ of features using their binary signatures to get the closest feature n_{qj}.
6. Accumulate the scores for the image that contains n_{qj}, as in Algorithm 2.1.

---

using random projections such that the expected hamming between the signatures equals the value of a shift-invariant kernel between the two features. For exponential kernels $K(x,y) = \exp(||x-y||^2/\gamma)$, the i^{th} bit is defined as $b_i = 0.5[1+Q_t(\cos(\langle \omega, x\rangle + b))]$ where $t \sim Unif[-1,1]$, $Q_t(u) = \text{sign}(u+t)$, $\omega \sim N(0,\gamma I) \in \mathbb{R}^N$, and $b \sim Unif[0,2\pi]$.

By reducing the dimensions of the input features, we can significantly reduce the amount of storage required. Instead of using the full SIFT features, we can instead compute binary signatures or PCA and use that instead. The binary signatures are then used to build the Kd-Trees, and to compute the K-nearest neighbors (see Algorithms 2.2–2.3 and Figure 5.1). Comparison results are detailed in Section 5.4.2.

5.3 Compact Kd-Trees (CompactKdt)

Using the binary signatures to build and search the Kd-Tree provides an order of magnitude saving in storage over using the full features, but it still incurs a loss anywhere between 2%

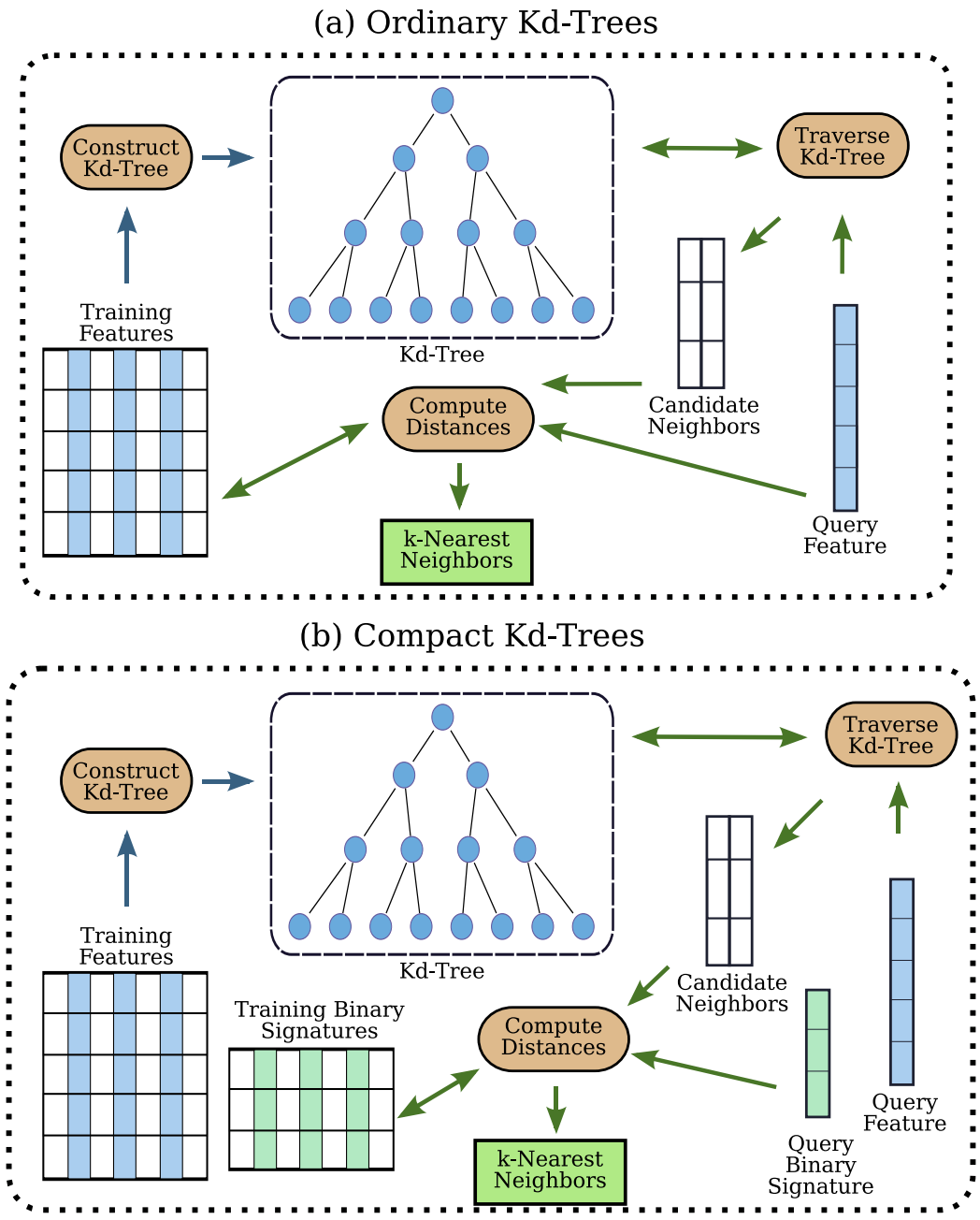


Figure 5.1: **Ordinary Vs. Compact Kd-Tree**. *Blue* arrows refer to the construction phase, while *green* arrows refer to the search phase. (a) In ordinary Kd-Trees, the training features (*left*) are used to construct the tree, see Algorithm 2.2. Given a query feature (*right*), the tree is traversed for candidate nearest neighbors, which are then filtered by computing distances using the training features to obtain the k-nearest neighbors (*bottom*); see Section 2.3. (b) In Compact Kd-Trees, the full training features are used to construct the tree as usual, and can be discarded after that. Given a query feature, the tree is traversed for candidate nearest neighbors using the full query feature descriptor. However, distances are computed using the query binary signature and the training binary signatures to obtain the k-nearest neighbors; see Algorithm 5.1. See Section 5.3.

and 36% in precision (see Section 5.4.2). We can get the best of both worlds by using all the information available, i.e., use the full features to get a *good* list of candidate nearest neighbor features, and then using the binary signatures to select the actual nearest features. This idea is easily applicable to Kd-Trees. We build the Kd-Trees using the full training features, then these features can be discarded and we only store their binary signatures. At query time, we traverse the Kd-Trees using the full query features to get the list of candidate close features, but only verify the correct nearest neighbor using the binary signatures (see Algorithm 5.1 and Figure 5.1). This significantly improves the performance while using the exact same storage and computational cost as using binary signatures with ordinary Kd-Trees (see Table 5.2 and Section 5.2). Experimental results for Compact Kd-Trees are detailed in Section 5.4.3.

5.4 Experimental Results

5.4.1 Setup

In this chapter, we only use Scenarios 1, 2, and 4 from Table 4.2 in Chapter 4. Evaluation was done by choosing 100 K images from the distractor set in addition to all the model images from the probe set. We also use the same performance metric (i.e., precision@1, see Section 4.3).

We use SIFT [24] feature descriptors (128 dimensions) with hessian affine [25] feature detectors. We used the binary available from http://tinyurl.com/vgg123. All experiments were performed on machines with Intel dual Quad-Core Xeon E5420 2.5 GHz processor

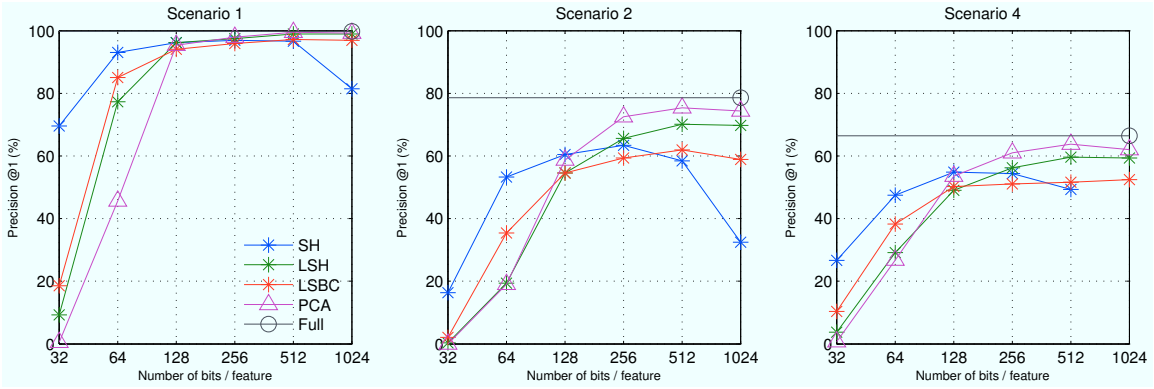


Figure 5.2: **Recognition Performance for Binary Signatures and PCA**. The X-axis shows the number of bits / feature, while the Y-axis shows the Precision@1. The *black* circles show the performance using the full features (128 bytes). PCA outperforms the binary signatures beyond 128 bits, while the binary signatures (especially SH) perform very well for 32 and 64 bits. See Sections 5.2 and 5.4.2. PCA achieves performance within 90% of using the full features with ~ 3.6 compression factor. To achieve even higher savings, see Figure 5.3.

and 32 GB of RAM. For SH, we used the code available from the authors of [35]. We implemented the rest of the algorithms using Matlab and Mex/C++ scripts. For SH, we used a random set of 2.5 M for training. For BoW, the dictionary is built from a random set of 10 M features.

5.4.2 Binary Signature Comparison

Figure 5.2 shows the results of using the binary signatures compared to using Principal Component Analysis (PCA) and to using the full features with Kd-Trees. For PCA, all the dimensions were quantized to 8-bits (e.g., with 128 bits we only kept the top 16 dimensions). We note the following:

- The scenarios have different difficulty, with Scenario 1 the easiest (full feature precision of about 99%), Scenario 2 harder (~ 79% precision), and Scenario 4 the hardest

(~ 66% precision).

- PCA is quite competitive with the binary signatures for sizes starting at 128 bits per feature (16 PCA dimensions). By using 256 bits (32 PCA dimensions), we can reach performance within 90% of the full features while achieving ~3.6 compression i.e. using almost one fourth of the storage.

- Below 128 bits, the binary signatures are significantly better than PCA. SH provides the best performance, followed by LSBC, then LSH. However, beyond 128 bits, the performances of SH and LSBC deteriorate, while that of LSH becomes better.

- At 128 bits (~ 6-fold savings in total storage) we lose ~ 2% of the precision using full features for Scenario 1, ~ 28% for Scenario 2, and ~ 21% for Scenario 4.

- At 64 bits (~ 10-fold savings in total storage) we lose ~ 4% of the precision using full features for Scenario 1, ~ 36% for Scenario 2, and ~ 31% for Scenario 4.

Though we can achieve a ~ 3.6 compression factor with only minimal loss to the performance, which is promising, we are seeking even higher compression rates with similar performance. We will show next how to achieve this.

5.4.3 Compact Kd-Tree

Figure 5.3 shows the results of running CompactKdt using the binary signatures and PCA from Section 5.2 compared to using SH with ordinary Kd-Trees (which provides the best performance from Figure 5.2), all using 1 Kd-Tree. We note the following:

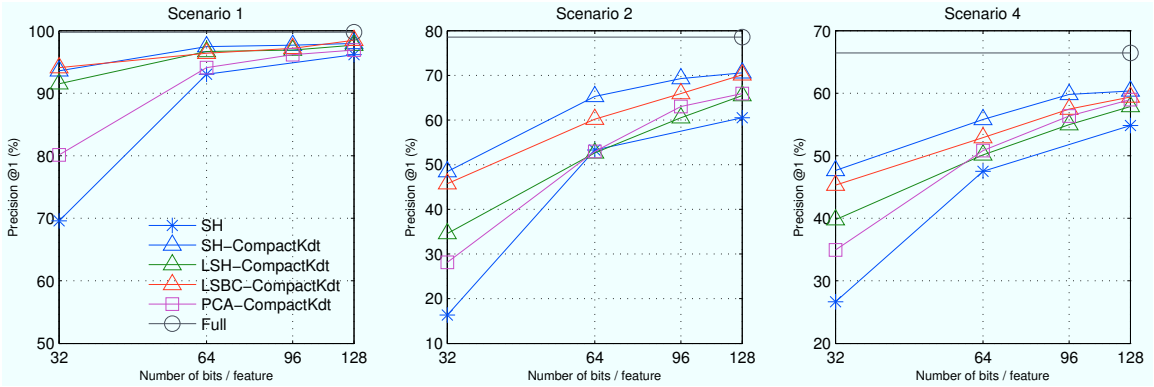


Figure 5.3: **Recognition Performance for Compact Kd-Trees (CompactKdt)**. The X-axis shows the number of bits / feature, while the Y-axis shows the Precision@1. The *black* circles show the performance using the full features (128 bytes), while the *blue* stars show the SH with ordinary Kd-Trees which provided the best performance from Figure 5.2. We note that CompactKdt provides significant performance gains while using the same storage as using ordinary Kd-Trees with binary signatures. CompactKdt with SH gives the best performance, followed by LSBC, LSH, and PCA. See Section 5.3.

- CompactKdt with SH gives the best results, while using LSH, LSBC, and PCA are worse.

- CompactKdt achieves significant performance improvements over using the binary signatures with ordinary Kd-Trees, while using the same storage. For example, using 32-bits, we achieve 35–200% improvements.

- For Scenario 1, we reach within ~ 5% of the full feature precision using only 32 bits with SH-CompactKdt compared to SH alone, a 14-fold saving in storage.

- For Scenarios 2 and 4, we reach within ~ 10% (16%) of the full precision using only 96 bits (64 bits) for ~ 8-fold (10-fold, respectively) of savings in storage. Therefore, if the application is willing to lose 10–16% of the precision, we can achieve a compression factor of 8–10 times.

Parameter	Description	Typical Value
I	# images	100K
F	# features/image	1,000
b	# bytes/feature dim	1
d	feature dimension	128
W	# visual words	10^6
B	# bits / signature	64

Table 5.3: **BoW Parameter Definitions**. See Section 5.4.4.

Algorithm	Storage (GB)	Comp. (MFLOP/im)	Query Time (msec/im)
Kd-Tree	13.2	232	232
CompactKdt-64	1.23	35	35
CompactKdt-48	1.03	27	27
BoW-HE	1.1	51	51

Table 5.4: **CompactKdt and BoW Storage and Computational Cost Comparison** using the values in Tables 5.1 and 5.3. See Sections 5.2, 5.3, and 5.4.4 and Figure 5.4. Storage is in GB, computations are in MFLOP/image, and query time is in msec/image (using a 1 GFLOPS processor).

- Overall, we can save anywhere between 8–14 times in the total storage and stay within 5–16% of the best performance. This is a significant storage saving, since we can store an order of magnitude more images on the same machine while losing minimal performance.

5.4.4 Comparison with Bag of Words

Bag of Words (BoW) techniques have been shown to take an order of magnitude less storage than FR methods, however, they suffer from poor performance (see Figure 5.4 and

Chapters 3 and 4). The storage taken by BoW methods can be expressed as (see Table 3.2):

$$S_{BoW} = Wbd + IF\left\lceil \frac{\log_2 I}{8} \right\rceil$$

where the parameters are defined in Table 5.3. The first term is for storing the visual words, and is negligible. The second term is for storing the inverted file [36]. We do not need to store word counts for large dictionaries as the histogram usually becomes sparse and binary [20]. For Hamming Embedding (HE) [20], the storage needed is as follows:

$$S_{HE} = Wbd + IF\left(\left\lceil \frac{\log_2 I}{8} \right\rceil + \frac{B}{8}\right)$$

where we need additional storage for the binary signatures of the features. For the typical values in Table 5.3, $S_{BoW} = 0.3$ GB and $S_{HE} = 1.1$ GB, compared to $S_{Kdt} = 13.2$ GB, which is an order of magnitude more. Using CompactKdt, however, the storage shrinks to $S_{CompactKdt-64} = 1.23$ GB with 64-bit signatures and to $S_{CompactKdt-48} = 1.03$ GB with 48-bit signatures (see Section 5.3 and Tables 5.2 and 5.4) which is equivalent to or less than the storage taken by HE.

The number of operations per image for HE is as follows:

$$T_{HE} \approx Ft(\log_2 W + 2d + 1) + 2F\frac{FI}{W}\frac{B}{8} + FB \times (2d + 1)$$

where the first term is for computing the visual words for the image features using a Kd-Tree with W leaves and $\log_2 W$ levels, the second term is for computing the Hamming

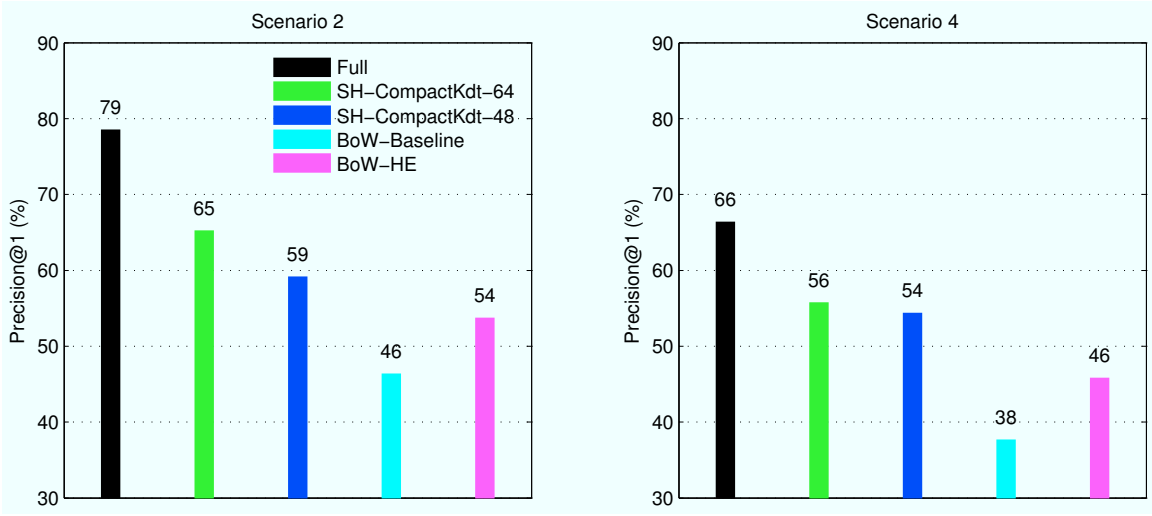


Figure 5.4: **Comparison of CompactKdt with BoW**. The X-axis shows different algorithms, while the Y-axis shows their Precision@1. The full features with 1Kd-Tree is in *black*. Two bars for CompactKdt with 1 tree using 64-bit signatures (*green*) and 48-bits (*red*). Baseline Bag of Words with 1 M visual words is in *cyan*, while Hamming Embedding BoW with 100 K visual words is in *magenta*. See Section 5.4.4.

distance between the binary signatures (assuming the features are evenly distributed among the visual words, such that each of the W visual words has $\frac{FI}{W}$ features given FI total features), and the third term is for computing the signatures themselves. With typical values in Table 5.3, this gives $T_{HE} \approx 51$ MFLOP compared to $T_{CompactKdt-64} \approx 35$ MFLOP (see Table 5.4).

Figure 5.4 shows results for: (a) Baseline BoW: the standard BoW method with l_1 histogram normalization and l_1 distance using 1 M visual words (*cyan*); (b) HE BoW: with tf-idf weighting, l_2 normalization, l_2 distance, and hamming distance threshold of 25 using 100 K visual words (*magenta*); (c) Full SIFT features without compression using 1 Kd-Tree (*black*); and (d) CompactKdt with SH 64-bit (*green*) and 48-bit (*blue*) signatures using 1 Kd-Tree. We note the following:

- Kd-Trees with full features (*black*) are significantly superior to both Baseline (*cyan*) and HE BoW (*magenta*) methods, at the cost of using an order of magnitude more storage.

- CompactKdt, with either 48 or 64 bits (*green & blue*), is clearly superior to both Baseline BoW and HE BoW. It provides significantly superior recognition performance with equivalent or less storage and computational cost.

5.5 Summary

We presented a novel algorithm, Compact Kd-Tree, for reducing the storage of SIFT features using compact binary signatures together with Kd-Tree constructed with the full features. We find that CompactKdt can reduce the computational cost and the storage by an order of magnitude, down to levels comparable to BoW methods, while retaining the superior performance of FR methods. Specifically, we showed an order of magnitude less storage (~ 8-fold saving) with performance within 10% of the performance using the full features using 12 bytes per feature. In addition, CompactKdt achieves significantly better performance than the state-of-the-art Hamming Embedding method with equivalent or less storage and computational cost.

Chapter 6

Multiple Dictionaries for Bag of Words

6.1 Introduction

As concluded from Chapters 3–4, the Bag of Words approach requires significantly less storage than the Full Representation approach, but suffers from worse performance. In this chapter, we explore ways to boost its performance. Specifically, we focus our attention on the most important component of the BoW algorithm: the dictionary of visual words (see Section 2.6). We present a novel method, Multiple Dictionaries for BoW (MDBoW), that uses more visual words (~ 5 M) while significantly increasing the performance. Unlike previous approaches, we use more words from different independent dictionaries instead of adding more words to the same dictionary. The caveat is that the storage grows linearly with the number of dictionaries used, however, with recent techniques for compact BoW [21, 22, 29], we can get improved performance using comparable storage. We also show significant performance gains by building the dictionary using all available features (i.e., using features from *all* images indexed in the database). We argue that this poses no risk of overfitting or bias, since this is exactly the approach taken by other image search methods that rely on matching individual features using, e.g., Kd-Trees [24, 4]. Finally, we report

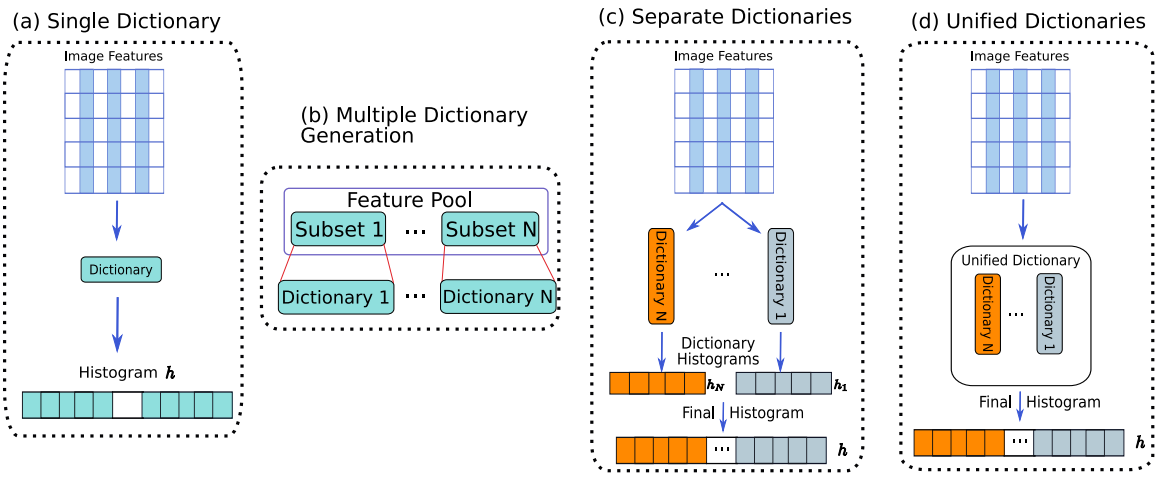


Figure 6.1: **Multiple Dictionaries for BoW**. *(a) Single Dictionary*: a single dictionary of visual words is generated from the pool of features, which is used to generate the histogram for the image. *(b)* Every dictionary D_n is generated with a different subset of the image features. *(c) Separate Dictionaries (SDs)*: the image gets a histogram h_n from every dictionary D_n which are concatenated to form a single histogram h. Notice that every feature gets N entries in the histogram h, one from every dictionary. *(d) Unified Dictionaries (UDs)*: a single unified dictionary is built from the concatenation of visual words from the dictionaries $1 \ldots N$, and the image gets a single histogram h. Note that every feature gets only one entry in the histogram h.

results significantly better than the state-of-the-art Hamming Embedding [20] method.

Section 6.2 describes the idea behind MDBoW. Section 6.3 presents the experimental details. Finally, Section 6.4 presents the experimental results.

6.2 Multiple Dictionaries for Bag of Words (MDBoW)

We propose a novel way to increase the recognition performance of BoW: Multiple Dictionaries for Bag of Words (MDBoW). Our motivation is the view of BoW as an approximation to matching individual features [20] and the idea of Randomized Kd-Trees [6, 24, 26]. In this view, we are matching individual features using visual words as a proxy, i.e., features that have the same visual word are considered matched. However, since we are dealing with

Algorithm 6.1 Multiple Dictionaries for Bag of Words (MDBoW)

1. Generate N random (possibly overlapping) subsets of the image features $\{S_n\}_1^N$.
2. Compute a dictionary D_n independently for each subset S_n. Each dictionary D_n has a set of K_n visual words (cluster centers) $\{W_k^n\}_{k=1}^{K_n}$.
3. Compute the histogram h for any image using the combination of the N dictionaries in one of two ways:
   (a) ***Separate Dictionaries*** **(SDs)**: For every image feature f_j, get its visual word w_j^n from every dictionary D_n. Accumulate these visual words into individual histograms h_n for each dictionary. The final histogram $h = [h_1^T \dots h_N^T]^T$ is the concatenation of the individual histograms.
   (b) ***Unified Dictionaries*** **(UDs)**: Combine the visual words $\{W_k^n\}$ from all the dictionaries into a unified set of words W. For every image feature f_j, get its visual word w_j from the unified set of words. Accumulate the visual words into the final histogram h.

very large dimensions, this approximation depends greatly on the random partitioning of the space offered by AKM. We can solve this problem by having multiple independent dictionaries that give complementary partitions of the feature space, in the spirit of Randomized Kd-Trees. Algorithm 6.1 outlines the idea, please consult Figure 6.1 for illustrations.

The advantage of the SDs method is that it is flexible and can be used with any kind of dictionary. There is no restriction on the way the dictionaries are generated e.g. we can combine AKM and Hierarchical K-Means (HKM) dictionaries [27] with varying sizes. In fact, we show results on SDs with HE in Section 6.4.3. The drawback is that we are increasing the storage requirements in the inverted file, since each feature in the image has N entries in the final histogram, as well as the time to generate the visual words (see Table 6.1). On the other hand, UDs have the advantage of requiring less memory than SDs, since each feature has only one entry in the final histogram. However, these dictionaries must be of a type where it is easy to combine visual words from different dictionaries, which is not

Parameter	Description	Typical Value
I	no. of images	10^6
s	bytes/feature dim	1
d	feature dimension	128
N	# dictionaries	varies
F	#features/image	1,000
T	# kd-trees	8
B	# backtracking	100
W	# words	varies

	Storage (B)	Computation (FLOP/f)
Single Dictionary	$Wsd + FI(\frac{\log_2 I}{8} + 1)$	$B(2d + 1 + \log_2 W)$
Separate Dictionaries	$Wsd + NFI(\frac{\log_2 I}{8} + 1)$	$NB(2d + 1 + \log_2 \frac{W}{N})$
Unified Dictionaries	$Wsd + FI(\frac{\log_2 I}{8} + 1)$	$B(2d + 1 + \log_2 W)$

Table 6.1: **MDBoW Parameter Definitions and Properties**. *(Top)* Definitions of parameters affecting the computational and memory requirements (*bottom*) of MDBoW. Storage is in bytes and computation is in *FLOP/feature*. See Section 6.2 and Figure 6.2.

the case with HKM dictionaries, for instance [27]. Table 6.1 and Figure 6.2 summarize this comparison for dictionaries of 100 K and 1 M words each. We note the following:

- The storage requirements of BoW is dominated by the inverted file structure (second term in the formula in the second column in Table 6.1), while storing the words themselves is negligible. The second term grows linearly with the number of entries in the inverted file, which is controlled by the number of images I and the number of separate dictionaries N.

- The computational cost to generate visual words for the features is almost independent of the number of words. This is because the dominant factor is the backtracking in the set of Kd-Trees (the first two terms in the formula in the third column in Table 6.1), while traversing the depth of the tree (last term) grows only logarithmically with the number of words.

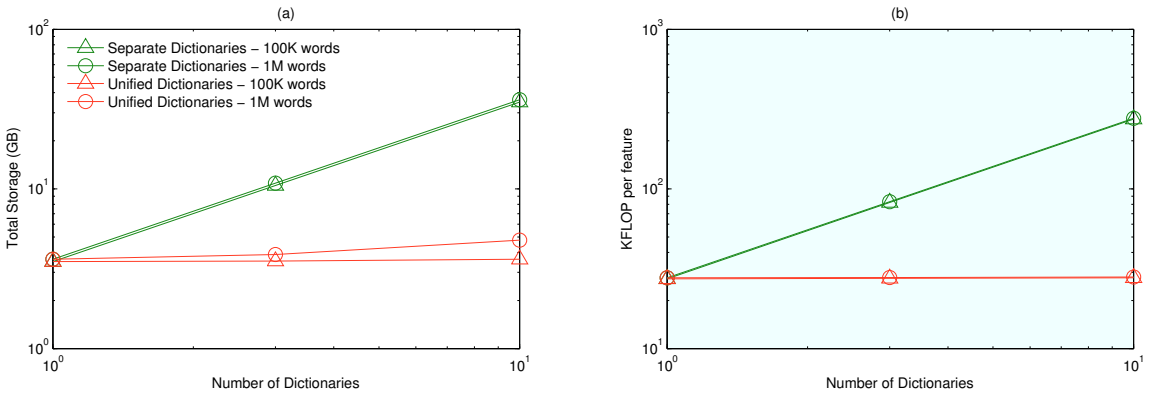


Figure 6.2: **MDBoW Memory and Computational Requirements**. The X-axis shows the number of dictionaries, while the Y-axis plots *(a)* storage requirements for 1 M images and *(b)* computational cost per feature in KFLOP. Individual dictionaries have either 100 K words (*triangles*) or 1M words (*circles*). We notice that the cost (in memory and computation) of SDs increases linearly with the number of dictionaries, independent of the number of words. Parallelizing these dictionaries on a set of machines would exhibit run time comparable to a single dictionary albeit with much higher recognition performance. Also the cost of UDs is almost independent of the number of words, similar to a single dictionary. See Section 6.2, Table 6.1, and Figure 6.4.

- SDs storage increases linearly with the number of dictionaries, as the entries in the inverted file multiply.
- SDs run time also grows linearly with the number of dictionaries. This is because we have separate Kd-Trees for every dictionary, whereas the UDs have just one set of Kd-Trees for the unified set of words.

6.3 Experimental Details

6.3.1 Setup

In this chapter, we only use Scenarios 2 and 4 from Table 4.2 in Chapter 4. Evaluation was done by choosing 100 K images from the distractor set in addition to all the model images

from the probe set. We also use the same performance metric i.e. precision@1, see Section 4.3.

We use SIFT [24] feature descriptors (128 dimensions) with Hessian affine [25] feature detectors. We used the publicly available binary available from http://tinyurl.com/vgg123. All experiments were performed on machines with an Intel dual Quad-Core Xeon E5420 2.5 GHz processor and 32 GB of RAM. We implemented all the algorithms using Matlab and Mex/C++ scripts. The dictionaries were generated with random subsets of features of size 10 M, out of ~ 150 M features in the distractor set.

6.3.2 Bag of Words Details

In this chapter we focus on two variants of BoW Inverted File search (see Chapter 2):

1. **Baseline IF**: This is the standard way of using IF with BoW (see Section 2.6.1 for details). The dictionaries are built using Approximate K-Means (AKM) [30], which uses a set of randomized Kd-Trees inside K-Means to get the nearest cluster centers. Histograms are normalized to have unit l_1 norm, then the l_1 distance is used to measure similarity between histograms. This combination has been shown to outperform using the l_2 norm and the cos distance [27, 1].

2. **Hamming Embedding IF (HE)**: This is the method introduced by Jégou *et al.* [20]. The idea is to be more discriminative when computing distances between histograms. Instead of matching any two features that belong to the same visual word, each feature has an $N-$bit binary signature $b \in \{0,1\}^N$, such that two features f_i and f_j that belong to the same word will match only if the hamming distance $d_H(b_i, b_j) \leq T$

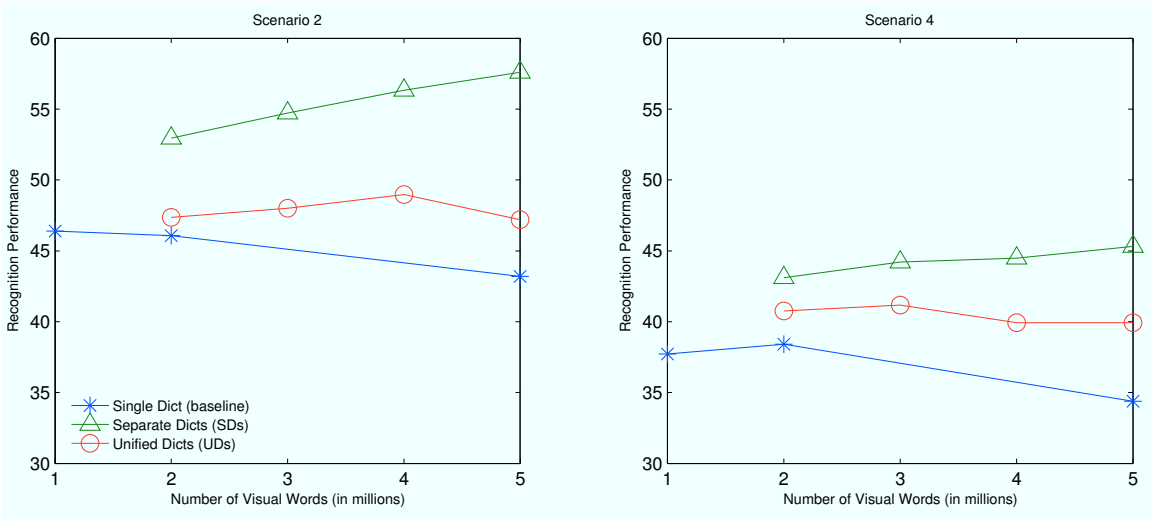


Figure 6.3: **Multiple Dictionaries for BoW Results**. The X-axis shows the total number of visual words, while the Y-axis plots the recognition performance for 100K images using Baseline IF. For UDs and SDs, individual component dictionaries have 1 M words each e.g. SDs with 2 M visual words have 2 dictionaries. We notice a performance increase of about 25% using SDs with 5M words over the baseline of using 1 M words. We also note that SDs have far superior performance than UDs. Note that simply increasing the number of words in a single dictionary (*blue*) is still worse than both SDs and UDs. See Section 6.2 and 6.4.1.

where T is some threshold. We use the settings from [20]: $N = 64$ bits, T=25, Tf-Idf histogram weighting, l_2 normalization, and the cos distance.

6.4 Experimental Results

6.4.1 Multiple Dictionaries for BoW (MDBoW)

Figure 6.3 shows results of using multiple dictionaries compared to using a single dictionary. All the dictionaries are generated with a random subset of 10 million features and use the baseline IF (see Sec. 6.3.2). We note the following:

- Using UDs (*red*) improves the performance slightly by about 3% over the baseline

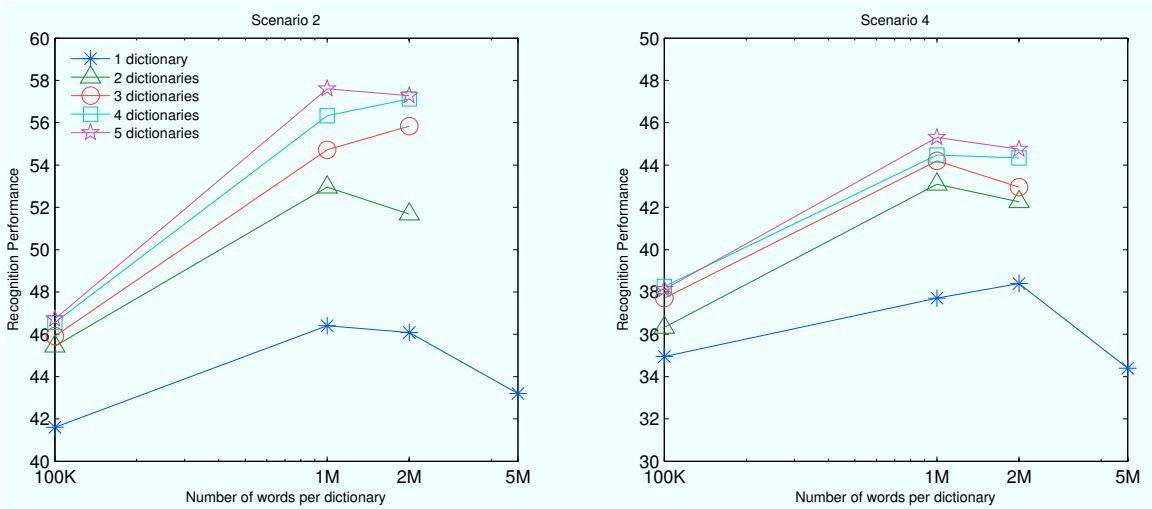


Figure 6.4: **Parallelization of Multiple Dictionaries for BoW**. The X-axis shows the number of visual words per dictionary, while the Y-axis plots the recognition performance for 100 K images. Each point depicts the performance (on the Y-axis) of using a certain number of dictionaries (from 1 up to 5 dictionaries), one on each machine, each having the same number of words (on the X-axis). The best performance is achieved using 5 dictionaries (*magenta* curve) with 1 M words each. Note that increasing the number of words in a single dictionary (*blue* curve) does not help beyond 1 M or 2 M words. See Sections 6.2 and 6.4.1.

(*blue*).

- Using SDs (*green*) significantly improves the performance by about 20–25% over the baseline with 1 dictionary (*blue*). This can be explained by the fact that SDs use more information than UDs. In particular, every image feature in SDs gives information in each of the individual histograms, while in UDs that is not the case. This helps increase the performance, as we get multiple independent partitions of the feature space.

- Simply increasing the number of visual words in a single dictionary (*blue*) does not help, and in fact decreases the performance, as reported in [30].

We can view SDs as a trade-off between storage and computation required and recognition performance. Using more independent dictionaries enhances performance, but increases storage and computation (see Figure 6.2). One way to solve this problem is to deploy each dictionary on a separate machine and then combine the results, see Figure 6.4. The running time of the whole system would be the same as using one dictionary on a single machine, since they are running in parallel and we will have a significant increase in performance. We note that we can get the best performance when using 5 dictionaries/machines with 1 M words each. Another way to solve the problem of increased storage is to apply SDs to one of the recent compact BoW methods [21, 22, 29], which have been shown to give comparable performance to the baseline IF with 1 dictionary while requiring a fraction of the storage (e.g., 16-32 bytes per image).

6.4.2 Model Features

The standard approach for building the dictionary of visual words has been to either use a set of unrelated images [30, 20] or to exclude the model images (see Section 4.2) from this process [4, 1]. The motivation for this distinction is to obtain some kind of a universal dictionary, that is unrelated to the set of images to be searched. We argue the opposite, that this is not necessary and is in fact harmful, for the following reasons:

1. In the actual setting, say, for example, the Barnes and Noble application, we have a set of images of book covers that we want to index. It would not make sense to build the dictionary on a separate set of unrelated images. We should be able to optimize the system for the database at hand, and use these images to generate the dictionary.

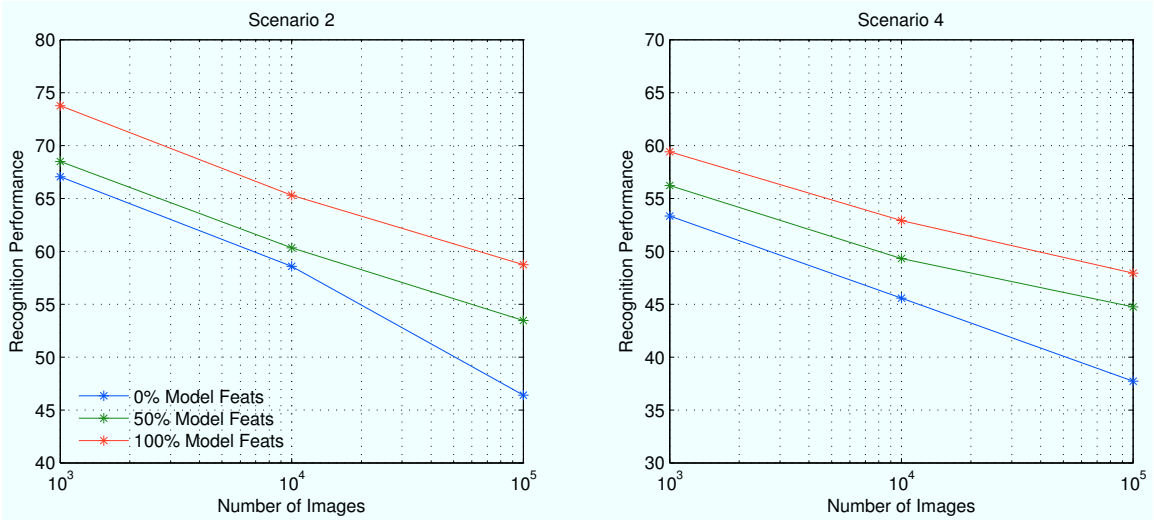


Figure 6.5: **Model Features Results**. The X-axis shows the number distractor images in the database, while the Y-axis plots the recognition performance using Baseline IF. We notice a significant performance increase by including features of the model images in the dictionary generation. See Section 6.4.2.

This does not pose any danger of overfitting, because we are not using the probe (query or test) images, we are rather using the training images in the optimal way.

2. Other image search approaches that are based on feature matching (e.g., Kd-Trees) do not have this distinction. They use all features available when matching the features of the probe image. That might be one reason why they have been reported to outperform BoW methods in this application [4].

Figure 6.5 shows the results of including the model features in the dictionary generation process. Dictionaries are generated with a random subset of 10 M features with Baseline IF and 1 M words (see Section 6.3.2). Probe images have on average 1 K features each (i.e., they constitute ~ 1% of the number of features used to build the dictionaries). Different percentages of model features were included in the random 10M features: 0% (no model features), 50%, and 100% (all the model features). We notice the following:

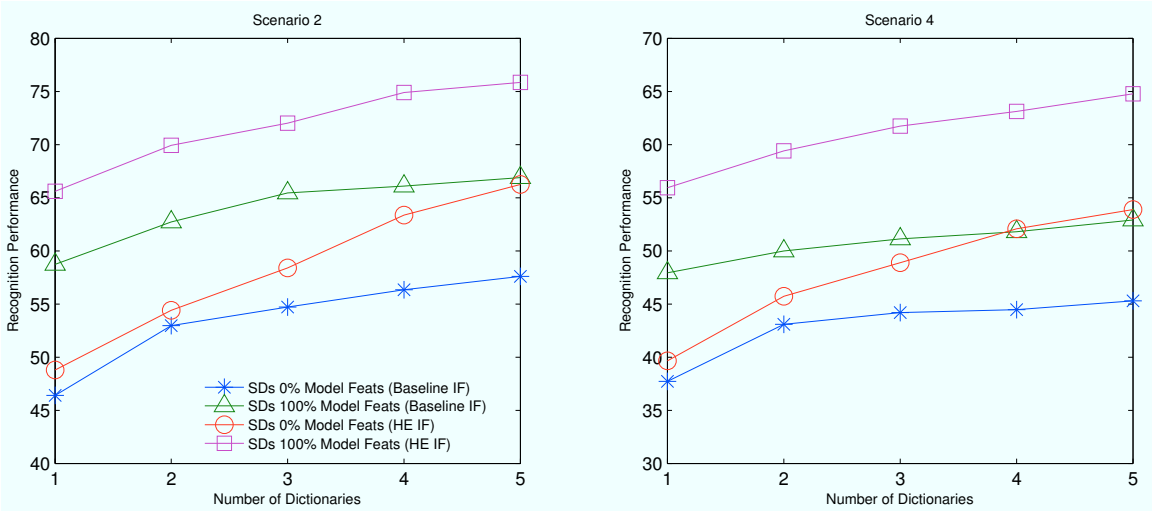


Figure 6.6: **Combining Multiple Dictionaries (MDBoW) with Model Features**. The X-axis shows the number of dictionaries, while the Y-axis plots the recognition performance. We plot the baseline IF with/without model features and HE IF with/without model features. We notice about 40–45% increase in recognition performance over the baseline (1 dictionary) by combining model features and SDs (*green*) and about 20% increase over SDs alone (*blue*). We also notice that applying MDBoW to HE achieves 65–75% over the baseline and 25–38% over the state-of-the-art HE with 1 dictionary. See Figure 6.3–6.5 and Sections 6.2 and 6.4.2.

- Adding all model features improves performance by up to 25% of the baseline.
- In the actual setting, where we use a subset of all the features we have, the performance increase will be lower. However, it will still be much better than using an unrelated set of features for building the dictionary.
- This modification does not alter the computation nor the storage requirements and can be easily applied to MDBoW approach (see Figure 6.6).

6.4.3 Putting It Together

Figure 6.6 shows the results of combining the two ideas: using separate multiple dictionaries (SD MDBoW) (from Section 6.2) and including model features in dictionary generation

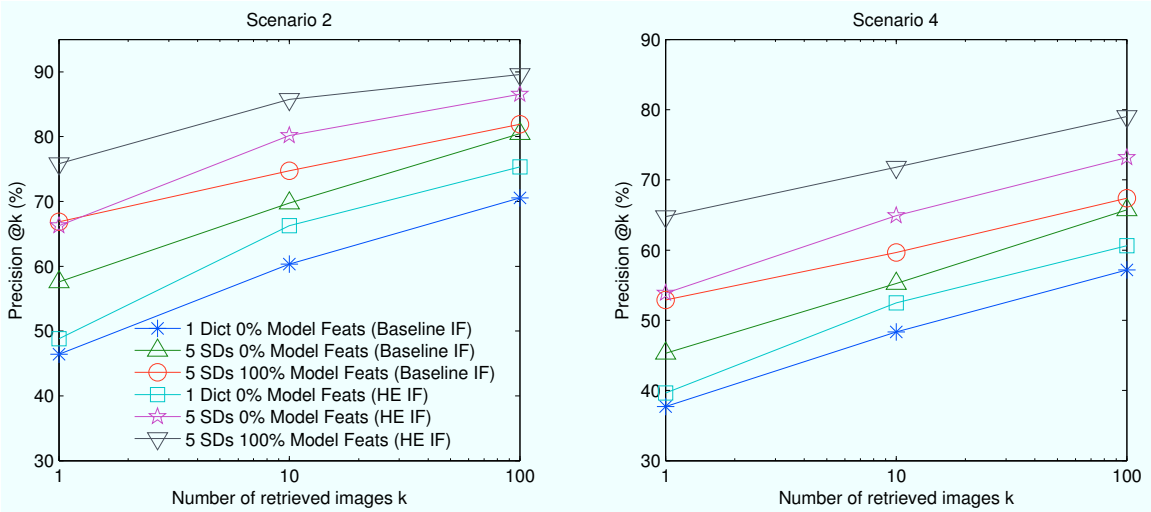


Figure 6.7: **Precision@*k***. The X-axis shows the number of retrieved images k, while the Y-axis shows the precision@*k*, i.e., the recognition rate when the ground truth image is among the top *k* images. We note significant improvement for SDs (*green* & *magenta*) over the baseline IF (*blue*) and state-of-the-art HE IF (*cyan*) with only 1 dictionary. We also note significant improvements by including the model features (*red* & *black*). See Section 6.4.3 and Figure 6.6.

(from Section 6.4.2). It compares both Baseline IF and Hamming Embedding (HE) IF [20] (see Section 6.3.2). We notice that combining model features with MDBoW using either Baseline IF (*green*) or HE IF (*cyan*) significantly improves the performance over not using them (*blue* and *red*).

Figure 6.7 shows the precision@*k* for Baseline and HE IF with 1 dictionary and 5 SD MDBoW with/without model features. We note the following:

- For Baseline IF, using the model features with SD MDBoW (*red*) provides 18–25% performance increase over not using them (*green*). It also gives a performance increase of about 40–45% over using 1 dictionary with Baseline IF (*blue*).

- MDBoW with baseline IF (*green*) significantly outperforms the state-of-the-art HE IF with 1 dictionary (*cyan*) by ~ 15%.

- For Hamming Embedding IF, using MDBoW with HE without the model features (*magenta*) gives a performance increase of ~ 32% over the state-of-the-art HE with 1 dictionary (*cyan*). Moreover, combining the model features with MDBoW using HE (*black*) gives a performance increase of about 65–75% over the Baseline IF with 1 dictionary (*blue*) and about 25–38% performance increase over HE with 1 dictionary (*cyan*).

6.5 Summary

We explored ways to boost the performance of BoW image search methods by using more visual words. We presented a novel algorithm, MDBoW, and showed that using multiple independent dictionaries built from different subsets of the features increases significantly the recognition performance of BoW systems. We analyzed its cost and provided a simple way to parallelize N dictionaries over N machines and retain the run time of the baseline method. We showed performance improvements by 20–32% over the baseline and ~ 15% over the state-of-the-art. We argued that including features from indexed images when building the dictionaries is the right thing to do, and showed that it provides 25% improvement. We finally showed that combining these two ideas can yield 40–45% improvement in recognition performance over the baseline IF and 25–38% improvement over the state-of-the-art Hamming Embedding method.

Chapter 7

Distributed KD-Trees

7.1 Introduction

Chapter 3 discussed different ways to parallelize Kd-Trees to go beyond one machine and to be able to scale up the recognition system to millions of images. It discussed Independent Kd-Trees (IKdt), where the images are divided equally among different machines, each building its own Kd-Tree from its chunk of images. It also introduced the novel idea of Distributed Kd-Trees (DKdt), where a Kd-Tree is divided into a "root subtree" that resides on a root machine, and several "leaf subtrees", each residing on a leaf machine (see Section 3.3.2). In this chapter we explore this idea further, and provide practical implementations for both IKdt and DKdt using the MapReduce paradigm [16]. We compare the two methods, discuss the effect of the different parameters on the performance, and run experiments on up to 100 images on over 2000 machines. Our experiments show the superiority of DKdt over IKdt in terms of recognition performance and run time. Specifically, DKdt provides 30% better recognition performance than IKdt with 100 times more throughput, processing a query image in a fraction of a second.

Section 7.2 gives a brief description of the MapReduce paradigm, followed by descrip-

```
Map(String key, String value):
 for each word w in value:
  EmitIntermediate(w, "1");

Reduce(String key, Iterator values):
 int result = 0;
 for each v in values:
  result += ParseInt(v);
 Emit(AsString(result));
```

Figure 7.1: **Canonical MapReduce Example**. The *Map* function outputs a value of "1" for every input word. The *Reduce* function sums up these values, and outputs for every word the number of times it appeared in the input document. See Section 7.2.

tion of the implementation of IKdt and DKdt using MapReduce in Section 7.3. Section 7.4 details the experimental setup, and Section 7.5 discusses the experimental results.

7.2 MapReduce Paradigm

MapReduce [16] is a software framework introduced by Google to support distributed processing of large data sets on large clusters of computers. The software model takes in a set of *input* key/value pairs and produces a set of *output* key/value pairs. The user needs to supply two functions: **(1)** ***Map***: takes an input pair and produces a set of *intermediate* key/value pairs. The library collects together all intermediate pairs with the same key *I* and feeds these to the reduce function. **(2)** ***Reduce***: takes an intermediate key *I* and a set of values associated with it and "merges" these values together and outputs zero or more output pairs.

In addition, the user supplies the *specifications* for the MapReduce job to be run, for example specifying the required memory, disk, the number of machines, etc. The canonical example for MapReduce is counting how many times each word appears in a document (see Figure 7.1). The *Map* function is called for every line of the input document, where the key

might be the line number and the value is a string containing that line of text. For every word in the line, it emits an intermediate pair, with the word as the key and a value of "1". MapReduce takes care of grouping all intermediate pairs with the same key (word), and presents them to the *Reduce* function, which just sums up the values for every key (word) and outputs a pair with the word as the key and number of times that word appeared as the value.

We use MapReduce because it has been proven successful and used in a lot of companies, most notably Google, Yahoo!, and Facebook. It is simple, reliable, and scalable [16]. The user provides two functions, and the infrastructure takes care of all intermediate processing, key sorting, inter-machine communications, machine failures, job restarts, etc. It can easily scale up to tens of thousands of machines. Furthermore, there is already a strong open source implementation, called Hadoop, that is widely used.

7.3 Distributed Kd-Tree (DKdt)

The basic Full Representation image search approach and the Kd-Trees method have been discussed in Chapter 2 (see Section 2.5 and Algorithms 2.1, 2.2, and 2.3). In this chapter, we are interested in parallelizing Kd-Tree over a number of machines, because the number of images exceeds the memory capacity of one machine. We explore two ways to parallelize Kd-Trees (see Figure 3.2 in Section 3.3.2, which is reproduced in Figure 7.2 for convenience):

1. **Independent Kd-Trees (IKdt)**: The simplest way of parallelization is to divide the images into independent *chunks*, where each chunk can fit in the memory of one

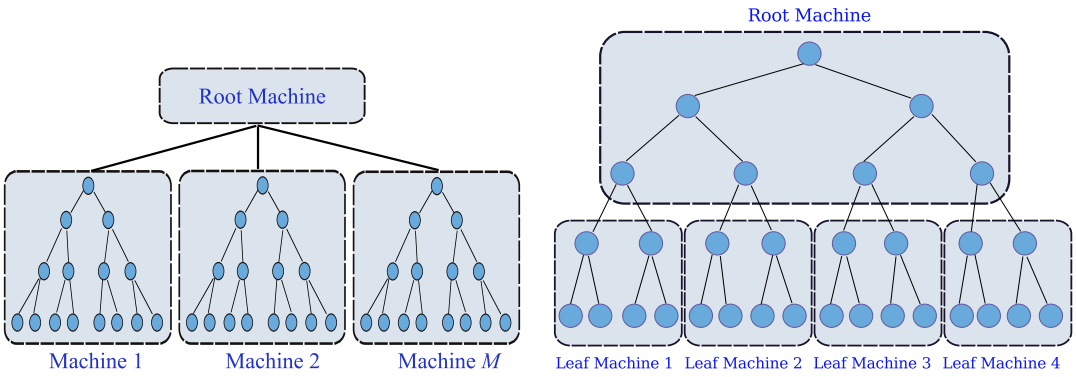


Figure 7.2: **Kd-Tree Parallelizations**. (*Left*) *Independent Kd-Tree (IKdt)*. The database images are divided evenly into M machines, each building its own independent Kd-Tree. At query time, a *root* machine accepts the query image, and directs the features into all the machines. It then accepts the query results and combines them into the final matches. (*Right*) *Distributed Kd-Tree (DKdt)*. The *root* machine stores the top of the tree, while the *leaf* machines store the bottom subtrees of the tree, including the leaves. At query time, the root machine directs features to a subset of the leaf machines, which leads to higher throughput. See Section 7.3.

machine. Then each machine builds an independent Kdt for its chunk of images. A single *root* machine accepts the query image, and passes the query features to all the machines, which then query their own Kdt. The *root* machine then collects the results, performs the counting, and outputs the final sorted list of images.

2. **Distributed Kd-Trees (DKdt)**: Build just one Kdt, where the top of the tree resides on a single machine, the *root* machine. The bottom part of the tree is divided among a number of *leaf* machines, which also store the features that end up in leaves in these parts. At query time, the root machine directs the features into the appropriate leaf machines depending on where they exit the tree on the root machine. The leaf machines compute the nearest neighbors within their subtree and send them back to the root machine, which performs the counting and outputs the final sorted list of images.

The most obvious advantage of DKdt is that a single feature will only go to a small subset of the leaf machines, and thus the leaf machines will be processing multiple features at the same time. This is justified by the fact that most of the computations are performed in the leaf machines [3]. The root machine might become a bottleneck when the number of leaf machines increases, and this can be resolved by having multiple copies of the root (see Section 7.5 and Figure 7.8). The two main challenges with DKdt are: (a) how to build a Kd-Tree that contains billions of features since it does not fit on one machine, and (b) how to perform backtracking in this distributed Kdt.

We solve these two problems by noticing the properties of Kd-Trees: (a) We do not build the Kdt on one machine, we rather build a feature "distributor", that represents the top part of the tree, on the root machine. Since we can not fit all the features in the database in one machine, we simply subsample the features and use as many as the memory of one machine can take. This does not affect the performance of the resulting Kdt since computing the means in the Kdt construction algorithm subsamples the points anyway. (b) We only perform backtracking in the leaf machines, and not in the root. To decide which leaf machines to go to, we test the distance to the split value, and if it is below some threshold S_t, we include the corresponding leaf machine in the process.

The MapReduce architecture for implementing both IKdt and DKdt is shown in Figure 7.3. It proceeds in two phases:

1. **Training Phase**: The Feature MapReduce directs the training features into the different machines, which then build the Kdts with the features assigned to it during the

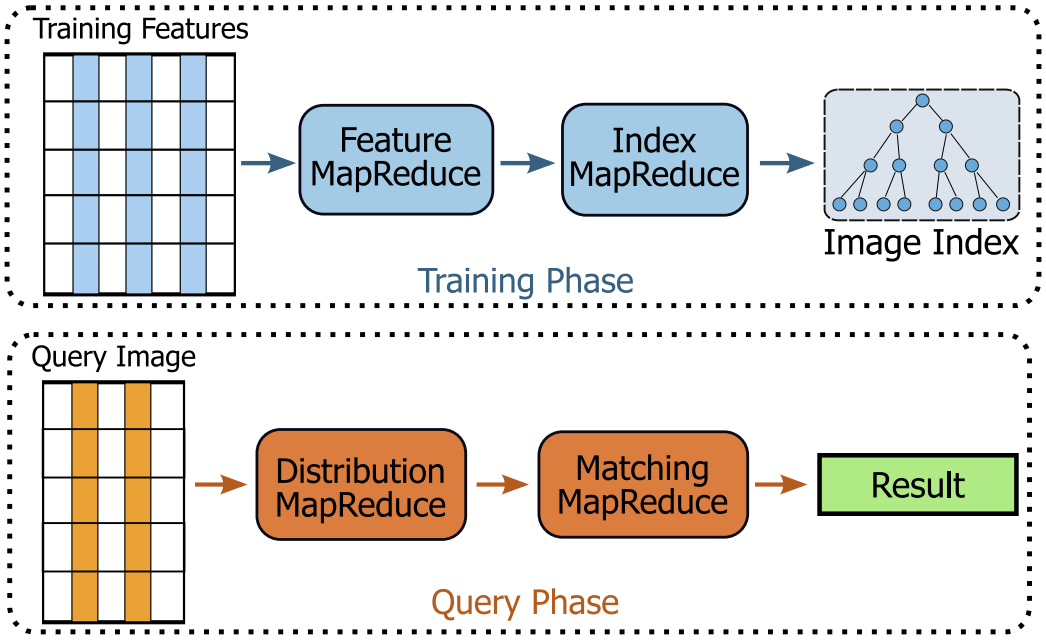


Figure 7.3: **Parallel Kd-Tree MapReduce Schematic**. In the *training phase*, the Feature MapReduce distributes the features among the different machines, which then build the different Kdts during the Index MapReduce. In the *query phase*, the query image is first routed through the Distribution MapReduce, which routes the query features into the appropriate machines, whose results are then picked up by the Matching MapReduce, that queries the respective Kdts and outputs the results. See Section 7.3.

Index MapReduce.

2. **Query Phase**: The Distribution MapReduce directs the query features into the appropriate machines, which perform the Matching MapReduce.

The implementation of IKdt with MapReduce is straightforward (see Algorithm 7.1*(left)*). At training time, the Feature MapReduce is empty, while the Index MapReduce builds the independent Kd-Trees from groups of images, where the Map distributes features according to the image id, and the Reduce builds the Kdt with the features assigned to every machine. At query time, the Distribution MapReduce dispatches the features to all the M Kdts (machines). The Matching MapReduce searches the Kdts on each machine in the Map

and performs the counting and sorting in the Reduce.

The implementation of DKdt is outlined in Algorithm 7.1*(right)*. The notable difference from IKdt is the Feature MapReduce, which builds the top of the Kd-Tree. Given M machines, the top part of the Kdt should have $\lceil \log_2 M \rceil$ levels, so that it has at least M leaves. The Feature Map subsamples the input features by emitting one out of every input $skip$ features, and the Feature Reduce builds the Kdt with those features. The Index MapReduce builds the M bottom parts of the tree, where the Index Map directs the database features to the Kdt that will own it, which is the first leaf of the top part that the feature reaches with depth first search. The Index Reduce then builds the respective leaf Kdts with the features it owns. At query time, the Distribution MapReduce dispatches the query features to zero or more leaf machines, depending on whether the distance to the split value is below the threshold S_t. The Matching MapReduce then performs the search in the leaf Kdts and the counting and sorting of images, as in IKdt.

7.4 Experimental Setup

We use Scenario 2 from Chapter 4 (see Table 4.2). Specifically,we use the *Pasadena Buildings* dataset, and for the distractor set we use *Flickr Buildings*. However, since that distractor set goes only up to 1 M images, we downloaded from the Internet a set of ~ 100 million images searching for landmarks. So in total we have over 100 M images in the database, with 625 query images. The first 1 M images have on average 1800 features each, while the rest of the 100 M images have 500 features on average. The total number of features for all the images is ~ 46 billion features. We report the performance as precision@k, i.e.,

Algorithm 7.1 Parallel Kd-Trees with MapReduce

Independent Kd-Tree (IKdt)

```
Feature Map(key, val)
// nothing
Feature Reduce(key, vals)
// nothing
Index Map(key, val)
Emit(val.imageid mod M, val.feat);
Index Reduce(key, vals)
index = BuildIndex(vals);
Emit(key, index);
Distribution Map(key, val)
for (i = 0; i < M; ++i)
 Emit(i, val);
Distribution Reduce(key, vals)
// nothing
Matching Map(key, val)
nn = SearchNN(val.feat);
Emit(val.imageid, nn);
Matching Reduce(key, vals)
matches = Match(vals);
Emit(key, matches);
```

Distributed Kd-Tree (DKdt)

```
Feature Map(key, val)
Emit(val.id mod skip, val.feat);
Feature Reduce(key, vals)
top = BuildTree(vals);
Emit(key, top);
Index Map(key, val)
indexId = SearchTop(val.feat);
Emit(indexId, val.feat);
Index Reduce(key, vals)
index = BuildIndex(vals);
Emit(key, index);
Distribution Map(key, val)
indexIds = SearchTop(val.feat, S_t);
for id in indexIds:
 Emit(id, val.feat);
Distribution Reduce(key, vals)
// nothing
Matching Map(key, val)
nn = SearchNN(val.feat);
Emit(val.image id, N);
Matching Reduce(key, vals)
matches = Match(vals);
Emit(key, matches);
```

the percentage of the queries that had the ground truth matching image in the top k returned images. Specifically, precision@$k = \frac{\sum_q \{r_q \leq k\}}{\#queries}$ where r_q is the resulting rank of the ground truth image for query image q and $\{x\} = 1$ if $\{x\}$ is *true*. We wish to emphasize at this point that the Pasadena Buildings dataset is very challenging and that no method scores even near 100% correct on this dataset even when tested within a small database of $10^3 - 10^4$ images (see Section 4.4).

For both IKdt and DKdt, we fix the budget for doing backtracking for every feature, and this is shared among all the Kd-Trees searched for that feature. So for example, in the case of DKdt with a budget of 30 K backtracking steps, if a feature goes to two leaf machines, each will use B =15 K, while if ten machines are accessed each will use B =3 K. For IKdt with M machines, each machine will get B/M backtracking steps. This decision was made

to have a fair comparison between IKdt and DKdt in the sense of fixing the CPU cycles used in searching the Kd-Trees in both of them. The CPU time measurements in Section 7.5 count the matching time excluding the time for feature generation for the query images.

We use SIFT [24] feature descriptors (128 dimensions) with hessian affine [25] feature detectors. We used the binary available from http://tinyurl.com/vgg123. We implemented IKdt and DKdt using the proprietary MapReduce infrastructure at Google. The number of machines ranged from 8 (for 100 K images) up to 2048 (for 100 M images). The memory per machine was limited to 8 GB.

7.5 Experimental Results

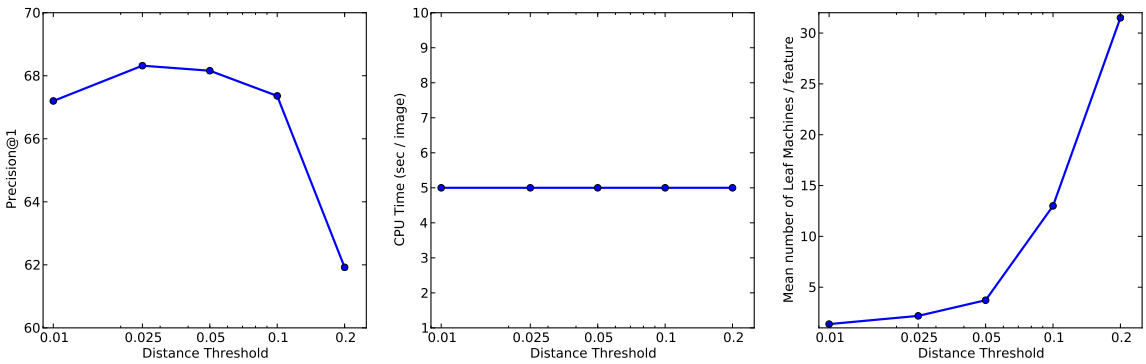


Figure 7.4: **Effect of Distance Threshold** S_t. The X-axis depicts the distance threshold S_t which controls how many leaf machines are queried in DKdt (see Section 7.3). The Y-axis depicts precision@1 (*left*), CPU time (*center*), and mean number of leaf machines accessed per feature (*right*), using 1 M images. We note that using a bigger threshold leads to accessing more leaf machines, and since B is fixed, the leaf machines are not explored enough which results in worse performance. See Section 7.5.

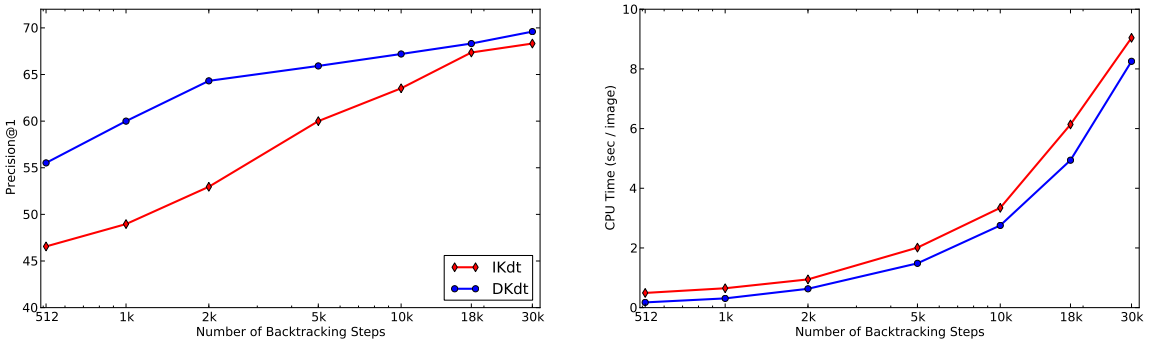


Figure 7.5: **Effect of Backtracking Steps** B. The X-axis depicts the number of backtracking steps B which controls how deep the Kd-Trees are searched, and consequently the CPU time it takes (see Section 7.3). The Y-axis depicts precision@1 (*left*), and total CPU time (*right*), using 1 M images. S_t was set to 0.025 for DKdt. We note that, for the same value of B, DKdt is better than IKdt in terms of both recognition performance and CPU time. See Section 7.5.

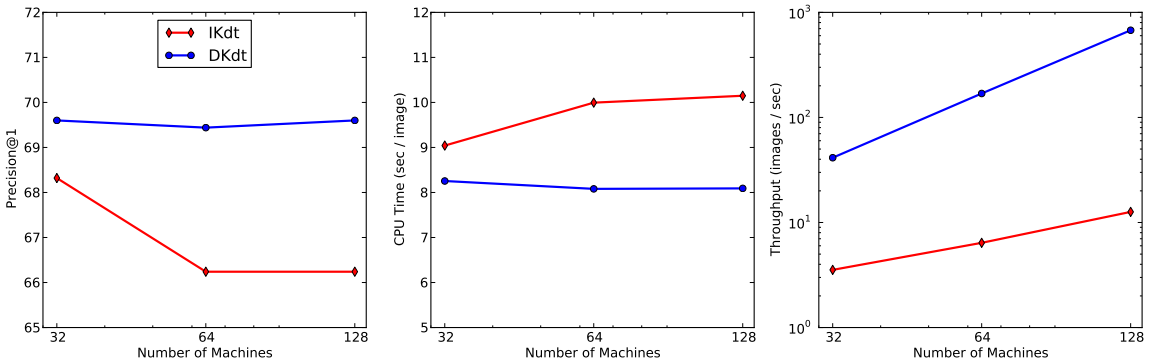


Figure 7.6: **Effect of Number of Machines** M. The X-axis depicts the number of machines used to build the system M. The Y-axis depicts precision@1 (*left*), CPU time (*center*), and Throughput (*right*), using 1 M images. We note the the recognition performance of DKdt remains almost constant with different number of machines, while the throughput increases. That is not the case for IKdt, whose performance decreases when using more machines, since B is fixed. See Section 7.5.

7.5.1 System Parameters Effect

We first explore, using 1 M distractor images from *Flick Buildings*, the different parameters that affect the performance of the distributed Kd-Trees: distance threshold S_t, the number of backtracking steps B, and the number of machines M. Figure 7.4 shows the effect of using

different values for the distance threshold S_t, which affects how many leaf machines are searched at query time when using DKdt (see Section 7.3). The CPU time counts the sum of the computational cost on all the machines, and stays almost constant with increasing number of machines since we have a fixed budget for backtracking steps B (see Section 7.4). The best tradeoff is with $S_t = 0.025$, which gives ~ 3 leaf machines per feature, while using a bigger S_t means more leaf machines are queried and each will not be explored deep enough.

Figure 7.5 shows the effect of the number of the total backtracking steps B. DKdt is clearly better than IKdt for the same B, since it explores less Kdts but goes deeper into them, unlike IKdt which explores all Kdts but with lower B. Figure 7.6 shows the effect of the number of machines M used to build the system. For DKdt, this defines the number of levels in the top of the tree, which is trained in the Feature MapReduce, Section 7.3. For IKdt, this defines the number of groups the images are divided into. For the same number of machines, DKdt is clearly superior in terms of precision, CPU time, and throughput. In particular, with increasing the number of machines, the CPU time of DKdt stays almost constant while that of IKdt grows, because despite B being distributed over the all the machines, the features still need to be copied and sent to all the machines, and this memory copy consumes a lot of CPU cycles (see Figure 7.7). We also note that the throughput increases with the number of machines, and that DKdt has almost 100 times that of IKdt.

7.5.2 Results and Discussion

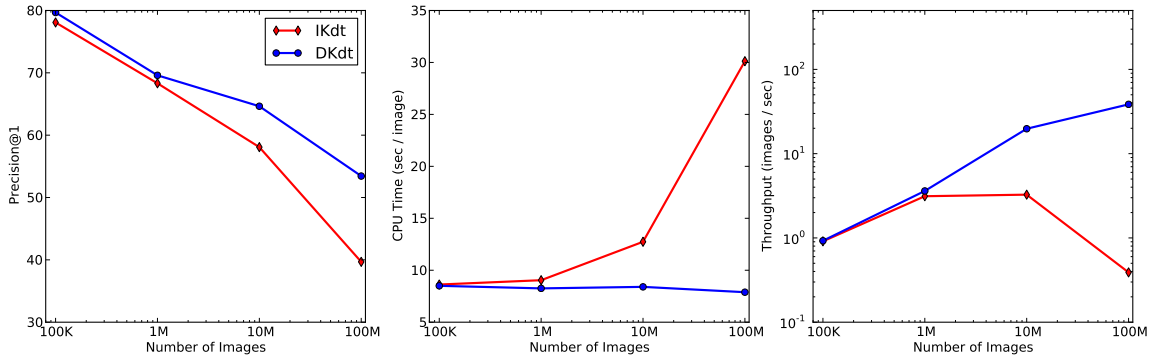


Figure 7.7: **Effect of the Number of Images**. The X-axis depicts the number of images in the database. The Y-axis depicts precision@1 (*left*), CPU time (*center*), and Throughput (*right*). We note the superiority of DKdt over IKdt in terms of both recognition performance and throughput. For 100 M images, DKdt has 30% more precision@1 than IKdt, with 100 times more throughput. See Section 7.5.

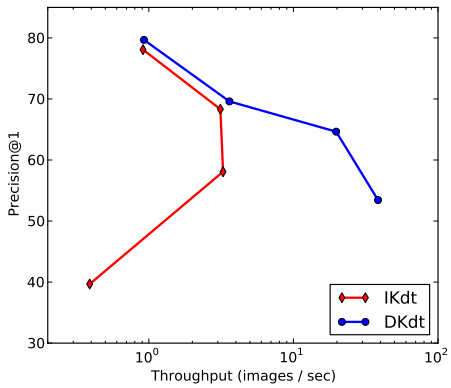


Figure 7.8: **Precision@1 Vs. Throughput**. Every point represents one database size, from 100 K up to 100 M, going from the top left to the bottom right. DKdt is clearly much better than IKdt. See Section 7.5.

Figure 7.7 shows the effect of the number of images indexed in the database. We used

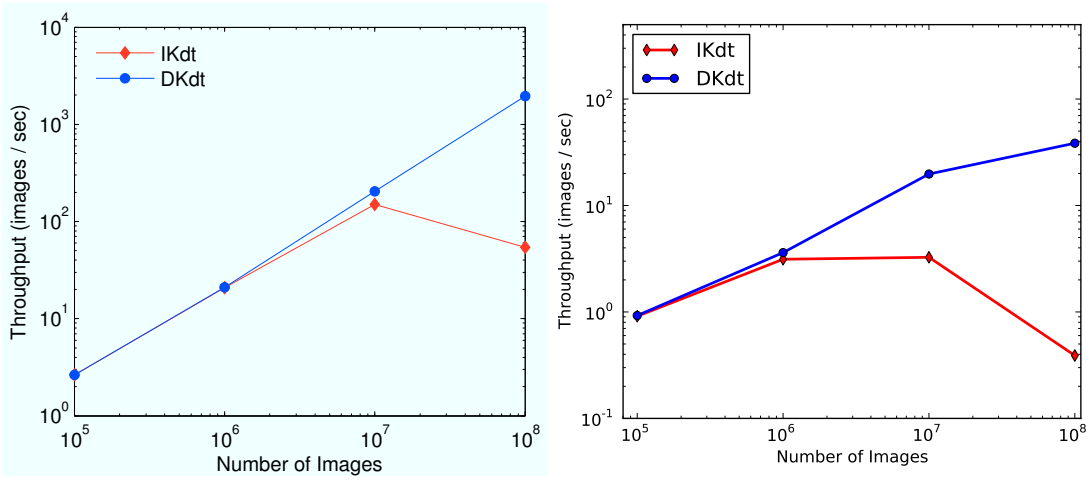


Figure 7.9: **Throughput: Theory Vs. Practice**. The X-axis depicts the number of images in the database, while the Y-axis depicts the theoretical estimate of the throughput (*left*) and the experimental throughput (*right*). We notice general agreement of the theoretical estimates with the experimental measurements. See Section 7.5.

8 machines for 100 K images, 32 for 1 M images, 256 for 10 M images, and 2048 for 100 M images. DKdt clearly provides superior precision to IKdt, with lower CPU cost and much higher throughput. For 100M images, DKdt has precision about 32% higher than IKdt (53% vs. 40%), with throughput that's about 30 times that of IKdt (~ 12 images/sec vs. ~ 0.4), i.e., processes images in a fraction of a second. Figure 7.8 shows another view of the precision vs. the throughput. It is clear that by increasing the number of images, the precision goes down. Paradoxically, the throughput goes up with larger databases, and this is because we use more machines, and in the case of DKdt, this allows more interleaving of computation among the leaf machines and thus more images processed per second. Figure 7.9 shows a comparison of the theoretical estimate of the throughput of the distributed system vs. the experimental measurements. We notice a general agreement between the estimates and the measurements, except for the experimental measurements being more than an order of magnitude lower, which we attribute to other practical considerations, e.g.,

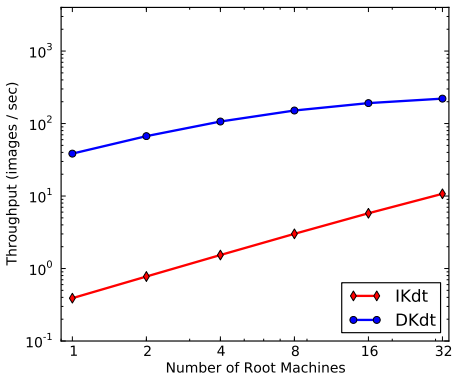


Figure 7.10: **Throughput Vs. the Number of Root Machines**. The X-axis shows the number of root machines used, while the Y-axis shows the throughput for a database of 100 M images. We note that the throughput multiplies by increasing the number of root machines, which provide multiple entry points to the recognition system. See Section 7.5.

cache effects, memory copying, operating system overhead, etc.

We note a drop in the throughput after some point with adding more machines, this is because the computations cannot all be parallelized. The *root* machine accepts the query image, computes the features, and dispatches them to the *leaf* machines that hold the Kd-Trees. It then gets back the results and performs the final scoring. While the Kdt search is parallelized, the other computations are not, and by adding more leaf machines, the bottleneck of the root machines starts decreasing the throughput. Figure 7.10 shows how the throughput increases when adding replicas of the root machine (using 100 M images), which provide multiple entry points for the system and allow more images to be processed at the same time. The throughput grows with the number of root machines added, for example growing to ~ 200 images/sec for DKdt with 32 machines vs. ~ 10 images/sec with 1 root machine.

The precision@1 for 100M images for DKdt might seem disappointing, standing at about 53%. However, users usually do not just look at the top image, they might also

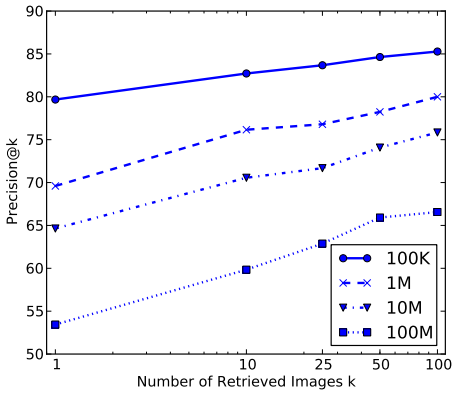


Figure 7.11: **Precision@k**. The X-axis shows the number of retrieved images k, while the Y-axis shows the precision at that value of k, for DKdt. We note that for 100 M images, the precision goes from 53% @1 to 63% @25 and up to 67% @100. See Section 7.5.

examine the top 25 results, which might constitute the first page of image results. Figure 7.11 shows the precision@k for different values of k, ranging from 1 to 100. For 100M images, the precision jumps from 53% @1 to 63% @25 and up to 67% @100 retrieved images. This is remarkable, considering we are looking for 1 image out of 100M images, i.e., probability of hitting the correct image by chance is 10^{-8}. One more thing to note is that all the experiments were run with one Kd-Tree, and we anticipate that using more trees will give higher precision values at the expense of more storage per tree, see [26, 3].

Finally, Figure 7.12 shows results for combining the Distributed Kd-Trees of this chapter with the Compact Kd-Trees of Chapter 5. It shows different compression values, namely 10-fold compression using 64 bits per feature, 8-fold compression using 96 bits, and 6-fold compression using 128 bits (see Table 5.2). With 96 bits, we can get within 22% of the recognition performance of using the full features with only one eighth of the storage, i.e., we can use one eighth of the number of machines. With 128 bits, we get within %13 of the recognition performance of using the full features with only one sixth of the storage. This

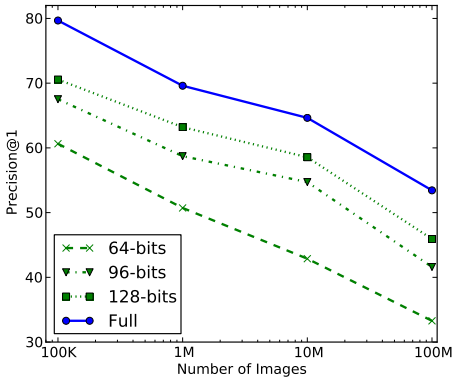


Figure 7.12: **Compact Distributed Kd-Trees**. The X-axis shows the number of images in the database, while the Y-axis shows the precision@1. The different curves show different compression values for Compact Distributed Kd-Trees: using 64 bits, 96 bits, and 128 bits per feature. See Section 7.5.

demonstrates the power of CompactKdt, where, with a fraction of the storage, we can lose only a fraction of the performance.

7.6 Summary

In this chapter, we explored parallel Kd-Trees for scaling up the image search problem to millions of images. We presented implementations of two ways, Independent Kd-Tree and Distributed Kd-Tree, to parallelize Kd-Trees using the MapReduce architecture. We compared the two methods and ran experiments on databases with up to 100 M images. We showed the superiority of DKdt which, for 100 M images, has over 30% more precision than IKdt and at the same time over 30 times more throughput, and processes a query image in a fraction of a second.

Chapter 8

Conclusions

In this thesis we explored the problem of image search in large-scale image collections. We focused on the specific instance of searching collections of DVD covers and building images (Chapter 2). We started with a thorough benchmark of the two leading approaches, both theoretically estimating the computational and memory requirements (Chapter 3) and experimentally comparing their recognition performance and run time (Chapter 4). We then looked at the shortcomings of these methods, and worked on reducing the memory requirements of Kd-Trees (Chapter 5) and boosting the recognition performance of Bag of Words (Chapter 6). We finished with large-scale experiments, building a distributed recognition system with 100 million images using over two thousand machines (Chapter 7).

In this thesis we challenged the mainstream in large-scale object recognition that favors working on Bag of Words since it requires less storage. We showed that Full Representation provides much better performance, even with comparable memory requirements. FR allows us to build recognition systems that scale to hundreds of millions of images and return the result in a fraction of a second.

We believe there is room for improvement in reducing the memory requirements and

run time, and increasing the precision of such large-scale systems. For example:

- Identify the important local features in the image and getting rid of useless features that do not contribute to the search process, which can save both memory and run time.
- Improve local feature representation, both in terms of extraction speed and specificity.
- Devise better compression techniques for features to reduce storage.
- Come up with better shape descriptors to generalize to texture-less objects, e.g., spoons, smooth spheres, etc.
- Use more than one Kd-Tree in the Distributed Kd-Trees system, which could boost the precision at the expense of using more memory.

We retierate the contributions of the thesis:

1. We provide a comprehensive comparison of the two leading image indexing approaches: Full Representation and Bag of Words. In particular, we provide:
   (a) Theoretical estimates of the memory requirements, computational cost, and parallelizability of these methods as a function of the number of images.
   (b) Experimental evaluation of these different methods on four real world datasets. We report the recognition performance and run time as the number of images grows.

2. We challenge the conventional wisdom in image indexing methods. We argue that FR approach is the way to go, since although it requires an order of magnitude more storage, it provides superior recognition performance to BoW, especially with large datasets.

3. We present novel methods to remedy some of the shortcomings of these two methods:

   (a) Compact Kd-Trees that are able to cut the memory usage and run time of FR methods by an order of magnitude while achieving comparable recognition performance.

   (b) Multiple Dictionaries for BoW that are able to significantly boost the recognition performance of BoW methods to levels comparable to FR methods, at the expense of increased memory and computational costs.

4. We present a novel way of parallelizing Kd-Trees, Distributed Kd-Trees, and run experiments on thousands of machines with 100 million images. The system outperforms the state-of-the-art in both recognition performance and throughput, and can process a query image in a fraction of a second.

Bibliography

[1] Mohamed Aly. Online Learning for Parameter Selection in Large Scale Image Search. In *CVPR Workshop OLCV*, June 2010.

[2] Mohamed Aly, Mario Munich, and Pietro Perona. Bag of words for large scale object recognition: Properties and benchmark. In *International Conference on Computer Vision Theory and Applications (VISAPP)*, March 2011.

[3] Mohamed Aly, Mario Munich, and Pietro Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *WACV*, 2011.

[4] Mohamed Aly, Peter Welinder, Mario Munich, and Pietro Perona. Scaling object recognition: Benchmark of current state of the art techniques. In *ICCV Workshop WS-LAVD*, 2009.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[6] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.

[8] Andrei Broder, Moses Charikar, and Michael Mizenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:630–659, 2000.

[9] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29:8–13, 1997.

[10] A.Z. Broder. On the resemblance and containment of documents. In *Proc. Compression and Complexity of Sequences 1997*, pages 21–29, 1997.

[11] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. of 34th STOC*. ACM, 2002.

[12] O. Chum, J. Philbin, M. Isard, and A. Zisserman. Scalable near identical image and shot detection. In *CIVR*, pages 549–556, 2007.

[13] O. Chum, J. Philbin, and A. Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *British Machine Vision Conference*, 2008.

[14] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[15] N. Dalai and B. Triggs. Histograms of oriented gradients for human detection. In *CVPR*, volume 1, pages 886–893vol.1, 20-25 June 2005.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[17] Matthijs Douze, Hervé Jégou, Harsimrat Sandhawalia, Laurent Amsaleg, and Cordelia Schmid. Evaluation of gist descriptors for web-scale image search. In *International Conference on Image and Video Retrieval*. ACM, july 2009.

[18] D. Forsyth and J. Ponce. *Computer Vision: A modern approach*. Prentice Hall, 2002.

[19] J. M. Geusebroek, G. J. Burghouts, and A. W. M. Smeulders. The amsterdam library of object images. *IJCV*, 61:103–112, 2005.

[20] H. Jégou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *ECCV*, 2008.

[21] H. Jégou, M. Douze, and C. Schmid. Packing bag-of-features. In *ICCV*, sep 2009.

[22] H. Jégou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. In *CVPR*, 2010.

[23] Y. Ke, R. Sukthankar, and L. Huston. An efficient parts-based near-duplicate and sub-image retrieval system. In *MULTIMEDIA*, pages 869–876, 2004.

[24] David Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[25] K. Mikolajczyk and C. Schmid. Scale and affine invariant interest point detectors. *IJCV*, 2004.

[26] M. Muja and D. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP*, 2009.

[27] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. *CVPR*, 2006.

[28] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *IJCV*, 42:145–175, 2001.

[29] F. Perronnin, Y. Liu, J. Sánchez, and H. Poirier. Large-scale image retrieval with compressed fisher vectors. In *CVPR*, 2010.

[30] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. *CVPR*, 2007.

[31] M. Raginsky and S. Lazebnik. Locality sensitive binary codes from shift-invariant kernels. In *NIPS*, 2009.

[32] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

[33] T. Terasawa and Y Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. *Proceedings of the Workshop on Algorithms and Data Structures*, 2007.

[34] P. Turcot and D. Lowe. Better matching with fewer features: The selection of useful features in large database recognition problems. In *ICCV Workshop WS-LAVD*, 2009.

[35] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, 2008.

[36] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 2006.

Index