Scalable Nearest Neighbor Search with Compact Codes

by

Sepehr Eghbali

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Miriam Capretz, Professor Department of Electrical and Computer Engineering Western University
Supervisor:	Ladan Tahvildari, Professor Department of Electrical and Computer Engineering University of Waterloo
Internal Member:	Otman Basir, Professor Department of Electrical and Computer Engineering University of Waterloo
	Mark Crowley, Assistant Professor Department of Electrical and Computer Engineering University of Waterloo
Internal-External Member:	Olga Vechtomova, Associate Professor Department of Management Sciences University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Parts of this dissertation are based on some publications that I have co-authored. In particular, Chapter 3 is based on a joint work with Ladan Tahvildari [40]. Also, Chapter 4 is based on two joint works with Hassan Ashtiani and Ladan Tahvildari [37, 38]. Finally, Chapter 5 is based on a joint work with Ladan Tahvildari [39].

Abstract

An important characteristic of the recent decade is the dramatic growth in the use and generation of data. From collections of images, documents and videos, to genetic data, and to network traffic statistics, modern technologies and cheap storage have made it possible to accumulate huge datasets. But how can we effectively use all this data? The growing sizes of the modern datasets make it crucial to develop new algorithms and tools capable of sifting through this data efficiently. A central computational primitive for analyzing large datasets is the Nearest Neighbor Search problem in which the goal is to preprocess a set of objects, so that later, given a query object, one can find the data object closest to the query. In most situations involving high-dimensional objects, the exhaustive search which compares the query with every item in the dataset has a prohibitive cost both for runtime and memory space. This thesis focuses on the design of algorithms and tools for fast and cost efficient nearest neighbor search. The proposed techniques advocate the use of compressed and discrete codes for representing the neighborhood structure of data in a compact way. Transforming high-dimensional items, such as raw images, into similarity-preserving compact codes has both computational and storage advantages as compact codes can be stored efficiently using only a few bits per data item, and more importantly they can be compared extremely fast using bit-wise or look-up table operators. Motivated by this view, the present work explores two main research directions: 1) finding mappings that better preserve the given notion of similarity while keeping the codes as compressed as possible, and 2) building efficient data structures that support non-exhaustive search among the compact codes. Our large-scale experimental results reported on various benchmarks including datasets upto one billion items, show boost in retrieval performance in comparison to the state-of-the-art.

Acknowledgements

I want to express my sincere appreciation to my adviser, Professor Ladan Tahvildari, for her constant support, encouragement, patience, and for providing me with the right set of tools throughout this work. She gave me the freedom to follow my own interests while guiding my ideas towards a coherent thesis.

I would like to thank my thesis committee members - Professor Miriam Capretz, Professor Olga Vechtomova, Professor Otman Basir, and Professor Mark Crowley - for their useful comments on this work.

I also want to especially thank Hassan Ashtiani, who is a collaborator in parts of this dissertation, for the helpful discussions and brainstorming sessions towards solving research problems.

I had a great time in Waterloo with my incredible friends who played an important role in making my life more enjoyable. Thank you for all the marvelous time together and supporting me.

Above all, my deepest gratitude I devote to my family who laid the foundations of all of this for me. The seeds of their contribution have always permeated my entire life and will continue to do so. No words will be enough to express all my appreciation for their support and everything they have done for me.

Dedication

To my lovely parents.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AMIH Angular Multi-Index Hashing

ANN Approximate Nearest Neighbor

AQ Additive Quantization

AQBC Angular Quantization-based Binary Codes

AVQ Augmented Vector Quantization

CDR Compact Discrete Representation

CKM Cartesian K-Means

CQ Composite Quantization

DQN Deep Quantization Network

DSQ Deep Spherical Quantization

FALCONN FAst Lookups of Cosine and Other Nearest Neighbors

HWT Hamming Weight Tree

ILS Iterated Local Search

KNN K Nearest Neighbor Search

LSH Locality Sensitive Hashing

LSQ Local Search Quantization

LSQ++ Local Search Quantization++

MCQ Multi-Codebook Quantization

MIH Multi-Index Hashing

MIPS Maximum Inner Product Search

NNS Nearest Neighbor Search

OPQ Optimized Product Quantization

PQ Product Quantization

RVQ Residual Vector Quantization

SCO Sparse Composite Quantization

SDSQ Sparse Deep Spherical Quantization

SQ Supervised Quantization

SUBIC SUpervised structured BInary Code

UNQ Unsupervised Neural Quantization

VQ Vector Quantization

List of Symbols

$\mathcal{B}$ set of binary codes

B matrix of binary codes

C_j j-th codebook matrix

C codebook matrix

K number of nearest neighbors to retrieve

$\mathbf{b}_i$ i-th binary vector

$\mathcal{X}$ set of input samples

δ distance function

d number of dimensions

$h(.)$ hash function

$\|.\|_H$ Hamming norm

$\mathbf{x}_i$ i-th sample

k number of codewords (per codebook)

m number of codebooks or number of hash tables

n number of input points

p length (dimension) of binary codes

$\mathbf{q}$ query vector

$r_{1\to 0}^{\mathbf{q},\mathbf{b}_i}$ number of bit positions in which $\mathbf{q}$ is 1 and $\mathbf{b}_i$ is 0

$r_{0\to 1}^{\mathbf{q},\mathbf{b}_i}$ number of bit positions in which $\mathbf{q}$ is 0 and $\mathbf{b}_i$ is 1

r radius of search

Chapter 1

Introduction

1.1 Motivation

We live an era where most of our activities are increasingly leaving a digital trace. Consumer-grade cameras, such as smart-phone cameras, take images or records videos, GPS trackers record our movements and vehicle sensors such as lidar and sonar capture the surroundings dynamic to navigate safely. In addition to the unprecedented rate of data generation, access to large collections of data has been made effortless. This can be easily detected by the growing number of available large-scale datasets on the Internet which have become the standard benchmark for a broad array of data-driven tasks. Instances of large-scale datasets include the Google n-gram dataset obtained from over 5 million books, Imagenet [33] with more than 14 million annotated images spanning more than 20,000 categories, JFT-300M [120] with more than 300 million images and YouTube-8M [1] with more than 6.1 million annotated videos.

Now that we have this data, terabytes and petabytes of it, what do we do with it? What are the basic computational problems that we want to solve on large datasets? Broadly speaking, we need to read, store, analyse and search through this data in order to extract the most relevant information to solve problems. The prospect of exploring sheer volume of data for extracting information is interesting. In machine learning, particularly, it has been shown that large training datasets beside strong computational power are the keys for good performance. For example, the performance improvement achieved by using a very large training set in [123] emphasizes that large annotated datasets fuel progress in object recognition. Other research papers reiterate in many places how large training sets helped in overcoming the problem of overfitting [122].

There is no doubt that search has been one of the most successful applications of information technology today which, not surprisingly, has been mostly concerned with large collections in which both the number of data items and features are huge. A fundamental and extensively studied subclass of search problems is the Nearest Neighbor Search (NNS) in which unseen queries are matched against a given dataset of items to find the closest item to the query. NNS has been a key problem in theoretical and applied computer science with a broad range of applications in data processing and analysis including machine learning, document retrieval, data compression and bio-informatics.

Despite decades of intense research, performing NNS has been typically a bottleneck in large-scale applications, mainly because the current similarity search techniques do not easily scale to more than several million data points, where storage overheads and similarity computations become prohibitive. For example, the majority of center-based clustering algorithms such as k-means need to execute multiple instances of nearest neighbor queries in each iteration. The problem is that the number of query instances often grows linearly with the size of input, thus without a scalable solution, the nearest neighbor search can quickly become the computational bottleneck of clustering.

1.2 Problem Description and Scope

This thesis aims at finding efficient solutions for large-scale nearest neighbor search problem. Given a measure of similarity and a set of items, our goal is to explore techniques that can return the subset of relevant results as fast as possible knowing that the size of the dataset is in the order of billions.

Problem Statement: *How to develop efficient similarity search tools and algorithms with minimal memory and computation costs, to facilitate the use of web-scale datasets?*

In the absence of any practical exact similarity search technique that supports large-scale datasets, this dissertation follows the recent direction for solving nearest neighbor search which relies on trading accuracy for scalability, resulting in a rich family of techniques collectively known as the Approximate Nearest Neighbor (ANN). Roughly speaking, in this relaxed class of search, the algorithm is allowed to return points that are close to the query which may not necessarily be the closest one. The focus in this thesis is on exploring new data structures and algorithms for boosting the nearest neighbor search performance without losing much in the accuracy.

1.3 Research Questions

This dissertation advocates utilization of Compact Discrete Representation (CDR); a subclass of ANN, which has witnessed a surge of interest over the last decade as it facilitates fast distance computation with low memory overhead. The main idea of CDR is to map high-dimensional items into similarity-preserving short vectors that can be represented with only a few bits (often at most 256 bits). In this setting, solving the nearest neighbor search among the high-dimensional items is approximated with solving the same problem among the compressed codes. Adopting compact codes can significantly reduce the storage and computational costs as, instead of high-dimensional items, one can just store the compact codes in memory, and more importantly, such short codes can be compared much faster than the original items.

Recent studies in CDR have focused on designing more accurate mappings that generate codes which are more compact and more faithful to the giving notion of similarity. The first half of this thesis, however, takes a complementary approach by developing search techniques for fast retrieval among the generated discrete codes. Moving forward, the thesis studies supervised and unsupervised techniques for producing more efficient codes in order to achieve higher recall rates.

Our approach aims at answering the following two high-level research questions. We prioritize clarity over technical precision here but more rigorous and detailed research questions accompany each separate chapter as new questions and concepts are introduced:

1.3.1 Research Question 1: How can we preprocess the compact discrete codes to achieve fast nearest neighbor search?

It has been shown that performing linear scan for solving NNS (that exhaustively compares the query with every element in the dataset and returns the closets item) even among the compact codes can take up to several minutes [97] which is not acceptable in dealing with realtime applications such as search engines. Although linear scan requires no preprocessing, we are often interested in spending some tolerable preprocessing costs in order to reduce the search time. Chapters 3 and 4 aim at answering this question by proposing two different indexing data structures. The main idea in both approaches is to partition the dataset into different sets such that the query is only compared with a subset of points rather than the whole dataset.

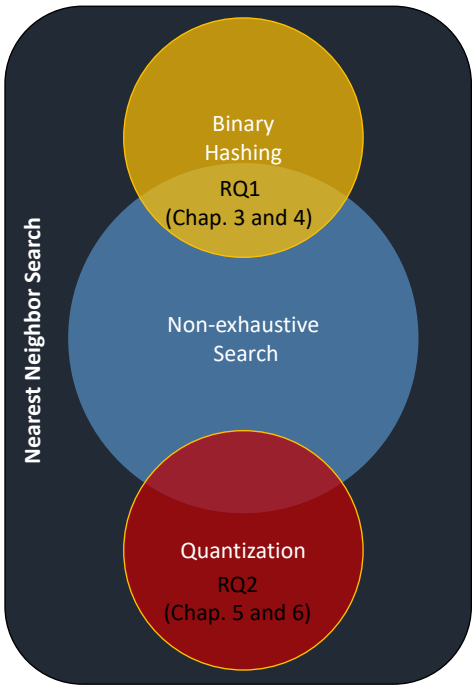


Figure 1.1: Concepts related to thesis and their relationship with research questions.

1.3.2 Research Question 2: How can we design efficient compact code that are more similarity-preserving?

One way to map high-dimensional items into meaningful compact codes is through vector quantization. Chapter 5 and 6 draw upon a family quantization-based techniques, known as *multi-codebook quantizaiton*, and propose two compositional models that, compared to existing models, enjoy easier optimization procedures and better recall rates.

Figure 1.1 shows the concepts related to this thesis and their relationships with discussed research questions.

1.4 Summary of Contributions

In the following, the outline of our contributions within this dissertation is presented:

- **Efficient Search with Multiple Hash Tables**: The problem of exact K nearest neighbor search in a large dataset of compact codes is addressed first. Given a dataset

of compact codes where each item is represented with a short binary string, the goal is to preprocess the data such that when later given a query, the nearest neighbors can be find efficiently. To this aim, we propose to partition each string of binary code into shorter substrings and then build hash tables on binary code substrings. Given the query, a sequential algorithm determines the correct sequence of buckets that must be looked up in each hash table in order to find the exact closest points. Our approach is storage efficient and straight-forward to implement. Theoretical analysis shows that the proposed algorithm exhibits sub-linear behavior for uniformly distributed codes. In addition, the empirical analysis with non-uniformly distributed codes shows dramatic speedups over several baselines for datasets up to one billion codes.

- **Non-exhaustive Search in Dynamic Datasets**: This study concerns the problem of exact dynamic K nearest neighbor search where dynamic means that the dataset is open-ended and items appear over time thus the size and distribution of data are not known beforehand. For this purpose, we propose a tree-search data structure that partitions the input space by exploiting the Hamming weights of codes and their substrings. The proposed data structure constructs a tree over data points in an incremental fashion by routing incoming points to the leaves. We empirically show that the proposed technique can accelerate nearest neighbor search specially when the size of the dataset is large.

- **Supervised Quantization with Deep Networks**: This thesis also extends over the existing supervised quantization techniques and proposes a deep supervised quantization that outputs similarity-persevering compact codes. The proposed approach benefits from a simple objective function with no extra constraints and can be learned end-to-end on the raw input images. The advantages of a simple formulation are not merely aesthetic; in practice, they result in a straightforward optimization procedure and less overhead for the programmer. Furthermore, a more standard formulation might render other variants of the problem easier to solve as well. We demonstrate one such use case, by implementing a formulation of quantization that enforces sparsity on the codebooks. The empirical results on standard image benchmarks show that the proposed technique achieves higher recall rates in comparison to existing supervised quantization techniques.

- **Unsupervised Quantizaiton with a Single Quantizer** We revisit the unsupervised quantizaition problem and propose a new formulation which allows learning unconstrained codebooks. The state-of-art in quatization for nearest neighbor search suggests using two quantizers, one for quantizing the input vectors themselves and a separate quantizer for encoding the norms. Our new formulation discards the norm

quantizer and spend all of its quantization budget on the vector quanizer. This in turn leads to smaller quantization error and thus higher recall rates in practice. This study also derives the first distribution-free generalization bound for the family of multi-codebook quantiaztion techniques which shows that sample complexity grows only polynoimally with the number of codebooks.

1.5 Document Organization

The rest of this thesis is organized as follows. Chapter 2 reviews some background concepts and related works. Chapter 3 details our multi-index hashing approach for solving angular nearest neighbor search. Chapter 4 focuses on solving similarity search in dynamic datasets. Chapter 5 discusses a supervised compositional quantization models useful for fast distance computation. Chapter 6 introduces an unsupervised quantization technique for further reduction of quantization error. Finally, Chapter 7 recaps the contributions and concludes the thesis with a discussion of interesting directions for future research.

Chapter 2

Background

This chapter reviews some background and related literature for research presented in this thesis. The chapter starts with the formal definition of nearest neighbor search and its approximate variant in Section 2.1. Sections 2.2 and 2.3 discuss prior work on data-independent and data-dependent hashing. Finally, Section 2.4 reviews non-exhaustive search techniques for compact codes.

2.1 Nearest Neighbor Search

Nearest neighbor search is defined as follows: Given a dataset $\mathcal{X}$ of n points $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, $\mathbf{x}_i \in \mathbb{R}^d$, and a distance function δ: $\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ preprocess $\mathcal{X}$ to prepare for quickly answering queries of following kind: given a d-dimensional query point $\mathbf{q}$, find the closest data point in $\mathcal{X}$ to the query.

$$NN(\mathbf{q}) = \arg\min_{\mathbf{x} \in \mathcal{X}} \delta(\mathbf{q}, \mathbf{x}) \tag{2.1}$$

The K Nearest Neighbor Search (KNN) is the generalization of NNS that aims at finding the K closests items to the query. An item $\mathbf{x} \in \mathcal{X}$ is in the result of $KNN(\mathbf{q})$ if and only if it satisfies the following condition:

$$|\{t \in X | \delta(\mathbf{t}, \mathbf{q}) < \delta(\mathbf{x}, \mathbf{q})\}| < K \tag{2.2}$$

For notational convenience, often $X \in \mathbb{R}^{d \times n}$ is represented as a matrix, in which all the d-dimensional database vectors $\mathbf{x}_i$ are horizontally stacked.

One can define NNS (and KNN) problem with respect to any distance function. However, the most studied case is when the dataset lie in a d-dimensional vector space $\mathbb{R}^d$ equipped with ℓ_1 or ℓ_2 distances. A useful special case of ℓ_1 scenario is when the dataset and query lie in the hypercube $\{0,1\}^d$. This corresponds to the Hamming distance. An important special case of ℓ_2 setting is when data points lie on a unit sphere $\mathbb{S}^{d-1} \in \mathbb{R}^d$, this is equivalent to the nearest neighbor search with respect to the cosine similarity.

The NNS is one of the prototype proximity problems with a long history in computational geometry and plays an important role in many applications. Perhaps the most obvious one is the similarity search for various types of data such as text documents, images, audio files and protein. A typical approach is to take a dataset and map it into $\mathbb{R}^d$ by computing a certain feature representation. Oftentimes, this step requires significant domain expertise, however the burgeoning deep architectures has equipped us with *feature learning*, automatic learning of needed representation from raw data. Nevertheless, once the feature vectors are computed, the similarity search directly corresponds to executing the nearest neighbor search in the feature space.

The NNS admits a straightforward solution: omit the preprocessing stage and whenever given a query, linearly compute all distances $\delta(\mathbf{q}, \mathbf{x})$ for $\mathbf{x} \in \mathcal{X}$ and keep track of the entry with the minimum distance. However, linear scan ends up being too slow for high-dimensional large datasets as it linearly depends on both the number of dimensions and number of data points, $O(nd)$. Since in many applications n can be large, it was necessary to develop faster methods that find the nearest neighbor without explicitly computing all distances from $\mathbf{q}$ (sublinear in n). Those methods compute and store additional information about set $\mathcal{X}$, which is then used to find nearest neighbors more efficiently. To illustrate this idea, consider another simple solution for the case where points lie in the d dimensional Hamming space, $\{0,1\}^d$. In this case, one could precompute and store in memory the answers to all 2^d possible queries, and given $\mathbf{q}$ return its nearest neighbor by performing only a single memory lookup. Unfortunately, this approach requires memory of size 2^d, which again is intolerable in practice.

These two solutions can be viewed as extreme ends in tradeoff between the time to answer a query (query time), and the amount of memory used (space). The study of this tradeoff dates back to the work of Minsky and Papert [85], and has become one of the key topics in the field of computational geometry. For special case when $d = 1$, sorting data points and performing binary search guarantees finding the exact nearest neighbor in logarithmic time. The case of $d = 2$ can also be efficiently solved using Voronoi diagrams in $O(\log n)$ time and $O(n)$ space. For $d \geq 3$ there is no sublinear algorithm with efficient memory and preprocessing time. Nevertheless, for low-dimensional problems (*e.g.*, with d up to 20) a handful of tree-based techniques, such as *KD-tree* and *ball tree*, manage to

exhibit good practical performance (see [109] for an overview). While some of such space partitioning algorithms guarantee sublinear "expected" search time, no satisfactory worst case performance can be guaranteed.

That said, for relatively high-dimensional data, the exact NNS problem is unsolved both in theory and practice. In particular, despite decades of intensive research, for moderate feature dimensional (*e.g.*, $d > 20$), exact NNS solutions with sublinear (in n) query time requires storage cost that is exponential in the number of dimensions. More specifically, to this day, there is no exact algorithm with polynomial preprocessing and storage costs which guarantees sublinear query time even for simple distances such as Hamming distance [90]. This shows that NNS, like many other proximity problems, suffer from a phenomena known as the *curse of dimensionality*: as number of dimensions grows, all algorithms quickly (both theoretically and practically) degrade to linear search [130].

Due to the lack of success in removing the exponential dependency on the number of dimensions, many researchers conjecture that no efficient solution exists for this problem when the number of dimensions is large. At the same time, researches raised the question: is it feasible to eliminate the exponential dependency on dimensions, if we allow the answers to be *approximate*? Fortunately, this question has been answered positively for many distance measures. In the following, first define the approximate variant of KNN problem is defined and then some important achievements in this line of research are reviewed.

Approximate Nearest Neighbor Search. To overcome the apparent difficulty of devising algorithms that are capable of finding the exact solution of nearest neighbor problem, there has been an increasing interest to resort to ANN search which trades accuracy for scalability.

In ANN formulation, the algorithm is allowed to return an item whose distance from the query is at most c times ($c > 1$) the distance to its nearest neighbor. The problem is often relaxed to do this with high probability. There are two lines of research addressing approximate NNS problem: theoretical (such as [3, 55]) and applied (such as [59]). Theoretical research aims at improving the approximation ratios of the NNS solutions, and their space and worst case query time complexity. In addition, theoreticians try to develop hardness results for NNS under different metrics. Applied research, such as the current thesis, mainly concerns experimental evaluation of techniques, and while it draws inspiration from theory, it does not compare methods based on their worst case query time complexity, but instead based on their average query time performance and retrieval accuracy on standard benchmarks.

2.2 Data-independent Hashing

2.2.1 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a randomized hashing scheme for solving nearest neighbor search. The main building block of LSH is the family of *locality sensitive* hashing functions which map similar items to same (or close) hash values with high probability. This idea was introduced by Indyk and Motwani in 1998 in a seminal paper [55] which was described as a breakthrough in reducing the computational cost of similarity search and circumventing the curse of dimensionality. The problem that they tackled was a variant of ANN that is called (c, r)-ANN. In this problem, the data structure is allowed to return any data point whose distance from the query is at most cr, for an approximation factor $c > 1$. The basic idea of LSH is to map each database vector into a hash value using a hash function randomly chosen from a locality sensitive hash function family, formally defined as follows:

Definition 2.1. Locality Sensitive Function *Given the example space S, a family $\mathcal{H}$ of functions $h(.)$ is called* locality sensitive*, or more specifically (r, c, p_1, p_2) sensitive, if for any $\mathbf{u}, \mathbf{v} \in S$ we have:*

- *if $\delta(\mathbf{u}, \mathbf{v}) < r$ then $Pr(h(\mathbf{u}) = h(\mathbf{v})) \geq p_1$*
- *if $\delta(\mathbf{u}, \mathbf{v}) > cr$ then $Pr(h(\mathbf{u}) = h(\mathbf{v})) \leq p_2$*

where Pr denotes the probability.

In order for a locality sensitive hash family to be useful, it has to satisfy inequality $p_1 > p_2$. Intuitively, a hash function is locality sensitive if its probability of collision is higher for nearby points than for far apart points. The parameter c quantifies the gap between near and far apart points.

Given such an LSH family of hash functions for distance measure dis, for $\rho = \frac{\log(1/p_1)}{\log(1/p_2)}$, there exists an algorithm that solves the (c, r) ANN which uses $O(dn + n^{1+\rho})$ space with query time dominated by $O(n^{\rho})$ distance computation and $O(n^{\rho} \log_{1/p_2} n)$ evaluations of hash functions from $\mathcal{H}$. The parameter ρ governs the search performance, the smaller the ρ, the better search performance [32].

Informally speaking, the main idea is to partition the input space using several hash functions (each corresponds to a separate hash table) from the family $\mathcal{H}$. The goal is

to ensure that the probability of collision is much higher for close points (at distance $< r$) than for those far apart (at distance $> cr$). During preprocesing phase, all the points in the dataset are separately indexed in each of the hash tables, then upon query arrival, one computes its hash keys by applying the same hash functions, thus the query also hashes to certain buckets where it collides with some small portion of the database vectors. Only these examples are searched and returned in ranked order as the output of nearest neighbor search. For sufficiently large number of hash tables, LSH guarantees that the nearest neighbor lies in the resulting output with high probability. Note that LSH is considered data-independent as the hash functions are selected randomly, independent of data distribution. As we will discuss later, it is possible to exploit the distribution of data (by using a training set) in order optimize the parameters of hash function.

To illustrate the concept of LSH, consider the following example.

Example 2.1. Assume that the data items are in binary space, that is we have $\mathbf{u} \in \{0,1\}^p$. Also, assume that Hamming distance is used as the measure of similarity in this space. A simple yet efficient family of hash functions $\mathcal{H}$ contains all the projections of the input point on one of the coordinates. In other words, $\mathcal{H}$ contains all functions h_i from $\{0,1\}^p$ to $\{0,1\}$ such that $h_i(\mathbf{u}) = u^{(i)}$ where $u^{(i)}$ indicate the i-th coordinate of $\mathbf{u}$. Choosing one hash function h at random from $\mathcal{H}$ indicate that $h(\mathbf{u})$ selects one of the coordinate randomly. It is easy to show that $\mathcal{H}$ is a locality sensitive with non-trivial parameters. The probability that $Pr(h(\mathbf{u}) = h(\mathbf{v}))$ is equal to the fraction of points on which $\mathbf{u}$ and $\mathbf{v}$ share the same value of bit. In particular, we have $p_1 = 1 - \frac{R}{p}$ and $p_2 = 1 - \frac{cR}{p}$. Since $c > 1$, we have $p_1 > p_2$ where R is the Hamming distance between the two points. Therefore, random selection of coordinates in the Hamming space is a locality sensitive hash function.

Earlier works in LSH proposed hash functions for NNS on binary codes in Hamming distance. Such methods also extend to Euclidean distance by embedding Euclidean space into Hamming space. Next, Datar *et al.* [32] proposed a LSH scheme that works directly for Euclidean space which was later improved by a follow-up work [2]. There exist a large body of work on designing LSH functions other similarity measures, such as angular distance [25] and Jaccard similarity [16, 17]

LSH is widely studied in theory and extensively applied in different array of fields such as near-duplicate web page detection [16], image retrieval [62], clustering [64], and filtering [31]. The natural question that emerged in the theoretical community is: what is the smallest possible value of ρ achievable via locality sensitive hashing approach? The original paper of LSH [55] proposed hash functions with $\rho < 1/c$ for both Hamming and Euclidean spaces. It is shown that for the case of Hamming space using the random

coordinates hash function actually provides the best possible value of ρ. For the case of Euclidean distance, hash functions with better exponent are achievable [2, 32] such that $\rho \leq 1/c^2$. It has been shown that for the Euclidean case, which turns out to be most widely applied case in practice, this bound is tight [99]. Also in [32], it has been shown that hashing-based algorithms cannot achieve $\rho < 0.462/c^2$. This result indicates that the best possible LSH exponent has been almost settled and making improvements seems impossible.

Many other studies improve or adapt LSH in various aspects. For example, using a prior to design more efficient hash functions [110], using sampled data [36], using LSH on distributed computing platforms [11, 121], and taking advantages of specific hardware [100, 115]. Parameter tuning for LSH is important, and this is discussed in detail in [118].

2.2.2 Limitations of Locality Sensitive Hashing

There are mainly three problems with using LSH techniques in practice:

- From the practical point of view, to achieve acceptable retrieval accuracy usually the number of hash tables should be relatively large [132, 140]. Experimental studies indicate that LSH-based techniques need over a hundred [45] and sometimes several hundred hash tables [18]. Since the size of hash table is proportional to the number of data objects, in some applications, using LSH incurs intolerable memory cost and long query time. Some heuristics to address this issue are studied such as "Entropy-based LSH" [101] and "Mutli-prob LSH" [75] but unlike the basic LSH, they do not provide guarantee on the performance of their algorithms.

- Existing tight bounds for the LSH scheme suggest that improving LSH with the current formulation is not achievable and this family of techniques is almost completely understood [104]. Even this line of research in theoretical computer science communities has recently shifted towards allowing the hash functions to depend on the data [4, 5, 104].

- Instead of curse of dimensionality, LSH suffers from *curse of approximation* as its performance has exponential dependency on the accuracy of search with lower bounds matching this intuition [2, 24].

These problems have motivated researchers to design data-dependent hash functions that can exploit the distribution of the dataset. The goal of data-dependent techniques is

to optimize a parametric hash function such that the similarities in the input space are kept in the target discrete space.

2.3 Data-dependent Hashing

The main focus of this dissertation is on data-dependent hashing scheme in which, instead of using random partitions, the parameters of hash functions are adjusted during a learning phase. Using a training set, the distribution of data is exploited in order to optimize the parameters of hash functions such that the input space can be efficiently partitioned based on the empirical distribution. Because of this preprocessing, the codes trained with data-dependent hashing tends to be shorter and better preserve the similarities.

Data dependent hashing techniques are roughly in two streams depending on how the distance between the compact codes are calculated:

- **Binary Hashing** techniques encode high-dimensional real-valued vectors with compact binary codes subject to preserving a given notion of similarity. Hamming distance or angular distance are then used to compare the resulting binary codes which can be executed extremely fast using bitwise operations.
- **Multi-Codebook Quantization**, as a generalization of k-means, partitions the feature space into a number of non-overlapping regions each with a unique center. By storing the distances between centers in a lookup and approximating each point with the center of its region, computing distance between a query and an approximated vector can be swiftly performed with a series of lookup and addition operations.

In what follows, a summary of some of the well-known techniques in both categories is provided. A more detailed and deeper review can be found in [127].

2.3.1 Binary Hashing

Binary hashing aims at training hash functions, $h : \mathbb{R}^d \to \{0,1\}^p$, such that the close/far vectors in the input domain have corresponding close/far binary strings (see Figure 2.1). Once such hash functions are trained, all items of database are mapped into the binary space. Then, during search time, the query is similarly mapped and search is performed among the short compact codes.

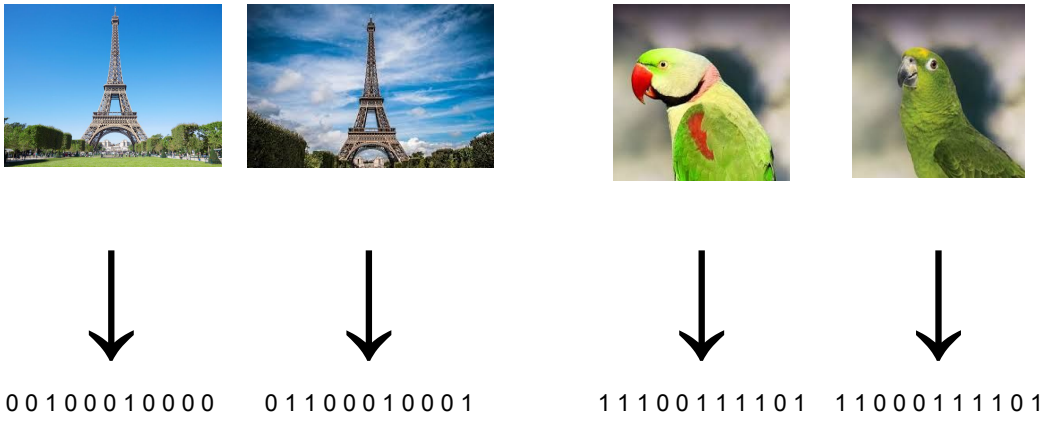


Figure 2.1: Binary hashing aims at encoding similar/dissimilar items with close/far binary codes.

Binary hashing has been an extremely active field of research over the last decade, resulting in a rich family of techniques tailored for different applications. Earlier hashing technique used simpler hash functions $h(.)$, such as linear function $h(\mathbf{x}; W, b) = sign(W^T\mathbf{x}+\mathbf{b}))$, with a limited number of parameters [91], whereas recent studies are mainly focused on end-to-end training of deep models with multiple layers of non-linearity to find better codes. Also, existing hashing methods can be categorized into unsupervised and supervised. Unsupervised hashing methods learn functions that map data to binary codes using unlabeled data. Typical learning criteria are reconstruction error minimization [66, 23], preserving local neighborhood [73] and quantization error minimization [47]. Supervised hashing, on the other hand, aims at learning binary codes that are faithful to a given notion of semantic information such as pointwise (class labels) [68, 112], pairwise [19, 20, 27, 91] or tripletwise labels [93, 147]. The similarity measure among the binary codes can be also different. While most adopt Hamming distance to compare binary codes, other metrics such as angular distance [46], spherical Hamming distance [52] and weighted Hamming distance [140] are used.

Here, the hashing techniques are categorized into shallow and deep approaches:

Shallow hashing. Salakhutdinov and Hinton [108] were perhaps among the first researchers to enunciate the idea of data-dependent hashing for text retrieval and then to the vision community by Torrablba *et al.* [124]. In both of these studies, neural networks are trained to learn binary codes. The experimental studies show successful applications of these techniques for large-scale text and image retrieval.

Shortly after, Weiss *et al.* proposed *Spectral Hashing* (SH) [132], which aims at finding hash functions such that i) semantically similar items are mapped to similar hash codes based on the Hamming distance, and ii) bits are uncorrelated and balanced that is the probability of a bit to take the value of 0 or 1 is 0.5. In particular the cost function of SH is:

$$\begin{aligned} \min: \quad & \sum_{ij} s_{ij} ||\mathbf{b}_i - \mathbf{b}_j||_H \\ \text{s.t.} \quad & \mathbf{b}_i \in \{-1, 1\}^p, \quad \sum_i \mathbf{b}_i = 0, \quad \frac{1}{n} \sum_{ij} \mathbf{b}_i \mathbf{b}_j^T = I \end{aligned} \tag{2.3}$$

where p is the length of the binary codes and s_{ij} denotes the similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$ and $\|.\|_H$ denotes the Hamming norm.

The authors then transformed the above cost function to a graph partitioning problem and proposed several relaxation of the graph Laplacian decomposition step. While Spectral Hashing outperforms data-independent techniques for short binary codes, the relaxation used to propose the out of sample extension is simplistic. In particular, it assumes that the data is sampled from a uniform distribution. Several extensions for SH were proposed over the years; the kernelized version of SH was introduced by He *et al.* [50] which uses the same constraints but kernelizes the cost function. Sparse Spectral Hashing [111] combines the Sparse Principle Component Analysis [148] into SH. Multi-dimensional Spectral Hashing [131] is another SH-related technique that seeks hash codes such that the weighted Hamming affinity is equal to the affinity in the original space.

Around the same time, Kulis and Darrell [66] proposed learning projections with *Binary Reconstructive Embedding* (BRE) that directly minimizes the Euclidean distance between the binary codes and the original data points through coordinate distance. The cost function of BRE is:

$$\min: \quad \sum_{\{i,j\} \in \mathcal{N}} (||\mathbf{x}_i - \mathbf{x}_j||_2 - ||\mathbf{b}_i - \mathbf{b}_j||_H)^2 \tag{2.4}$$

where the hash function to compute $\mathbf{b}_i$ is parametrized as:

$$b_{ij} = h_j(\mathbf{x}_i) = \text{sign}[\sum_{t=1}^{T_j} w_{jt} K(\mathbf{e}_{j,t}, \mathbf{x}_i)] \tag{2.5}$$

where $\{\mathbf{e}_{j,t}\}_{t=1}^{t=T_j}$ are the sampled data items to form the hash function, $K(.,.)$ is a kernel function, and $\{w_{jt}\}$ are the weights to be learnt by minimizing the objective function over

the training set. The objective function in (2.4) is highly non-convex in $\mathbf{W}$ (the matrix of weights) and non-smooth due to the use of sign function. One limitation of BRE is its high storage requirement for training, making it impractical to be used on large datasets [91]. To address this issue, Norouzi and Fleet [91] formulated the hashing as a *structured prediction* problem, called *Minimal Loss Hashing* (MLH), to preserve semantic similarity (*i.e.*, for problems where each data point in the training set is assigned with a discrete label). Their technique does not separate the projection and thresholding steps, instead it directly learns the hash function to preserve the semantic similarity. The cost function of MLH is:

$$\begin{aligned}\min_{\mathbf{W}} \sum_{\{i,j\}\in X} I(s_{ij}=1)\max(||h(\mathbf{x}_i;\mathbf{W})-h(\mathbf{x}_j;\mathbf{W})||-\rho+1,0))+ \\ I(s_{ij}=0)\lambda\max(\rho-||h(\mathbf{x}_i;\mathbf{W})-h(\mathbf{x}_j;\mathbf{W})||+1,0)\end{aligned} \tag{2.6}$$

where ρ and λ are hyper parameters of the loss functions and $I(.)$ is the indicator function. ρ is the threshold in the Hamming space that differentiates the neighbours from non-neighbours. Intuitively, the loss function in (2.6) favours mapping similar items to binary codes that differ by no more than ρ bits. The other hyper parameter λ controls the ratio of the slopes of the penalties incurred for similar (or dissimilar) points when they are too far (or too close). Both of these hyper parameters are adjusted using the validation set.

One problem with such a loss function is that finding suitable hyper parameters with cross-validation is slow. Moreover, for many problems one cares more about the relative magnitude of pairwise distance than their precise numerical values. Thus, just considering pairwise Hamming distance over all pairs of codes with a single threshold is often restrictive. To make the loss function independent of hyper-parameters, in a follow up work, Norouzi *et al.* [93] offered a triple loss function defined in terms of *relative similarity*. Their loss function is defined over triples of items $(\mathbf{x},\mathbf{x}^+,\mathbf{x}^+)$ such that $\mathbf{x}$ is more similar to $\mathbf{x}^+$ than to $\mathbf{x}^-$. Thus, the goal is to optimize a hash function h such that $h(\mathbf{x})$ is closer to $h(\mathbf{x}^+)$ than to $h(\mathbf{x}^-)$ in terms of Hamming distance. Their final cost function takes the following form:

$$\min_{\mathbf{W}}: \sum_{(\mathbf{x},\mathbf{x}^+,\mathbf{x}^-)\in X} \max[||h(\mathbf{x};\mathbf{W})-h(\mathbf{x}^+;\mathbf{W})||_H-||h(\mathbf{x};\mathbf{W})-h(\mathbf{x}^-;\mathbf{W})||_H+1,0]+Tr(\mathbf{W}^T\mathbf{W}). \tag{2.7}$$

Two problems associated with this cost function are: i) discontinuity, and ii) non-convexity. Motivated by the upper bounds for *latent structural SVM* proposed in [139], the authors offered a continuous upper bound to mitigate the discontinuity problem and incorporated a perceptron-like optimization technique to find a local minimum for $\mathbf{W}$.

Iterative quantization (ITQ) [47] is also one of the important techniques in the earlier years of hashing. ITQ uses principal component analysis to map the data to a low dimensional space and then exploits an alternating minimization scheme to find a rotation matrix, which maps the data to binary codes with minimum quantization error.

Deep Binary Hashing. Not surprisingly, with the dawn of deep learning, most of the recent research effort in compact coding has been directed towards using deep networks for producing compact and functional binary codes. As shown in Figure 2.2, deep hashing methods simultaneously learn the representation and hash coding from raw inputs in order to pull the codes of similar items and push the codes of dissimilar items away.

One of the main obstacles in realizing the real end-to-end training of deep hashing is that due to the binary constraint on the codes, deep hashing is essentially a discrete optimization problem and thus can not be directly solved with back-propagation. To be specific, adopting sign function in the last layer of network to turn continuous features into binary codes cause a variant of *vanishing gradient* problem as the gradient of sign function is zero for all nonzero input and thus does not carry any information. To address this issue, majority of studies adopt the "relaxation and rounding" approach: first optimize a relaxed problem by replacing the sign function with a differentiable approximation (such as sigmoid or tanh) then, once the network is trained, the final binary codes are obtained by using the sign function [70, 138, 137, 145]. Some other techniques also add a penalty term to the loss in order to generate features as discrete as possible [69, 72, 146].

HashNet [22] takes an alternative approach and starts the training with a smoothed activation function $\tanh(\alpha x)$ and increases α until eventually almost behaves like the sign function. Discrete Supervised Discrete Hashing (DSDH) [68] also solves the discrete optimization problem with the discrete cyclic coordinate descent (DCC) [112] algorithm which can keep the discrete constraint during the optimization. Greedy Hash [119] takes a greedy principle by iteratively updating the network parameters towards the probable optimal discrete solution in each iteration. To generate the discrete codes, the model uses the sign function in the forward pass but transmits the original gradients through the backward pass.

Table 2.1 provides a summary of reviewed binary hashing techniques.

2.3.2 Multi-Codebook Quantization

An alternative approach for enhancing distance computation and compressing high-dimensional items is through Vector Quantization (VQ). We start off with a brief introduction of VQ

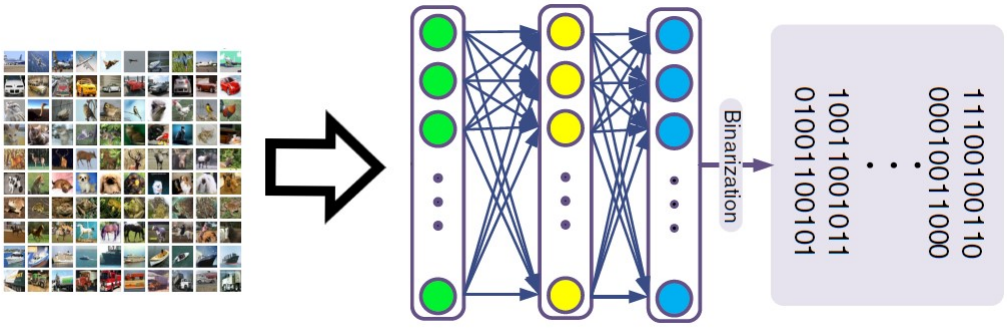


Figure 2.2: Deep hashing techniques pass the input item through a deep neural net and apply a binarization layer to the deep features to generate binary codes.

Technique	**supervised**?	**Information**	**Function**
Binary Reconstructive Embedding [66]	unsupervised	pairwise	kernel
Discrete Graph Hashing [73]	unsupervised	local structure	kernel
DistillHash [136]	unsupervised	local structure	deep network
Kernelized Discrete Graph Hashing [114]	unsupervised	local structure	kernel
Iterative Quantization [47]	unsupervised	quantization error	linear
AddLabelHash [138]	supervised	pointwise	deep network
Deep Supervised Discrete Hashing [68]	supervised	pointwise	deep network
Greedy Hash [119]	supervised	pointwise	deep network
HashNet [22]	supervised	pairwise	deep network
Kernelized Spectral Hashing [50]	supervised	pairwise	kernel
Hamming Metric Learning [93]	supervised	triplet	(non)linear
Minimal Loss Hashing [91]	supervised	pairwise	linear
Scalable Deep Hashing [30]	supervised	pointwise	deep network
Semantic-Preserving Discrete Hashing [143]	supervised	pointwise	deep network
Spectral Hashing [132]	supervised	pairwise	eigenfunction
Supervised Discrete Hashing [112]	supervised	pointwise	linear

Table 2.1: Summary of some of the important binary hashing techniques in literature.

and then discuss how it can enhance the distance computation in the context of nearest neighbor search.

VQ is a form of lossy data compression that is used to *quantize* continuous multi-dimensional vectors. Formally, a quantizer is a function that maps d-dimensional vector $\mathbf{x} \in \mathbb{R}^d$ into a discrete and finite set of items, $\mathcal{C}$. VQ allows efficient storage, retrieval, and manipulation of large-scale data sets, and therefore has a wide range of applications in information retrieval, machine learning, and signal processing. VQ has been extensively studied over the last few decades [59, 127]. In recent years, the challenges of using data analytic tools for large-scale and high-dimensional data sets – specially for embedded systems [56] – have called for creating more efficient and scalable VQ techniques.

The generic approach for quantizing a set of n data points, $\mathcal{X} = \{x_i \in \mathbb{R}^d\}_{i=1}^n$, is partitioning the domain (and therefore $\mathcal{X}$) into k subsets, and then encoding all the data points within each partition using the same *code*. These codes can then be decoded back to approximately reconstruct the original data points. In particular, the decoder maps each code to its corresponding *codeword*. The set of all codewords, $\mathcal{C} = \{\mathbf{c}_j\}_{j=1}^k$, is called the *codebook*. k-means clustering is perhaps the best-known approach for vector quantization, where the codewords are the cluster centers, and the reconstruction error for each data point is the square of its distance to its closest center.

The best VQ approach is not merely the one with the lowest reconstruction error (*i.e.*, lowest *distortion*); on top of that, the codebook should be efficiently learnable (*e.g.*, finding the centers in the k-means), the encoder/decoder should be efficient (*e.g.*, the time complexity of encoding a data point and the space complexity of storing the codebook should be small), and the approach should allow efficient computations on the compressed data for downstream data processing tasks (*e.g.*, finding the nearest neighbor of a query point within a set of quantized points).

Formally, let $\mathbf{x} \in \mathbb{R}^d$ be a point to be quantized. Given a fixed linear map $C : \{0,1\}^k \to \mathbb{R}^d$, one can try to encode $\mathbf{x}$ using $\hat{\mathbf{b}}$ by minimizing the following reconstruction loss:

$$\hat{\mathbf{b}} = \arg\min_{\mathbf{b} \in \mathcal{B}} \|\mathbf{x} - C\mathbf{b}\|_2^2 \tag{2.8}$$

where $\hat{\mathbf{b}}$ is called the assignment vector (or index vector) and $\mathcal{B} \subset \{0,1\}^k$ is the set of possible assignments. One can then use $\hat{\mathbf{b}}$ to approximately reconstruct $\mathbf{x}$ by simply calculating $\bar{\mathbf{x}} = C\hat{\mathbf{b}}$. In this setting, C is called the codebook and is implemented by a d-by-k matrix, where its columns correspond to k codewords, $\{\mathbf{c_i}\}_{i=1}^k$. Given a dataset $X^{d \times n} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$, one can find the codebook C that minimizes the empirical reconstruction error on X:

$$\min_{C} \sum_{i=1}^{n} \min_{\mathbf{b}_i \in \mathcal{B}} \|\mathbf{x}_i - C\mathbf{b}_i\|_2^2 \tag{2.9}$$

In standard VQ (*e.g.*, , k-means), the set of possible assignments, $\mathcal{B}$, is the set of all k-dimensional vectors whose entries are all zero except exactly one entry. Formally, $\mathcal{B} = \{\mathbf{b} | \mathbf{b} \in \{0,1\}^k, \|\mathbf{b}\|_1 = 1\}$. Therefore, each $\mathbf{x}_i$ is encoded using exactly one center/codeword, *i.e.*, $\bar{\mathbf{x}}_i = C\mathbf{b}_i \in \{\mathbf{c}_j\}_{j=1}^k$.

In the case of k-means clustering for VQ, clearly, one can reduce the distortion of k-means by increasing the size of the codebook, k. This, however, can result in prohibitive storage and run-time costs: as a rule of thumb, k should grow exponentially with the dimensionality of the data to allow for a fix bounded distortion – rendering the storage of the centers practically untenable.

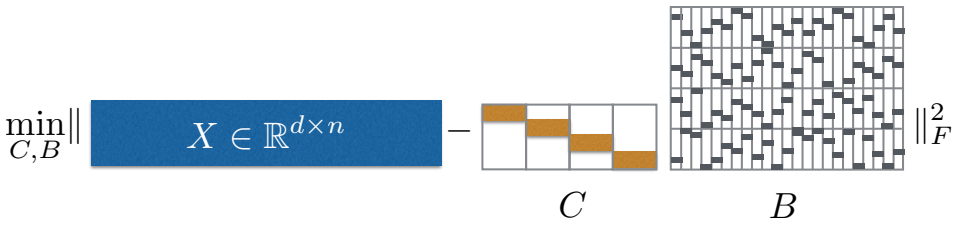
Recently, a family of techniques known as MCQ have emerged to address the aforementioned issue[59]. In MCQ, multiple codebooks , $\{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$, are learned and each data point is encoded using a combination of codewords (typically one from each codebook). Consequently, the number of "potential codewords" grows exponentially with m. This has allowed MCQ to achieve state-of-the-art quantization error [134] without the need for storing huge codebooks. Formally, in MCQ, we have m codebooks, $\{C_j\}_{j=1}^m$, and each point is encoded using a combination of codewords (one from each codebook). MCQ still seeks to minimize the distortion:

$$\min_{C} \sum_{i=1}^{n} \min_{\mathbf{b}_{ij} \in \mathcal{B}} \|\mathbf{x}_i - \sum_{j=1}^{m} C_j \mathbf{b}_{ij}\|_2^2 = \min_{C} \sum_{i=1}^{n} \min_{\mathbf{b}_i} \|\mathbf{x}_i - C\mathbf{b}_i\|_2^2 = \min_{C,B} \|X - CB\|_F^2 \tag{2.10}$$

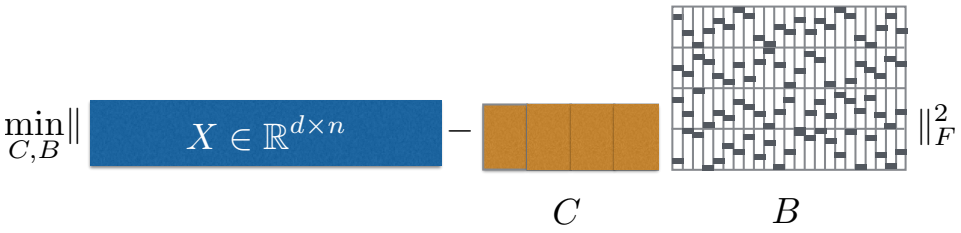
in which $\|.\|_F$ is the Frobenius norm and the definitions of C and $\mathbf{b}_i$ from (2.9) are overloaded in order to have a concise matrix form. In particular, here C is a d-by-mk matrix containing m codebooks, $C = [C_1, C_2, \ldots, C_m]$ (each $C_j \in \mathbb{R}^{d \times k}$ still contains k codewords). Furthermore, each $\mathbf{b}_i$ has a non-zero entry for *each codebook*: $\mathbf{b}_i^T = [\mathbf{b}_{i1}^T, \mathbf{b}_{i2}^T, \ldots, \mathbf{b}_{im}^T]$ where each $\mathbf{b}_{ij} \in \mathcal{B}$. Finally, B is the km-by-n assignment matrix, $B = [\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n]$.

It is easy to see that VQ is a special case of MCQ when $m = 1$. Furthermore, the "effective number of codewords/centers" in MCQ is exponential in m, as $\mathbf{b}_i$ can take k^m possible values. The quantization error of MCQ can therefore be much lower than that of standard VQ.

MCQ for NNS. One of the main applications of MCQ is large-scale NNS, in which the distance between two points is approximated with the distance between their quantized



(a) Orthogonal (product quantization)



(b) Non-orthogonal

Figure 2.3: Visual illustration of orthogonal and non-orthogonal multi-codebook quantization [81].

counterparts. The benefit is that the distance computation for the latter case can be significantly accelerated using lookup tables.

To be specific, given C and B, the distance between a query point $\mathbf{q}$ and a compressed vector $\bar{\mathbf{x}}_i = \sum_{j=1}^{m} C_j \mathbf{b}_{ij}$ can be written as:

$$\|\mathbf{q} - \bar{\mathbf{x}}_i\|_2^2 = (1-m)\|\mathbf{q}\|_2^2 + \sum_{j=1}^{m} \|\mathbf{q} - C_j \mathbf{b}_{ij}\|_2^2 + \sum_{j=1}^{m} \sum_{\substack{t=1 \\ t\neq j}}^{m} \langle C_j \mathbf{b}_{ij}, C_t \mathbf{b}_{it} \rangle. \tag{2.11}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product.

When searching for nearest neighbors, the first term of (2.11) can be ignored, as it is constant for all database vectors. To compute the second term, upon query arrival, one can create an $m \times k$ lookup table that stores the Euclidean distances between the query and all codewords. Then, this term can be computed using only m lookups and m additions in $O(m)$ time. The third term is explained below.

Variants of MCQ. MCQ techniques are often divided into three categories depending on how they treat the third term of (2.11), known as the *cross codebook inner product*

term. Orthogonal techniques [44, 59, 92], as the name suggests, restrict the codeword inner products across codebooks to be strictly zero. For example, Product Quantization (PQ) [60], which was the first study to use MCQ for nearest neighbor search, simply partitions the input features into m disjoint subsets and executes k-means in each subspace independently. Therefore, the final codebook matrix C would be block diagonal:

$$C = [C_1, \dots, C_m] = \begin{bmatrix} \hat{C}_1 & 0 & \dots & 0 \\ 0 & \hat{C}_2 & & 0 \\ \vdots & & \ddots & 0 \\ 0 & 0 & \dots & \hat{C}_m \end{bmatrix} \tag{2.12}$$

where $\hat{C}_j \in \mathbb{R}^{\lfloor d/m \rfloor \times k}$ (if d is not divisible by m,the convention is to assign the extra dimensions to the first few codebooks) denotes the centers learned in the j-th subspace. It is clear that the codewords of difference codebooks are mutually orthogonal in this setting which makes PQ an orthogonal MCQ technique. The authors also noticed that randomly rotating the data before applying the k-means could result in better recall rates. With this observation, Norouzi and Fleet [92] introduced Cartesian K-Means (CKM), a method that simultaneously learns a rotation of features and the centers in each subspace in order to further reduce the quantization error. Ge *et al.* [44] independently discovered the same method and published it in the same conference, under the name of Optimized Product Quantization (OPQ).

Semi-orthogonal techniques relax the orthogonality constraint and optimize for codebooks with constant inner products. In particular, techniques such as Composite Quantization (CQ) [128, 141] and Sparse Composite Quantization (SCO) [142], add the term $\sum_{j=1}^{m} \sum_{\substack{t=1 \\ t \neq j}}^{m} (\langle C_j \mathbf{b}_{ij}, C_t \mathbf{b}_{it} \rangle - \epsilon)^2$ to the quantization error, penalizing deviation of the cross codebook term from ϵ. This results in a non-linear constrained optimization problem that is solved with the limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm [89].

The advantage of orthogonal and semi-orthogonal codebooks is that the cross codebook term can be ignored when looking for the nearest neighbors of a query as it is constant and does not affect the ranks. However, the additional codebook constraint (orthogonality or semi-orthogonality) needs to be satisfied when optimizing (B.1).

Finally, non-orthogonal MCQ [7, 78, 80] imposes no constraints on the codebooks, which can result in codebooks with lower quantization error. In comparison to orthogonal and semi-orthogonal MCQ, non-orthogonal techniques pose a more challenging optimization problem with more learnable parameters; in addition, they need to calculate the cross codebook inner product term during distance computation. Additive Quantization (AQ) [7] first proposes to accelerate this process through creating an additional

lookup table that stores inner product between all codewords and then performing the nested summation during search time. However, this would increase distance computation cost from $O(m)$ to $O(m^2)$. Alternatively, the authors proposed to learn a separate scalar quantizer (*e.g.*, k-means) that compresses the inner product term for each vector. The advantage is that the distance computation cost remains linear, but the storage cost grows due to the additional quantizer. Local Search Quantization (LSQ) [78] and Local Search Quantization++ (LSQ++) [81] also adopt a separate norm quantizer but utilize Iterated Local Search (ILS), a heuristic search method, for optimizing B to enhance the quality of codes. The authors of LSQ++ also release a GPU implementation of their technique which significantly accelerates the learning phase.

Figure 2.3 shows orthogonal and non-orthogonal MCQ in a graphical and more intuitive way.

Supervised Quantization

While most of the research in supervised hashing is focused on supervised binary hashing, a handful of studies have been recently proposed on using MCQ in the supervised setting. Supervised MCQ techniques can be, for the most part, described as a combination of supervised loss function and one of the unsupervised MCQ techniques described above. In a supervised setting, one has access to both the data and its corresponding labels. In this scenario, it is possible to use the labels to improve the accuracy of systems that compute ANNs as part of a semantic retrieval or large-scale classification pipelines. The main challenge in this area is that current deep learning systems are usually trained with backpropagation, and are thus more efficiently trained with differentiable operations. Since quantization is in general non-differentiable, work mostly focuses on optimization techniques that approximate hard quantization assignments.

Supervised Quantization (SQ) [129] combines supervised ℓ_2 loss with CQ [128], however, the resulting optimization problem is hard to solve as it inherits the constant inter-dictionary-element-product constraint from CQ. Deep Quantization Network (DQN) [21] combines a deep architecture and PQ. One shortcoming of DQN is that during codebook optimization, it ignores the supervisory information.

More recently, Jain *et al.* [57] introduced SUpervised structured BInary Code (SUBIC), a neural network that both learns feature representations and produces compact codes. SUBIC manages to implicitly learn a set of codebooks by adding sparsity terms that ensure blockwise one-hot encodings, and balancing the use of all codebooks across the dataset. Klein and Wolf [63] have further improved upon SUBIC with a network that

Technique	Supervised?	Information	Codebooks
Additive Quantization [7]	unsupervised	quantization error	non-orthogonal
Cartesian k-means [92]	unsupervised	quantization error	orthogonal
Competitive Quantization [98]	unsupervised	quantization error	non-orthogonal
Composite Quantization [128]	unsupervised	quantization error	semi-orthogonal
Local Search Quantization [78]	unsupervised	quantization error	non-orthogonal
Local Search Quantization++ [80]	unsupervised	quantization error	non-orthogonal
Optimized Product Quantization [44]	unsupervised	quantization error	orthogonal
Optimize Tree Quantization [9]	unsupervised	quantization error	semi-orthogonal
Product Quantization [59]	unsupervised	quantization error	orthogonal
Sparse Composite Quantization [142]	unsupervised	quantization error	semi-orthogonal
Unsupervised Neural Quantization [86]	unsupervied	quantization error	non-orthogonal
Deep Triplet Quantization [71]	supervised	triplet	orthogonal
Deep Quantization Network [21]	supervised	pairwise	orthogonal
Supervised Quantization [129]	supervised	class-label	semi-orthogonal
SUBIC [57]	supervised	class-label	non-orthogonal

Table 2.2: Summary of some of the important MCQ techniques in literature.

explicitly learns soft- and hard-quantized representations. Followup work by Jain *et al.* [58] introduces a differentiable large-scale indexing structure that can be learned end-to-end together with SUBIC, resulting in the first complete image indexing pipeline that can be learned end-to-end.

Finally, very recently, Moroz and Babenko [86] proposed Unsupervised Neural Quantization (UNQ) using a deep auto-encoder like architecture where the encoder maps the input into compact discrete codes and the decoder tries to reconstruct the original input from the compressed codes. To ensure that the compressed code are faithful to the neighborhood structures of the input images, the authors add a triplet-loss to the auto-encoder loss function. The final model is then trained to minimize the objective using stochastic gradient descent.

Table 2.2 provides a summary of reviewed MCQ techniques.

2.4 Non-exhaustive Search Among Compact Codes

The data-dependent techniques discussed so-far try to reduce the cost of distance computation and storage by using compact representations. However, the search cost would

still remain linear in the number of items in the dataset, if no sublinear search algorithm is incorporated. Unfortunately, even if we use compact codes to enhance the distance computation, the search time of linear scan can still be in the order of minutes [97].

The concept of sublinear algorithms has been introduced for quite a long time, but initially it has been used to denote "pseudo sub linear" algorithms which achieve sublinear time at the cost of $\Omega(n)$ preprocessing time. Sublinear algorithms, as what the name shows, solve problems using less than linear time or space as against to the input. Perhaps the most well-known one dimensional sublinear algorithm is the binary search with $O(\log n)$ search time. Sublinear algorithms are key ingredients of similarity search algorithms as they do not require to read the entire input. Only recently a handful of studies have investigated the use of sublinear data structures and search algorithm for data-dependent high dimensional hashing techniques. Here, I review some of the state of the art sublinear search algorithms in the context of binary codes and quantized vectors.

An inverted index is a data structure that maps the content of an item (such as words or numbers) to its location in the dataset. In its most basic form, an inverted index is a simple hash table which maps words in the documents to some sort of document identifier (or the document itself). Similar ideas have been applied in computer vision domain where Bag of Words is used as the content of images. The inverted index can increase retrieval speed at the cost of increased processing time when an item is added to the dataset. It is one of the most popular data structures to achieve sublinear search time in a broad array of fields such as text mining and image retrieval. Given the query point, instead of searching through all items in the datasets, an inverted index data structure looks up the entries of dataset where the neighbours of the query may reside.

Perhaps one of the most compelling reasons for using discrete codes is that they can be incorporated in an inverted index data structure in which binary codes are treated as the direct indices of a hash table. This can potentially result in a considerable decrease in the search speed compared to the brute force search. However, using binary codes as direct indices is not necessarily efficient. To retrieve the nearest binary codes, one needs to check all the buckets within some specific Hamming ball around the query. The key problem is that the number of such buckets grows near-exponentially with the search radius. In particular, the number of buckets that lie in the Hamming ball with radius r centered at $\mathbf{q}$ is $\sum_{i=0}^{r} \binom{p}{i}$ where p is the length of the binary codes. Even for a small search radius, the number of buckets to examine is usually larger than the number of items in the dataset, making linear scan a better alternative. As a result of this problem, earlier approaches on binary codes, resort to exhaustive search for codes larger than 32 bits [66, 108, 124]

In the context of binary codes, Norouzi *et al.* [94] proposed the Multi-Index Hashing

(MIH) [49] technique to increase the search speed among a dataset of binary codes. MIH does not directly solve the KNN problem, instead it solves the radius search (find all items within a given radius from the query) which is then used to solve the KNN.

MIH partitions the binary codes into several disjoint substrings. Then, it creates one hash table per substring and populates each of the substrings into its corresponding hash table. In particular, each binary code such as $\mathbf{b} \in \{0,1\}^p$ is partitioned into m substrings, $\mathbf{b}^{(1)}$, $\mathbf{b}^{(2)}$,...,$\mathbf{b}^{(m)}$ where each substring has p/m bits (for the sake of simplicity, assume that p is divisible by m). Then, it populates the i-th hash table with the substring $\mathbf{b}^{(i)}$.

The key idea of MIH is that if two strings are close to each other, then their substrings must be also similar to each other. Formally, if the Hamming distance between two binary codes such as $\mathbf{g}$ and $\mathbf{b}$ is r, then according to the pigeon-hole principle, at least in one of their substrings, they cannot differ in more than $r' = \lfloor r/m \rfloor$ bits. During the query phase, MIH similarly divides the query binary string into substring $\mathbf{q}^{(1)}, \ldots, \mathbf{q}^{(m)}$. Then, to search for the codes whose Hamming distance from the $\mathbf{q}$ is at most r, the MIH searches all hash tables for entries that are within the Hamming distance of r' of $\mathbf{q}^{(j)}$, $j \in \{1, \ldots, m\}$. By doing that, we obtain a set of candidates neighbours from the j-th substring hash table, denoted by $\mathcal{T}_j(\mathbf{q})$. According to the pigen-hole principle mentioned above, the set $\mathcal{T}(q) = \cup_j \mathcal{T}_j(\mathbf{q})$ is the superset of the r-near neighbours of $\mathbf{q}$. In the last step of the algorithm, MIH computes the full Hamming distance between the query code and the candidates in $\mathcal{T}$, keeping those that are the true r-near neighbours.

While MIH is designed to solve the r-near neighbor problem, it can be easily used to solve the KNN problem: starting from Hamming distance radius equal zero, $r = 0$, one can progressively increase the radius until K number of neighbours is found.

The idea of using multiple hash tables and multi-index hashing to increase retrieval speed was originally proposed in [49] and one can think of the MIH as a multi-index hashing approach tailored for searching among binary codes. Experimental studies show that MIH can provide speed up factors up to orders of magnitude in comparison to linear scan. However, it is specific for solving the Euclidean KNN among binary codes. In some applications, binary codes are compared in terms of the cosine similarity rather than the Hamming distance [46, 117].

The idea of multi-index hashing has also been extended to MCQ coding space. Babenko and Lempitsky [8] proposed to use multiple hash tables to achieve sublinear search for compositional quantization techniques such as PQ and CKM. Their proposed data structure creates a separate hash table for each subspace and populates the items in the dataset in each of them. The entries of the hash tables are all possible tuples of codewords from the codebooks corresponding to different dimension groups. They propose an iterative

algorithm that produces the sequence of multi-index entries ordered by increasing distance between the query and the centroid of the corresponding entry. The main drawback of their approach is that the cost of searching for problems with more than 2 subspaces is prohibitive. Often in practice, the number of subspaces should be much more than 2 to achieve acceptable retrieval rates. Another technique is this context is PQTable [82] which adopts multi-sequence algorithm for efficiently finding candidate tables.

2.5 Summary

This Chapter reviews the state-of-the-art in approximate nearest neighbor search. I discussed binary hashing and multi-codebook quantization in ANN which are the main building blocks of this manuscript. I also reviewed some of the classical exact algorithms and showed their limitations in scalable applications. In the following Chapters, I draw upon discussed techniques and propose several approaches for enhancing the speed and accuracy of search. In particular, the first two Chapters introduce non-exhaustive search methods among binary codes. Moving forward, Chapters 5 and 6 extend existing MCQ techniques and propose a supervised and an unsupervised quantization techniques that outperform the state-of-the-art in quantization-based nearest neighbor search.

Chapter 3

Fast Cosine Similarity Search with Multi-index Hashing

One main advantage of incorporating binary codes is that the distance between two codes can be computed extremely fast using bitwise operators. For example, the Hamming distance between two codes can be computed by performing an XOR operation followed by counting the number of ones in the result (computed using the *population count* operator). This important feature makes binary codes a suitable fit for the task of KNN search. Nevertheless, performing linear search among the binary codes can still take several minutes [97] for large-scale binary datasets encountered in practice. Thus, it is imperative to obtain a solution with runtime that is *sublinear* in the dataset size.

A relevant question concerns the possibility of utilizing data structures that provide sublinear search time, such as a hash table. It turns out that binary codes are in fact a suitable fit for hash tables as binary codes lie in a discrete space. To find K nearest neighbors, a hash table is populated with binary codes where each code is treated as an index (memory addresses) in the hash table. Then, one can probe (check) the nearby buckets of the query point until K items are retrieved. For instance, if the Hamming distance is used as the measure of similarity, then the algorithm that solves the exact KNN problem is as follows: starting with a Hamming radius equal to zero, $r = 0$, at each step, the algorithm probes all buckets at the Hamming distance r from the query. After each step, r is increased by one, and the algorithm proceeds until K items are retrieved. However, in some applications, binary codes are compared in terms of cosine similarity, instead of the Hamming distance [13, 46, 117]. This is known as the *angular KNN* problem. In such cases, there are no exact sequential procedures for finding the correct sequence of probings. In practice, instead of using a hash table, researchers resort either to the

exhaustive search [46], or to the approximate similarity search techniques [117], such as LSH [55].

3.1 Contributions

This Chapter proposes a sequential algorithm for performing exact angular KNN search among binary codes. Our approach iteratively finds the sequence of hash table buckets to probe until K neighbors are retrieved. We prove that, using the proposed procedure, the cosine similarity between the query and the sequence of generated buckets' indices will decrease monotonically. This means, the larger is the cosine similarity between a bucket index and the query, the sooner the index will appear in the sequence.

Using a hash table for searching can in principle reduce the retrieval time, nevertheless, this approach is only feasible for very compact codes, *i.e.*, 32 bits at most [47, 94]. For longer codes (*e.g.*, 64 bits), many of the buckets are empty and consequently the number of buckets that must be probed to find the K nearest neighbors often exceeds the number of items in the dataset, making linear scan a faster alternative. MIH [83, 49, 77, 95, 94, 97] is a powerful technique for addressing this issue. The MIH technique hinges on dividing long codes into disjoint shorter codes to reduce the number of empty buckets. Motivated by the MIH technique proposed in [94], Angular Multi-Index Hashing (AMIH) technique is developed to realize similar advantages for the angular KNN problem. Empirical evaluations of our approach show orders of magnitude improvement in search speed in conjunction with large-scale datasets in comparison with linear scan and approximation techniques.

Given a binary query and a hash table populated with binary codes, this study raises the following research questions:

1. RQ 3.1: What is correct probing sequence for solving the angular KNN problem?
2. RQ 3.2: How can MIH technique be tailored for the angular KNN problem?
3. RQ 3.3: What is the effect of AMIH on the query time?

To answer these questions, which are related to our high-level research question RQ1 discussed in Chapter 1, first we establish a relationship between the cosine similarity and the Hamming distance. Relying on this connection, a fast algorithm for finding the correct order of probings is introduced. This allows modifying the multi-index hashing approach such that it can be applied to the angular KNN problem.

3.2 Fast Cosine Similarity Search

Given the dataset $\mathcal{B} = \{\mathbf{b}_i \in \{0,1\}^p\}_{i=1}^n$, our goal is to build a data structure such that when later given query $\mathbf{q} \in \{0,1\}^p$, the data structure can quickly report the K points in $\mathcal{B}$ with the largest cosine similarity to $\mathbf{q}$. This problem, known as Angular KNN, admits a straightforward solutions which linearly compares the query with all the points $\mathcal{B}$ however its computational cost is prohibitive for large-scale datasets.

Another related search problem is the R-near neighbor problem (RNN). The goal of RNN problem to report all data points lying at distance at most R from the query point. Similarly, if the Euclidean distance is used as the similarity measure, the problem is called the Euclidean R-near neighbor.

RNN and KNN problems are closely related. For binary datasets, one way to tackle the Hamming KNN problem is to solve multiple instances of the Hamming RNN problem. First, a hash table is populated with binary codes in $\mathcal{B}$. Then, starting from a Hamming radius equal to zero, $R = 0$, the procedure increases R and then solves the R-near problem by searching among the buckets at the Hamming distance R from the query. This procedure iterates until K items are retrieved. Nevertheless, if cosine similarity is used, the probing sequence will not be the same as the case of the Hamming distance. Unlike the Hamming distance, the angle between two binary codes is not a monotonically increasing function of their Euclidean distance. In other words, if binary codes $\mathbf{b}_1$ and $\mathbf{b}_2$ satisfy $\|\mathbf{q} - \mathbf{b}_1\|_H > \|\mathbf{q} - \mathbf{b}_2\|_H$, it does not necessarily lead to $sim(\mathbf{q}, \mathbf{b}_1) < sim(\mathbf{q}, \mathbf{b}_2)$ where $sim(\mathbf{x}, \mathbf{y})$ is the cosine of the angle between binary codes $\mathbf{x}$ and $\mathbf{y}$, and $\|.\|_H$ denotes the Hamming norm. Next, an algorithm is proposed that systematically finds the order of probings required for solving the angular KNN problem.

To reduce the search cost, we propose to use a hash table populated with binary codes. Given the dataset $\mathcal{B}$, the idea is to populate a hash table with items of $\mathcal{B}$, where each binary code is treated as the direct index of a hash bucket. The problem here is finding the K closest binary codes (in terms of cosine similarity) to the query. Evidently, for a given query $\mathbf{q}$, the binary code that yields the largest cosine similarity is $\mathbf{q}$ itself. Therefore, the first bucket to probe has the index identical to $\mathbf{q}$. The next bucket to probe has the second largest cosine similarity to $\mathbf{q}$, and so on. In the rest of this section, we propose an algorithm for efficiently finding such a sequence of probings to address our first research question (RQ 3.1).

The cosine similarity of two binary codes can be computed using:

$$sim(\mathbf{q}, \mathbf{b}_i) = \frac{\langle \mathbf{q}, \mathbf{b}_i \rangle}{\|\mathbf{q}\|_2 \|\mathbf{b}_i\|_2}, \tag{3.1}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product and $\|\cdot\|_u$ denotes the ℓ_u norm. In comparison to the Hamming distance, computing the cosine similarity is computationally more demanding. While computing Hamming distance requires an XOR followed by popcount operator, calculating cosine similarity needs the square roots and a division, refer to (3.1).

The key idea behind the proposed technique relies on the fact the set of all binary codes at the Hamming distance r from the query can be partitioned into $r + 1$ subsets, where the codes in each subset yield equal cosine similarities to the query. In particular, for two binary code $\mathbf{q}$ and $\mathbf{b}_i$ lying at Hamming distance r from each other, there are r bits that differ in the two vectors. Let $r_{1\to0}^{\mathbf{q},\mathbf{b}_i}$ denote the number of bit positions that are 1 in $\mathbf{q}$ and 0 in $\mathbf{b}_i$. Similarly, let $r_{0\to1}^{\mathbf{q},\mathbf{b}_i}$ denote the number of bit positions that are 0 in $\mathbf{q}$ and 1 in $\mathbf{b}_i$. By the definition of Hamming distance, we have $r_{1\to0}^{\mathbf{q},\mathbf{b}_i} + r_{0\to1}^{\mathbf{q},\mathbf{b}_i} = r$. Consequently, we can rewrite (3.1):

$$sim(\mathbf{q}, \mathbf{b}_i) = \frac{\|\mathbf{q}\|_1 - r_{1\to0}^{\mathbf{q},\mathbf{b}_i}}{\sqrt{\|\mathbf{q}\|_1} \times \sqrt{\|\mathbf{q}\|_1 - r_{1\to0}^{\mathbf{q},\mathbf{b}_i} + r_{0\to1}^{\mathbf{q},\mathbf{b}_i}}}. \tag{3.2}$$

It is clear that $0 \leq r_{1\to0}^{\mathbf{q},\mathbf{b}_i} \leq ||\mathbf{q}||_1$ and $0 \leq r_{0\to1}^{\mathbf{q},\mathbf{b}_i} \leq p - ||\mathbf{q}||_1$.

The dot product of two binary codes (the numerator of (3.1)) is equal to the number of positions where $\mathbf{q}$ and $\mathbf{b}_i$ are both 1, which is equal to $\|\mathbf{q}\|_1 - r_{1\to0}^{\mathbf{q},\mathbf{b}_i}$. The denominator simply contains the ℓ_2 norms of $\mathbf{q}$ and $\mathbf{b}_i$. In the rest of this chapter, we use (3.2) to compute the cosine similarity.

The important observation is that, for a given query $\mathbf{q}$, all binary codes which correspond to the same values of r_1 and r_2 lie at the same angle from $\mathbf{q}$. We use this observation to define the notation of *Hamming Distance Tuple* as follows:

Definition 3.1. (Hamming Distance Tuple) Given a query $\mathbf{q}$, we say a given binary code $\mathbf{b}_i$ lies at the Hamming distance tuple $\mathcal{H}_{\mathbf{q},\mathbf{b}_i} = (r_{1\to0}^{\mathbf{q},\mathbf{b}_i}, r_{0\to1}^{\mathbf{q},\mathbf{b}_i})$ from $\mathbf{q}$ if:

a) the number of bit positions in which $\mathbf{q}$ is 1 and $\mathbf{b}_i$ is 0 equals $r_{1\to0}^{\mathbf{q},\mathbf{b}_i}$, and,

b) the number of bit positions in which $\mathbf{q}$ is 0 and $\mathbf{b}_i$ is 1 equals $r_{0\to1}^{\mathbf{q},\mathbf{b}_i}$.

A Hamming distance tuple, such as (r_1, r_2), is *valid* if both of its elements are in valid ranges, *i.e.*, , $0 \leq r_1 \leq \|\mathbf{q}\|_1$ and $0 \leq r_2 \leq p - \|\mathbf{q}\|_1$.

Each Hamming distance tuple represents a set of binary codes lying at the same angle from $\mathbf{q}$. The number of binary codes lying at the Hamming distance tuple $(r_{1\to0}^{\mathbf{q},\mathbf{b}_i}, r_{0\to1}^{\mathbf{q},\mathbf{b}_i})$ from $\mathbf{q}$ is:

$$\binom{\|\mathbf{q}\|_1}{r_{1\to0}^{\mathbf{q},\mathbf{b}_i}} \times \binom{p-\|\mathbf{q}\|_1}{r_{0\to1}^{\mathbf{q},\mathbf{b}_i}}. \tag{3.3}$$

As all codes with the same Hamming distance tuple yield identical *sim* values, instead of searching for the correct probing sequence, we find the correct sequence of Hamming distance tuples.

We say that a Hamming distance tuple (r_1', r_2') is less than or equal to (r_1, r_2), shown by $(r_1', r_2') \preceq (r_1, r_2)$, if and only if $r_1' \leq r_1$ and $r_2' \leq r_2$.

Definition 3.2. ((r_1, r_2))-NEAR NEIGHBOR) A binary code $\mathbf{b}_i$ is called an (r_1, r_2)-near neighbor of $\mathbf{q}$, if we have $\mathcal{H}_{\mathbf{q},\mathbf{b}_i} \preceq (r_1, r_2)$.

Example 3.1. Suppose $\mathbf{q} = (1,1,1,0,0,0)$ and $\mathbf{b}_1 = (0,1,0,1,1,1)$, then $\mathbf{b}_1$ lies at the Hamming distance tuple $\mathcal{H}_{\mathbf{q},\mathbf{b}_1} = (2,3)$ from $\mathbf{q}$, and $\mathbf{b}_2 = (1,1,1,1,1,1)$ lies at the Hamming distance tuple $\mathcal{H}_{\mathbf{q},\mathbf{b}_2} = (0,3)$ from $\mathbf{q}$. Also, the Hamming distance tuple $(0,3)$ is less than the Hamming distance tuple $(2,3)$.

The partial derivatives of (3.2) with respect to $r_{1\to0}^{\mathbf{q},\mathbf{b}_i}$ and $r_{0\to1}^{\mathbf{q},\mathbf{b}_i}$ are both negative. This property indicates that, for a given Hamming distance tuple (x, y), all the binary codes with the Hamming distance tuple (x', y') satisfying $(x', y') \preceq (x, y)$ have larger *sim* values.

To visualize how the value of *sim* varies with respect to $r_{1\to0}^{\mathbf{q},\mathbf{b}_i}$ and $r_{0\to1}^{\mathbf{q},\mathbf{b}_i}$, the *sim* value as a function of $r_{1\to0}^{\mathbf{q},\mathbf{b}_i}$ and $r_{0\to1}^{\mathbf{q},\mathbf{b}_i}$ is plotted in Fig. 3.1. We are interested in sorting the tuples $(r_{1\to0}^{\mathbf{q},\mathbf{b}_i}, r_{0\to1}^{\mathbf{q},\mathbf{b}_i})$ (small circles in Fig. 3.1) in decreasing order of *sim* values. A naive way to construct the probing sequence is to compute and sort the *sim* values of all possible tuples. However, in a real application, we expect to use a small fraction of the Hamming distance tuples as we only need to probe the hash buckets until K neighbors are retrieved. Next, we propose an efficient algorithm that, in most cases, requires neither sorting, nor computing the *sim* values.

Definition 3.3. (HAMMING BALL) For a given query $\mathbf{q}$, the set of all binary codes with a Hamming distance of at most r from $\mathbf{q}$ is called the Hamming ball centered at $\mathbf{q}$ with radius r, and is shown by $\mathcal{C}(\mathbf{q}, r)$:

$$\mathcal{C}(\mathbf{q}, r) = \{\mathbf{h} \in \{0,1\}^p : \|\mathbf{q} - \mathbf{h}\|_H \leq r\}. \tag{3.4}$$

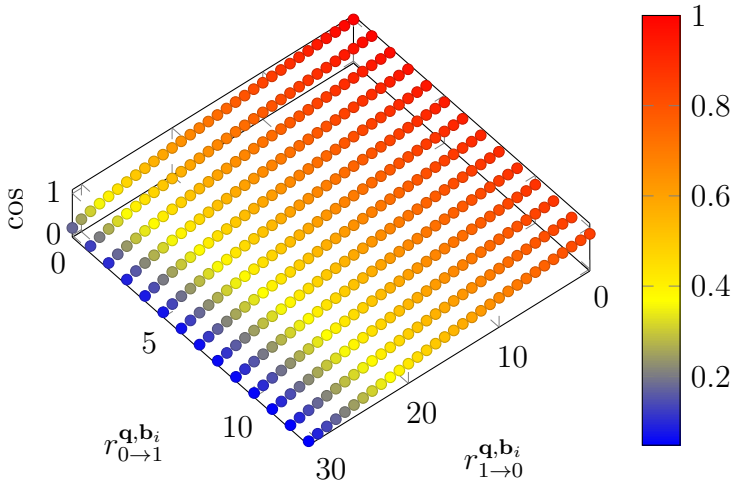


Figure 3.1: Plot of *sim* values for different values of $r^{\mathbf{q},\mathbf{b}_i}_{1\to0}$ and $r^{\mathbf{q},\mathbf{b}_i}_{0\to1}$ with $p = 45$ and $\|\mathbf{q}\|_1 = 32$.

Given $\mathbf{b}_i$, values of $r^{\mathbf{q},\mathbf{b}_i}_{1\to0}$ and $r^{\mathbf{q},\mathbf{b}_i}_{0\to1}$ can be computed efficiently using bitwise operations. However, to search for the K closest neighbors in the populated hash table, we are interested in progressively finding the values of $r^{\mathbf{q},\mathbf{b}_i}_{1\to0}$ and $r^{\mathbf{q},\mathbf{b}_i}_{0\to1}$ that lead to binary codes with the largest *sim* value. One observation is that, within all indices lying at the Hamming distance r from $\mathbf{q}$, indices with the Hamming distance tuples $(r^{\mathbf{q},\mathbf{b}_i}_{1\to0}, r^{\mathbf{q},\mathbf{b}_i}_{0\to1}) = (0, r)$ and $(r^{\mathbf{q},\mathbf{b}_i}_{1\to0}, r^{\mathbf{q},\mathbf{b}_i}_{0\to1}) = (r, 0)$ (provided that they are valid tuples) yield the largest and the smallest cosine similarities with the query, respectively. This fact leads to the following proposition:

Proposition 3.1. *Among all binary codes lying at the Hamming distance r from $\mathbf{q}$, those with larger values of $r^{\mathbf{q},\mathbf{b}_i}_{0\to1}$ yield larger cosine similarities.*

Proof: To prove this proposition, let us compute the derivatives of (3.2) with respect to $r^{\mathbf{q},\mathbf{b}_i}_{0\to1}$ or $r^{\mathbf{q},\mathbf{b}_i}_{1\to0}$ (in this proposition, to be able to take derivatives, we assume that $r^{\mathbf{q},\mathbf{b}_i}_{0\to1}$ and $r^{\mathbf{q},\mathbf{b}_i}_{1\to0}$ are continuous variables in $\mathbb{R}^+$). Suppose $r^{\mathbf{q},\mathbf{b}_i}_{1\to0} + r^{\mathbf{q},\mathbf{b}_i}_{0\to1} = r$, by replacing $r^{\mathbf{q},\mathbf{b}_i}_{1\to0}$ with $r - r^{\mathbf{q},\mathbf{b}_i}_{0\to1}$, we obtain:

$$sim(\mathbf{q}, \mathbf{b}_1) = \frac{\|\mathbf{q}\|_1 + r^{\mathbf{q},\mathbf{b}_i}_{0\to1} - r}{\sqrt{\|\mathbf{q}\|_1} \times \sqrt{\|\mathbf{q}\|_1 + 2r^{\mathbf{q},\mathbf{b}_i}_{0\to1} - r}}. \tag{3.5}$$

After some algebraic manipulations, it follows that $\frac{\partial sim}{\partial r^{\mathbf{q},\mathbf{b}_i}_{0\to1}} \geq 0$. Therefore, among all tuples

at the Hamming distance r from $\mathbf{q}$, the maximum of sim occurs at $(r_{1\to0}^{\mathbf{q},\mathbf{b}_i}, r_{0\to1}^{\mathbf{q},\mathbf{b}_i}) = (0, r)$, and its minimum occurs at $(r_{1\to0}^{\mathbf{q},\mathbf{b}_i}, r_{0\to1}^{\mathbf{q},\mathbf{b}_i}) = (r, 0)$.■

Proposition 3.1 states that, for tuples at the Hamming distance r from $\mathbf{q}$, sim is a growing function of $r_{0\to1}^{\mathbf{q},\mathbf{b}_i}$. As a result, the order of tuples in the direction of decreasing sim values is $(0, r), (1, r-1), \ldots, (r, 0)$. In other words, among all the binary codes that lie at the Hamming distance r from the query, those with larger ℓ_1 norms yield larger cosine similarities.

While the Proposition 3.1 specifies the direction of the search for a given Hamming distance, it does not establish the relationship between the Hamming distance and the cosine similarity for different Hamming distances.

Although the above may appear as a discouraging observation, we show that, for small Hamming distances, the cosine similarity and the Hamming distance are related to each other. In particular, the following proposition specifies the region where the cosine similarity is a monotonically decreasing function of the Hamming distance.

Proposition 3.2. *: If $\|\mathbf{q}\|_1 > \frac{r(r+t)}{t}$ for some $r, t \in \{1, \ldots, p\}$, then all binary codes in $\mathcal{C}(\mathbf{q}, r)$ yield larger cosine similarities to $\mathbf{q}$ than binary codes with Hamming distances at least $r + t$ from $\mathbf{q}$.*

Proof: According to Proposition 3.1, the maximum of the sim value for a fixed Hamming distance r occurs at $(r_{1\to0}^{\mathbf{q},\mathbf{b}_i}, r_{0\to1}^{\mathbf{q},\mathbf{b}_i}) = (0, r)$ with a sim value of $\sqrt{\frac{z}{z+r}}$, and its minimum occurs at $(r_{1\to0}^{\mathbf{q},\mathbf{b}_i}, r_{0\to1}^{\mathbf{q},\mathbf{b}_i}) = (r, 0)$ with a sim value of $\sqrt{\frac{z-r}{z}}$, where $z = \|\mathbf{q}\|_1$. The condition in Proposition 2 is satisfied if the smallest value of $sim(\mathbf{q}, \mathbf{b}_i)$, where $\mathbf{b}_i \in \mathcal{C}(\mathbf{q}, r)$, is larger than the largest value of $sim(\mathbf{q}, \mathbf{b}_j)$ where $\mathbf{b}_j$ lies at the Hamming distance $r + t$ from $\mathbf{q}$. Hence, we have:

$$\begin{aligned}
&\sqrt{\frac{\|\mathbf{q}\|_1 - r}{\|\mathbf{q}\|_1}} > \sqrt{\frac{\|\mathbf{q}\|_1}{\|\mathbf{q}\|\|_1 + r + t}} \\
&\Rightarrow \frac{\|\mathbf{q}\|_1 - r}{\|\mathbf{q}\|_1} > \frac{\|\mathbf{q}\|_1}{\|\mathbf{q}\|_1 + r + t} \\
&\Rightarrow (\|\mathbf{q}\|_1 - r)(\|\mathbf{q}\|\|_1 + r + t) > \|\mathbf{q}\|_1^2 \\
&\Rightarrow \|\mathbf{q}\|_1 > \frac{r(r+t)}{t}.
\end{aligned} \tag{3.6}$$

This concludes the proof.■

If the condition of Proposition 3.2 is satisfied for $t = 1$ and some radius of search r, then all the binary codes inside the Hamming ball $\mathcal{C}(\mathbf{q}, r)$ have larger cosine similarities than those outside of $\mathcal{C}(\mathbf{q}, r)$. Also, among all binary codes inside $\mathcal{C}(\mathbf{q}, r)$, those with larger Hamming distances from the query have smaller cosine similarities.That is, if $\mathbf{b}_i$ is closer to $\mathbf{q}$ than to $\mathbf{b}_j$ in terms of the Hamming distance, then $\mathbf{b}_i$ is also closer to $\mathbf{q}$ in terms of the cosine similarity.

Algorithm 1 Fast Cosine Similarity Search ($r \leq r'$)

Input: K: number of nearest neighbors to retrieve, $\mathbf{q}$: query, $\mathbf{H}$: hash table populated with binary codes
Output: $\mathcal{A}$: the set of retrieved items

Initialize set $\mathcal{A} = \varnothing$
Initialize integer $r = 0$
Initialize $\hat{r}$ with the positive root of the equation $r^2 + r - ||\mathbf{q}_1||$
$\hat{r} = \lfloor \hat{r} \rfloor$
while $|\mathcal{A}| < K$ and $r \leq \hat{r}$ **do**
  $R = (0, r)$
  while $R \neq (r + 1, -1)$ **do**
    if R is a valid tuple **then**
      Check the buckets in $\mathbf{H}$ lying at the Hamming distance tuple R from the query
      Add each of the found candidates to $\mathcal{A}$
      $R = R + (1, -1)$
    end if
  end while
  $r = r + 1$
end while

Therefore, for binary codes lying within the Hamming ball $\mathcal{C}(\mathbf{q}, r)$, cosine similarity is a decreasing function of the Hamming distance. In this case, the search algorithm is straightforward: for $t = 1$, the maximum integer r that satisfies the inequality condition in Proposition 3.2 is found. Let $\hat{r}$ denote the integer part of the positive root of the equation $r^2 + r - \|\mathbf{q}\|_1$ (this equation has only one positive root). Staring from $r = 0$, the search algorithm increases the Hamming radius until the specified number of neighbors are retrieved, or until r reaches $\hat{r}$. Further we know that, for each Hamming radius, the search direction should be aligned with the direction specified by the Proposition 3.1.

Definition 3.4. ((r_1, r_2)-NEAR NEIGHBOR PROBLEM) Given the query point $\mathbf{q}$ and dataset $\mathcal{B}$, the result of (r_1, r_2)-near neighbor problem is the set of all codes in $\mathcal{B}$ that lie at a Hamming distance tuple of at most (r_1, r_2) from $\mathbf{q}$.

The proposed approach, is effective in cases that K binary codes are retrieved before r reaches $\hat{r}$. It tackles the angular KNN problem by solving multiple instances of the (r_1, r_2)-near neighbor problem. An important advantage of the proposed algorithm is that it does not need to compute the actual *sim* values between binary codes. It can be efficiently implemented using bitwise operators and the popcount function. The rest of this section addresses the case of $r > \hat{r}$.

When the search radius is greater than $\hat{r}$, the *sim* value is not a monotonically decreasing function of the Hamming distance. However, we propose a sequential algorithm that can efficiently find the proper ordering of the tuples. The key idea is that, although the next tuple in the ordering can lie at many different Hamming distances, we show that it can be found by searching among a small subset of remaining tuples. In particular, first form a small set of candidate tuples is formed and then the one with the highest *sim* value is selected. To do that, one can first insert such candidate tuples (called *anchors*) into a priority queue and then sequentially select the one with highest priority. The priority of a tuple is evidently determined by its corresponding *sim* value. The queue is initialized with the tuple $(0, \hat{r} + 1)$. When a tuple is pushed into the queue, it is considered as traversed. At each subsequent step, the tuple with the top priority (the highest *sim* value) is popped from the queue. When a tuple is popped, two tuples are considered for insertion into the queue. Hereafter, these are called *the first anchor* and *the second anchor*, respectively. These two tuples are checked, and if "valid" and "not traversed", they are pushed into the queue.

Definition 3.5. (*First and Second Anchors of a Tuple)*

Given a query $\mathbf{q}$ and a Hamming distance tuple $R = (x, y)$, the first anchor and the second anchor of R are defined as follows:

- Among all tuples that lie at the Hamming distance $x + y + 1$ from $\mathbf{q}$, the tuple with the largest *sim* value is called the *first anchor* of R.
- Among all tuples that lie at the Hamming distance $x + y$ from $\mathbf{q}$ and have smaller *sim* values than R, the tuple with the largest *sim* value is called the *second anchor* of R.

Example 3.2. For the query $\mathbf{q}$ with $\|\mathbf{q}\|_1 = 10$ and $p = 32$, the first anchor of $v = (1, 4)$ is $(0, 6)$ and the second anchor is $(2, 3)$ (according to the Proposition 3.1).

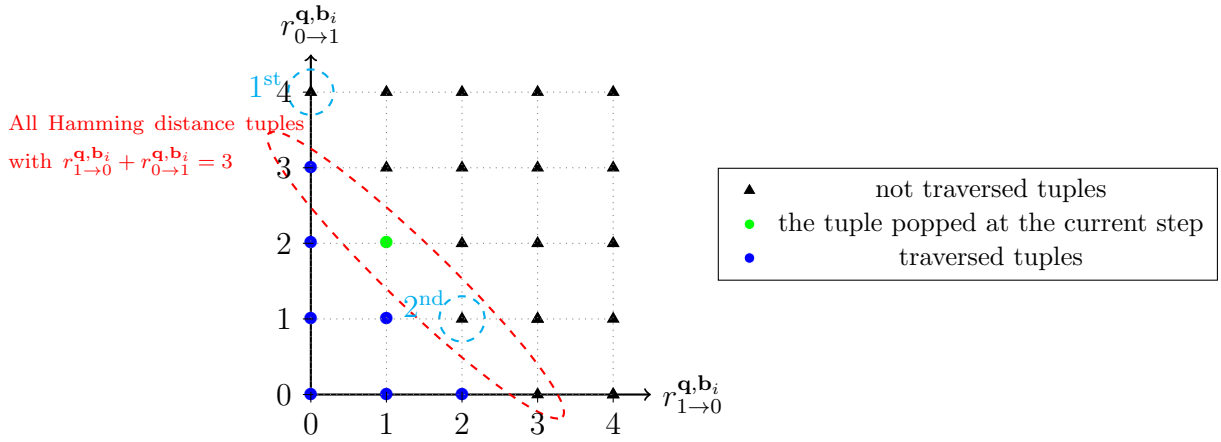


Figure 3.2: Visual representation of the "first anchor" and the "second anchor" of a Hamming distance tuple.

When a tuple is popped from the queue, the algorithm pushes the first and the second anchors of the popped tuple into the priority queue (provided that these are valid, and not traversed) and marks them as traversed. This procedure continues until either K elements are retrieved, or all valid tuples are traversed. Therefore, when a tuple such as $R = (x, y)$ is popped from the queue, the algorithm checks whether the following two tuples are valid or not:

a) **The first anchor of** R: This tuple, by definition, has the largest *sim* value among the tuples at the Hamming distance $x + y + 1$ from the query. According to Proposition 3.1, this candidate is $(0, x + y + 1)$ if $x + y + 1 \leq p - \|\mathbf{q}\|_1$. In general, to ensure that the two components of this tuple are in acceptable ranges, the first anchor of R takes the form $(c, x + y + 1 - c)$ where $c = \max(0, x + y + 1 - (p - \|\mathbf{q}\|_1))$. Note that the first component of any Hamming distance tuple is at most $\|\mathbf{q}\|_1$ (number of ones in $\mathbf{q}$) and its second component is at most $p - \|\mathbf{q}\|_1$ (number of zeros in $\mathbf{q}$).

b) **The second anchor of** R: Among the tuples that have smaller *sim* values than R, and lie at the Hamming distance $x + y$ from the query, this tuple is the one that has the largest *sim* value. Using Proposition 3.1, it is easy to show that the second anchor of R is $(x + 1, y - 1)$. This tuple is pushed into the queue if its components are in acceptable ranges (the second anchor is valid if $x + 1 \leq \|\mathbf{q}\|_1$ and $y - 1 \geq 0$).

Fig. 3.2 shows an example of the first/second anchors (shown in dashed circles) of a tuple that is selected in the current step (shown in green).

Algorithm 2 Fast Cosine Similarity Search

Input: K: number of nearest neighbors to retrieve, $\mathbf{q}$: query, $\mathbf{H}$: hash table populated with binary codes
Output: $\mathcal{A}$: the set of retrieved items

Initialize set $\mathcal{A} = \varnothing$
Initialize integer $r = 0$
Create an empty priority queue pq
Marked all candidates as not traversed
Initialize $\hat{r}$ with the positive root of the equation $r^2 + r - ||\mathbf{q}||_1$
$\hat{r} = \lfloor \hat{r} \rfloor$
$R = (0, 0)$
while $|\mathcal{A}| < K$ **do**
    if $r \leq \hat{r}$ **then**
        if R is a valid tuple **then**
            Check the buckets in $\mathbf{H}$ corresponding to the Hamming distance tuple R
            Add the retrieved items to $\mathcal{A}$
            Mark the tuple (r_1, r_2) as traversed
            $R = R + (1, -1)$
        end if
        if $R = (r + 1, -1)$ **then**
            $r = r + 1$
            $R = (0, r)$
            if $r > \hat{r}$ **then**
                $pq.push(R)$
            end if
        end if
    else
        if $pq.isempty()$ **then**
            return
        end if
        $R \leftarrow pq.pop()$
        Check the buckets in $\mathbf{H}$ corresponding to the Hamming distance tuple R
        Add the retrieved items to $\mathcal{A}$
        if The first anchor of R is not traversed **then**
            $pq.push$(the first anchor of R)
            Mark the first anchor of R as a traversed
        end if
        if The second anchor of R is a valid tuple **then**
            $pq.push$(the second anchor of R)
            Mark the second anchor of R as traversed
        end if
    end if
end while

Next, we prove that the proposed algorithm results in the correct ordering of Hamming distance tuples.

Proposition 3.3. *In each iteration, the Hamming distance tuple popped from the queue has a smaller sim value than the traversed tuples, and has the largest sim value among the not traversed tuples. Moreover, the algorithm eventually traverses every tuple.*

Proof: When $r < \hat{r}$, according to Propositions 3.1 and 3.2, the ordering is correct. For $r \geq \hat{r}$, we show that the selected candidate has the highest cosine similarity among the remaining tuples.

Assume that the algorithm is not correct. Let R be the first tuple that the algorithm selects incorrectly. This means another tuple, such as $R' = (x', y')$, yields the highest *sim* value and it has not been pushed into the priority queue because if R' had been pushed into the queue, then R' would have been popped from the queue instead of R. Let $r' = x' + y'$, meaning that r' is the Hamming distance between $\mathbf{q}$ and any binary code lying at the Hamming distance tuple (x', y') from the $\mathbf{q}$. Consider all binary codes that lie at the Hamming distance r' from $\mathbf{q}$. If there exists a tuple with the second component greater than y' that has not been traversed yet, then a contradiction occurs (this means R' does not yield the largest *sim* value). This stems from the fact that, at a fixed Hamming distance, tuples with larger second components have larger *sim* values (Proposition 3.1). As a result, y' yields the largest possible value among the not-traversed tuples lying at the Hamming distance r' from $\mathbf{q}$. However, we show that, this tuple should have been pushed into the priority queue in previous steps. One of the following cases may occur:

a) Until the current step, no Hamming distance tuple at the Hamming distance r' from $\mathbf{q}$ has been selected: According to Proposition 3.1, any tuple with the Hamming distance $r' - 1$ that is in the set $\mathcal{L} = \{(a, b) | (a, b)$ is a valid tuple and, $a + b = r' - 1, a \leq x', b \leq y\}$ has larger *sim* values than R'. Therefore, all of them must have been selected prior to R' in the sequence. However, the first time that a tuple from $\mathcal{L}$ was popped, R' was pushed into the priority queue. R' is in fact the first anchor of all the tuples in $\mathcal{L}$, and thus, it must have been pushed when any of the elements in $\mathcal{L}$ were popped from the priority queue.

b) At least one Hamming distance tuple with the Hamming distance r' from $\mathbf{q}$ has been traversed in previous steps: Similar to the previous case, R' was pushed into the priority queue when the algorithm popped the tuple $(x' - 1, y' + 1)$. In this scenario, R' is the second anchor of $(x' - 1, y' + 1)$.

It is concluded that R' must have been pushed into the priority queue during previous steps, which contradicts the assumption that R' is not a member of the priority queue.

We also need to prove that the algorithm is *complete*, *i.e.*, , the algorithm continues until it either finds the K neighbors, or it traverses all the valid tuples. Again, let us assume the contrary. This means that, at the final step, the algorithm pops the last tuple from the queue and the last tuple does not have any valid anchors. Thus, the queue remains empty and the algorithm will terminate while there are still some valid tuples that have not been traversed. It is clear that the not-traversed tuples cannot lie at the Hamming distance of r when at least one tuple with the Hamming distance r is traversed. This situation occurs because once the first tuple with the Hamming distance r is popped from the queue, the second anchor of this tuple is pushed into the queue. Therefore, one tuple with the Hamming distance r always exists in the queue until the last one of such tuples is popped, and such a last tuple does not have a valid second anchor. As a result, the only possible case is that all the tuples at a Hamming distance less than or equal to r have been traversed; and all of the tuples at a Hamming distance $r+1$ and greater have not been traversed. However, this is not possible because when a tuple at the Hamming distance r is popped from the queue, its first anchor is pushed into the queue and this tuple lies the Hamming distance $r+1$. Hence, all the tuples at the Hamming distance $r+1$ from the query will be covered eventually.■

3.3 Angular Multi-index Hashing

To achieve satisfactory retrieval accuracy, applications of binary hashing often require binary codes with large lengths (*e.g.*, , 64 bits). For such applications, it is not practical to use a single hash table mainly because of the computational cost of search. For long binary codes, it is frequently the case that $n \ll 2^p$ and thus most of the buckets in the populated hash table are empty. To solve the KNN problem in such a sparse hash tables, even for small values of K, often the number of buckets to be examined exceeds the number of items in the dataset. This means that the exhaustive search (linear scan) is a faster alternative than using a hash table. As shown in Fig. 3.3, the average number of probing required for solving the angular KNN query for the SIFT dataset with one billion items (the details of SIFT will be explained later), often exceeds the number of available binary codes in the dataset. This problem arises as the required number of probings grows near-exponentially with the values of $r^{\mathbf{q},\mathbf{b}_i}_{1\to 0}$ and $r^{\mathbf{q},\mathbf{b}_i}_{0\to 1}$ (refer to (3.3)).

Multi-Index Hashing (MIH) [49], and its variants [95, 94], are elegant approaches for reducing storage and computational costs of the R-near neighbor search for binary codes.

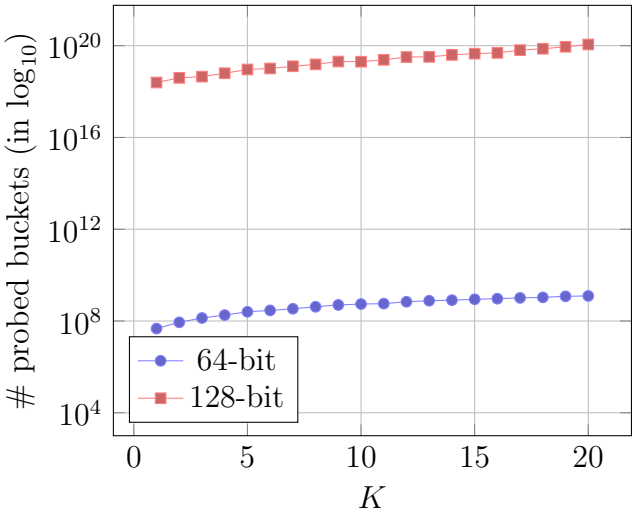


Figure 3.3: The average number of probings required for solving the angular KNN problem, if a single hash table is used for the SIFT dataset (with 10^9 items).

The key idea behind the multi-index hashing is that, as many of the buckets are empty, one can merge the buckets over different dimensions of the Hamming space. To do this, instead of creating one huge hash table, MIH creates multiple smaller hash tables with larger buckets, where each bucket may be populated with more than one item. To do this, all binary codes are divided into smaller disjoint (usually with the same length) substrings, then each substring is indexed within its corresponding hash table. Therefore instead of one creating one huge hash table, the idea of MIH is to form multiple smaller hash tables which can significantly reduce the storage cost.

More importantly, MIH reduces the computational cost of the search. To solve the R-near neighbor problem, the query is similarly partitioned into m substrings. Then, MIH solves m instances of the $\frac{R}{m}$-near neighbor problem, one per each hash table. By doing this, the neighbors of each substring in its corresponding hash table are retrieved to form a set of potential neighbors. Since some of the retrieved neighbors may not be a true R-near neighbor, a final pruning algorithm is used to remove the false neighbors.

Despite being efficient in storage and search costs, MIH cannot be applied to the angular preserving binary codes, since it is originally designed to solve the R-near neighbor problem in the Hamming space. The rest of this section proposes AMIH technique for fast and exact search among angular preserving binary codes which addresses the second research question (RQ 3.2).

Instead of populating one large hash table with binary codes, AMIH creates multiple smaller hash tables. To populate such smaller hash tables, each binary code $\mathbf{b} \in \{0,1\}^p$ is partitioned into m disjoint substrings $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(m)}$. For the sake of simplicity, in the following, it is assumed that p is divisible by m and use the notation $w = \frac{p}{m}$. As a result, the s-th hash table, $s \in \{1, \ldots, w\}$, is populated with $\mathbf{b}_i^{(s)}$ $i \in \{1, \ldots, n\}$. To retrieve the (r_1, r_2)-near neighbors of the query, $\mathbf{q}$ is similarly partitioned into m substrings, $\mathbf{q}^{(1)}, \ldots, \mathbf{q}^{(m)}$.

The following proposition establishes the relationship between the Hamming distance tuple of two binary codes and their substrings.

Proposition 3.4. *If* $\mathbf{b}$ *lies at a Hamming distance tuple of at most* (r_1, r_2) *from* $\mathbf{q}$*, then:*

$$\begin{aligned} \exists \quad 0 < t \leq m \quad s.t. \quad & \|\mathbf{q}^{(t)} - \mathbf{b}^{(t)}\|_H \leq \lfloor \frac{r_1 + r_2}{m} \rfloor \\ \wedge \quad & r_{1\to 0}^{\mathbf{q}^{(t)}, \mathbf{b}_i^{(t)}} \leq r_1 \\ \wedge \quad & r_{0\to 1}^{\mathbf{q}^{(t)}, \mathbf{b}_i^{(t)}} \leq r_2. \end{aligned} \tag{3.7}$$

The first condition follows from the Pigeon-hole principle. If in all of the m substrings, the Hamming distance is strictly greater than $\lfloor \frac{r_1+r_2}{m} \rfloor$, then we have $\|\mathbf{q}-\mathbf{b}\|_H \geq m(\lfloor \frac{r_1+r_2}{m} \rfloor + 1)$. This contradicts the assumption that $\mathbf{b}$ lies at a Hamming distance of at most $r_1 + r_2$ from $\mathbf{q}$. The second and the third conditions must in fact hold for all substrings, because if we have $r_{1\to 0}^{\mathbf{q}^{(t)}, \mathbf{b}_i^{(t)}} > r_1$, then we should have $r_{1\to 0}^{\mathbf{q}, \mathbf{b}_i} > r_1$. Similarly, if we have $r_{0\to 1}^{\mathbf{q}^{(t)}, \mathbf{b}_i^{(t)}} > r_2$, then we should have $r_{0\to 1}^{\mathbf{q}, \mathbf{b}_i} > r_2$. Thus, $\mathbf{b}$ is not a (r_1, r_2)-near neighbor of $\mathbf{q}$.■

In simple terms, Proposition 3.4 states that, if $\mathbf{b}$ is a (r_1, r_2)-near neighbor of $\mathbf{q}$, then at least in one of its substrings such as t, $\mathbf{b}^{(t)}$ must be a (r'_1, r'_2)-near neighbor of $\mathbf{q}^{(t)}$, where $r'_1 + r'_2 \leq \lfloor \frac{r_1+r_2}{m} \rfloor$, $r'_1 \leq r_1$ and $r'_2 \leq r_2$.

3.3.1 (r_1, r_2)-near Neighbor Search Using Multi-index Hashing

We have thus far established the necessary condition that facilitates the search among substrings. At the query phase, to solve a (r_1, r_2)-near neighbor search, AMIH first generates the tuples that satisfy the conditions of the Proposition 3.4. That is, to solve the (r_1, r_2)-near neighbor problem, AMIH generates the set of all tuples (r'_1, r'_2) such that $r'_1 + r'_2 \leq \lfloor \frac{r_1+r_2}{m} \rfloor$, where $r'_1 \leq r_1$ and $r'_2 \leq r_2$. This set is denoted by $\mathcal{T}_{r_1,r_2,m}$.

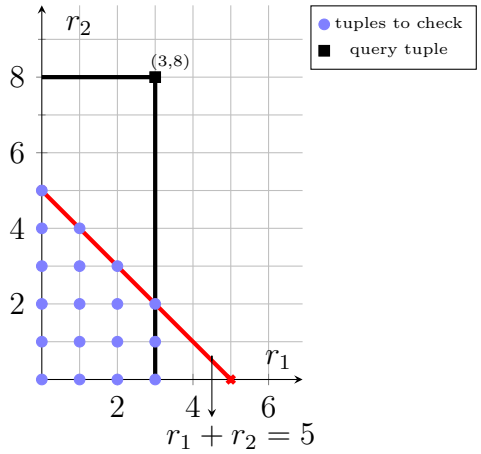


Figure 3.4: The tuples that must be checked for solving the $(3, 8)$-near neighbor problem with 2 hash tables.

Example 3.3. Suppose $m = 2$ and we are interested in solving $(3, 8)$-near neighbor problem. According to Proposition 3.4, we need to search among tuples with a Hamming distance of at most $5 = \lfloor \frac{3+8}{2} \rfloor$ that satisfy the conditions. These tuples are shown in Fig. 3.4. Notice that, for each tuple, the algorithm should probe all corresponding buckets in each of the hash tables.

Next, for each tuple such as $t = (r'_1, r'_2)$ in $\mathcal{T}_{r_1,r_2,m}$ and for each substring $\mathbf{q}^{(s)}$, $s \in \{1, \ldots, m\}$, AMIH solves the (r'_1, r'_2)-near neighbor problem for the query $\mathbf{q}^{(s)}$ in the s-th hash table. This step results in a set of candidate binary codes, denoted by $\mathcal{O}_{j,t}$. According to Proposition 3.4, the set $\mathcal{O} = \bigcup_{j,t} \mathcal{O}_{j,t}$ is the superset of (r_1, r_2)-near neighbors of $\mathbf{q}$. Finally, AMIH computes the Hamming distance tuples between $\mathbf{q}$ and all candidates in $\mathcal{O}$, discarding the tuples that are not the true (r_1, r_2)-near neighbors of $\mathbf{q}$.

The intuition behind this approach is that, since the number of buckets that lie at the Hamming distance tuple (a, b) grows near-exponentially with the values of a and b, it is computationally advantageous to solve multiple instances of (a', b')-near neighbor problem with $a' < a$ and $b' < b$, instead of solving one instance of (a, b)-near neighbor problem where a and/or b are relatively large. This requires a significantly smaller number of probings as compared to the case of deploying a single large hash table.

3.3.2 Cost Analysis

The cost analysis directly follows the performance analysis of MIH in [95]. As suggested in [95], it is assumed that $\lfloor \frac{p}{\log_2 n} \rfloor \leq m \leq \lceil \frac{p}{\log_2 n} \rceil$. Using AMIH, the total cost per query consists of the number of buckets that should be checked to form the candidate set $\mathcal{O}$, plus the cost of computing the Hamming distance tuple between retrieved binary codes in $\mathcal{O}$ and $\mathbf{q}$.

We start by providing an upper bound on the number of buckets that should be checked. Since the algorithm probes identical buckets in each hash table, the number of probings equals the product of m and the number of probings in a hash table.

To solve the (r_1, r_2)-near neighbor problem, for each tuple such as (a, b) in $\mathcal{T}_{r_1,r_2,m}$, the algorithm probes the buckets that correspond to (a, b). It is clear that, in the i-th hash table ($1 \leq i \leq m$), all binary codes corresponding to the tuples in the set $\mathcal{T}_{r_1,r_2,m}$ lie at a Hammming distance of at most $\lfloor \frac{r_1+r_2}{m} \rfloor$ from the $\mathbf{q}^{(i)}$ (Proposition 3.4). Therefore, in the i-th hash table, the indices of buckets that must be probed are a subset $\mathcal{C}(\mathbf{q}^{(i)}, \lfloor \frac{r_1+r_2}{m} \rfloor)$, and we can write:

$$\begin{aligned} \#\text{probings} &\leq \sum_{i=1}^{m} |\mathcal{C}(\mathbf{q}^{(i)}, \lfloor \frac{r_1 + r_2}{m} \rfloor)| \\ &= m \times \sum_{j=0}^{\lfloor \frac{r_1+r_2}{m} \rfloor} \binom{w}{j} \\ &= m \times \sum_{j=0}^{\lfloor \frac{w(r_1+r_2)}{p} \rfloor} \binom{w}{j}. \end{aligned} \tag{3.8}$$

Assuming that $\frac{r_1+r_2}{p} \leq 1/2$, we can use the following bound on the sum of the binomial coefficients [42].

For any $n \geq 1$ and $0 < \alpha \leq 1/2$, we have:

$$\sum_{i=0}^{\lfloor \alpha n \rfloor} \binom{n}{i} \leq 2^{H(\alpha)n}. \tag{3.9}$$

where $H(\alpha) := -\alpha \log(\alpha) - (1 - \alpha) \log(1 - \alpha)$ is the binary entropy of α.

Therefore, we can write:

$$\#\text{probings} \leq m \sum_{j=0}^{\lfloor \frac{w(r_1+r_2)}{p} \rfloor} \binom{w}{j} \leq m2^{wH(\frac{r_1+r_2}{p})}. \tag{3.10}$$

If binary codes are uniformly distributed in the Hamming space, the expected number of items per buckets of each hash table is $\frac{n}{2^w}$. Therefore, the expected number of items in the set $\mathcal{O}$ is:

$$E(|\mathcal{O}|) = \frac{n}{2^w} m \times 2^{wH(\frac{r_1+r_2}{p})}. \tag{3.11}$$

Empirically, it is observed that the cost of bucket lookup is marginally smaller than the cost of verifying a candidate. If we have: *single lookup cost*= $t\times$ *single candidate test cost*, for some $t \leq 1$, then using (3.10) and (3.11), we can write the total cost as:

$$cost \leq m2^{wH(\frac{r_1+r_2}{p})}(t + n/2^w). \tag{3.12}$$

For $m \approx p/\log_2 n$, by substituting $\log_2 n$ for w, we have:

$$cost = O(\frac{p}{\log_2 n} n^{H(\frac{r_1+r_2}{p})}). \tag{3.13}$$

For reasonably small values of $\frac{r_1+r_2}{p}$, the cost is sublinear in n. For example, for $\frac{r_1+r_2}{p} \leq 0.1$, the expected query cost would be $O(p\sqrt{n}/\log n)$.

The space complexity of AMIH comprises: a) the cost of storing n binary codes each with p bits, which takes $O(np)$, and b) the cost of storing n pointers to dataset items in each hash table. Each pointer can be represented in $O(\log_2 n)$ bits, therefore, the cost of storing pointers would be $O(mn \log_2 n)$. For $m = \lceil \frac{p}{\log_2 n} \rceil$, the total storage cost is $O(np+n \log_2 n)$.

3.4 Experiments

In order to answer RQ 3.3, this section empirically evaluates the performance of the proposed algorithm.

AMIH is coded in C++ on top of the MIH implementation provided by the authors of [95] (all codes are compiled with GCC 4.8.4). Our implementation is publicly available at https://github.com/sepehr3pehr/AMIH. The experiments have been executed on a single core of 2.0 GHz Xeon CPU with 256 Gigabytes of RAM.

3.4.1 Datasets

Two non-synthetic datasets are used in this section:

SIFT: The ANN_SIFT1B dataset [102] consists of SIFT descriptors. The available dataset has been originally partitioned into 10^9 items as the *base set*, 10^4 items as the *query set*, and 10^8 items as the *learning set*. Each data item is a 128-dimensional SIFT vector.

TRC2: The TRC2 (*Thomas Reuters Text Research Collection 2*) dataset [67] consists of 1,800,370 news stories covering a period of one year. 5×10^5 news are used as the learning set, 10^6 news as the base set, and the remaining as the query set. The data is preprocessed by removing common stop words, stemming, and then considering only the 2000 most frequent words in the learning set. Thus, each news story is represented as a vector composed of 2000 word-counts.

Since the items of these datasets lie in real space, a binary hashing technique is adopted to map the items to binary codes. For our experiments, we have used the angular-preserving mapping method called Angular Quantization-based Binary Codes (AQBC) proposed in [46], to create the dataset of binary codes. We implemented AQBC in Python following the initialization and parameter setting described in [46]. We have also made our implementation of AQBC publicly available at https://github.com/sepehr3pehr/AQBC. For each dataset, the learning set is used to optimize the parameters of the hash function. Once learning is completed, the learning set is removed and the learned hash function is applied to the base and the query sets. The base set is used to populate the hash tables. Then, the angular KNN problem is solved for all queries points and the average performance is reported.

3.4.2 AMIH vs Linear Scan

Our first experiment compares the performance of linear scan with AMIH in terms of the search speed. The norm of any binary code with p bits ranges from 0 to $\sqrt{p}$. Thus, to increase the speed of the linear scan, we initialize a look up table with all the possible norm values. Moreover, as the term $\sqrt{\|\mathbf{q}\|_1}$ in the denominator of (3.2) is independent of $\mathbf{b}_i$, there is no need to account for its value in searching.

We observed that the performance of linear scan is virtually independent of K (number of nearest neighbors). Consequently, for the sake of comparison, in the following, only the result of the linear scan for the 1NN problem is used. Note that the linear scan can benefit from caching, as it performs sequential memory access. Otherwise, it would be much slower.

Table 3.1: Speedup gains that AMIH achieves in comparison to linear scan. The last line shows the average query time of linear scan in seconds.

		SIFT 1B		**TRC2**	
	# bits:	**64**	**128**	**64**	**128**
Speedup gain	1NN	2672	1035	106	7.5
	10NN	2137	345	27.5	3.21
	100NN	1336	138	9.1	2.1
	Linear scan (s):	106	207	0.110	0.206

Fig. 3.5 shows the average query time as a function of the dataset size for 64-bit and 128-bit binary codes. In all experiments, the value of m (number of hash tables) for AMIH is set to $\frac{p}{\log_2 n}$, following [83, 49, 95]. The leftmost graphs show the search time in seconds in terms of the data base size. It is apparent that AMIH is significantly faster than the linear scan for a broad range of dataset sizes and K values. To differentiate between the performance of AMIH for different values of K, the middle graphs show the zoomed version of the leftmost graphs, and the rightmost graphs are plotted using logarithmic scale. As Figs. 3.5 illustrates, for linear scan, the query time grows linearly with the dataset size, whereas the query time of AMIH increases with the square root of the size. Consequently, the difference between the query times of the two techniques is more significant for larger datasets. For instance, linear scan spends more than three minutes to report the nearest neighbor in the 10^9 SIFT dataset with 128-bit codes, while AMIH takes about a quarter of a second. The dashed line on log-log plots shows the growth rate of the $\sqrt{n}$ up to a constant factor. The evident similarity between the slope of this function and that of AMIH query time indicates that, even for non-uniform distributions, AMIH can achieve sublinear search time.

Fig. 3.6 shows the percentage of queries for which the required radius of search gets larger than $\hat{r}$. As the size of the dataset grows, the number of empty buckets reduces, and the algorithm finds the nearest neighbors within a smaller search radius. Similarly, for shorter binary codes, the number of buckets reduces, and in turn, AMIH retrieves items before the search radius reaches $\hat{r}$.

Table 3.1 includes the speed up factors achieved by AMIH versus linear scan. Each entry in the table indicates the average query time using linear scan over the average query

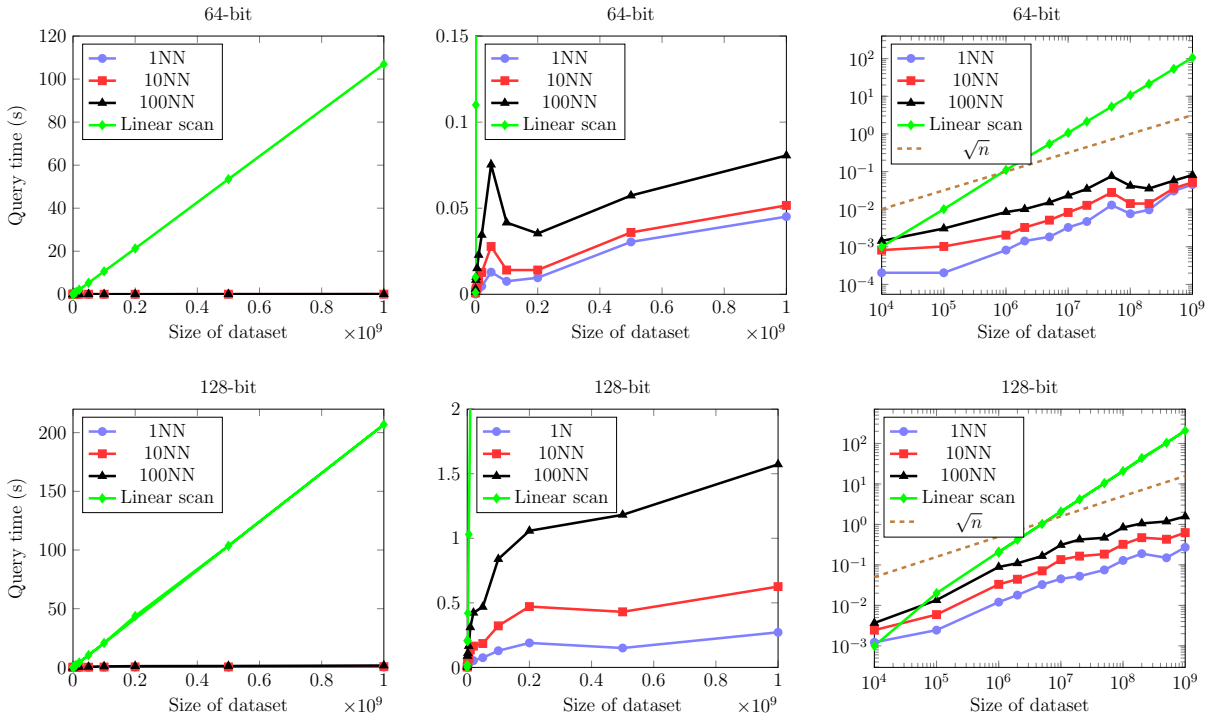


Figure 3.5: Average search time for 64-bit and 128-bit binary codes of the SIFT dataset. AMIH and linear scan are executed to solve the KNN problem with $K \in \{1, 10, 100\}$.

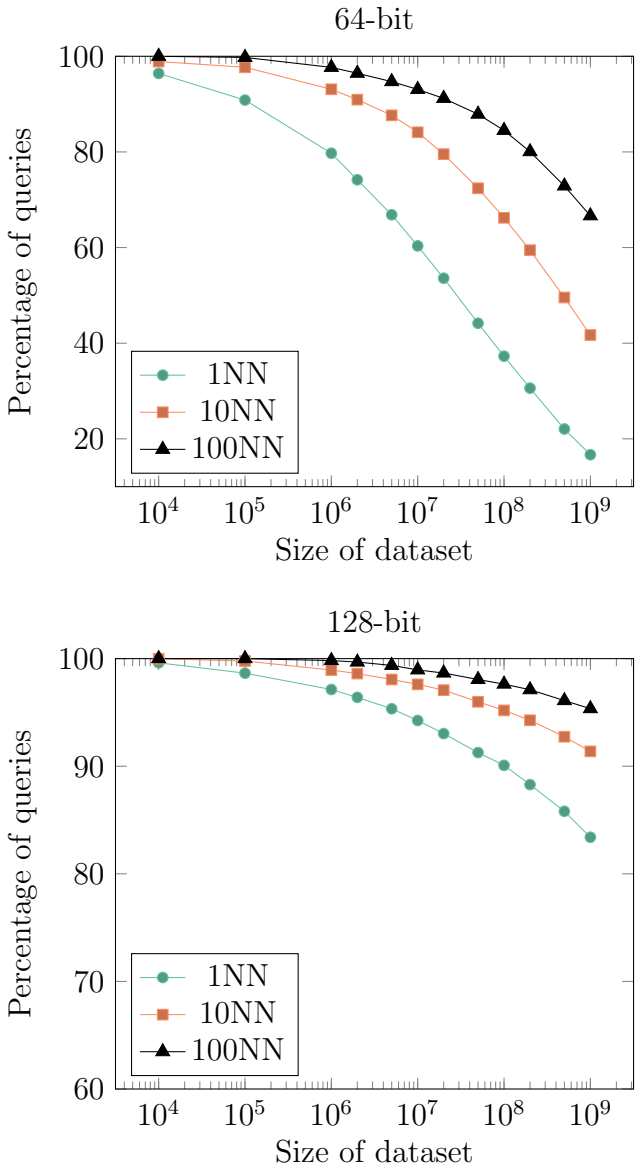


Figure 3.6: The percentage of queries for which the required radius of search gets larger than $\hat{r}$.

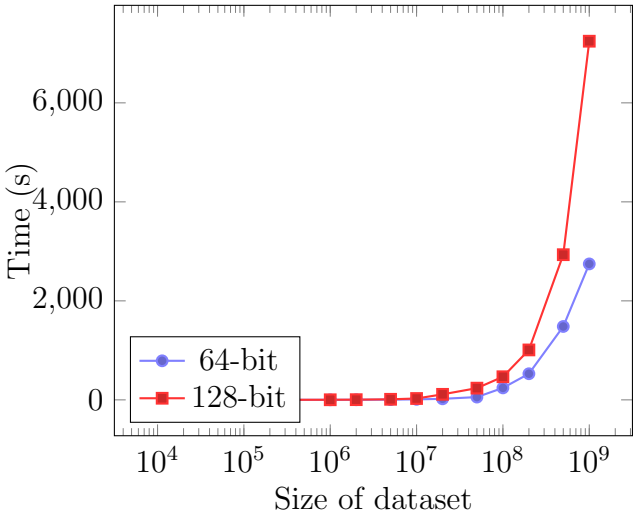


Figure 3.7: Indexing time of AMIH on 10^9 SIFT dataset.

time using AMIH for a specific value of K. Interestingly, AMIH solves the angular KNN problem up to hundreds and even thousands of times faster than linear scan. In particular, AMIH can solve the 100NN problem 138 times faster than the linear scan on a dataset of 10^9 binary codes each with 128 bits.

While the linear scan technique does not rely on any indexing phase, AMIH requires each binary code to be indexed in m hash tables. The indexing time for AMIH using the SIFT dataset is shown in Fig. 3.7. For 64 and 128 bit codes, the indexing phase takes about 1 and 2 hours, respectively.

3.4.3 AMIH vs Approximate Techniques

Due to the curse of dimensionality, linear scan is theoretically the fastest exact technique for solving the angular KNN problem in its general setting. However, a handful of approximation algorithms exist that provide sublinear search time for this problem. The most well-known representative among these is LSH [55], which offers a (provably) sublinear search time. In addition to LSH, some applied approximation algorithms have been proposed which work promising in realworld applications, such as *KGraph* [35] and *Annoy* [15], but do not necessarily guarantee (efficient) worst case analysis. In this section, we compare AMIH with some of the well-known approximation techniques for the task of nearest neighbor search (1NN).

The comparison between AMIH and other approximation techniques is performed in two different scenarios:

- First, we investigate the performance of different techniques for solving the angular nearest neighbor problem in the binary space. Similar to the experiments of section 3.4.2, we assume that we are given a binary dataset and the goal entails solving the angular nearest neighbor search for binary query points.
- In the second scenario, we assume that the original dataset lies in the real space. Given the dataset, we apply the approximate algorithms to the real dataset. However, since AMIH can be only applied to binary codes, we first use a hashing algorithm to map the dataset to binary codes and then use AMIH to solve the K nearest neighbor search within the binary dataset. Finally, among the K retrieved points, we select the one that is the closet to the query in the real space. Therefore, the returned nearest neighbor in this setting is approximate with respect to the original data points lying in the real space. This scenario mainly targets applications in which the original dataset items do not lie in the binary space because for binary datasets the result of AMIH is exact.

Approximate Techniques Used for Comparison

For both scenarios, we compare AMIH with three state-of-the-art ANN techniques. Discussing the details of these techniques is beyond the scope of this study but we briefly introduce each of them here.

Crosspolytope LSH [3] is a recently proposed LSH-based technique for solving the angular nearest neighbor search problem. The general idea behind LSH is to randomly partition the feature space using a specific family of hash functions that map similar items into the same buckets with high probability. Given such hash functions, during the preprocessing step, all items of the dataset are inserted in to l hash tables corresponding to l randomly chosen hash functions (each hash function represents a partitioning of the space). To find the nearest neighbors the query vector is similarly mapped l times, and the items in the corresponding l hash buckets are retrieved as the candidates for the nearest neighbor. The algorithm then passes through the retrieved points to find the closest one to the query. This variant of LSH is often called *Single-Probe* (SP) LSH as it probes only one bucket per hash table.

We also compare the performance of AMIH with the *MultiProbe* (MP) variant of the crosspolytope LSH. Multiprobe LSH [75] is an extension of LSH that can achieve significant

space reduction via reducing the number of required hash tables. The basic idea of multi-probe LSH is to not only consider the main bucket, where the query falls, but also probe other buckets that are *close* to the main bucket in every hash table. For our comparisons, we use the multiprobe variant of the crosspolytope LSH described in [3]. The source code of this method has been made publicly available as a part of the FAst Lookups of Cosine and Other Nearest Neighbors (FALCONN) [105].

KGraph [35] performs the nearest neighbor search by building a KNN graph over the datapoints. In the graph, each node corresponds to a data point and is connected to its M nearest point where M needs to be tuned. During query phase, the algorithm starts from one of the nodes and follows the paths with shorter distances to find the approximate nearest neighbors.

Annoy [15] decomposes the search space using multiple trees to achieve sublinear search time. At each non-leaf node, a random hyperplane is formed by taking the equidistant hyperplane of two randomly selected data points. Each internal node therefore divides the space into two subspaces where each subspace contains at least one data point. Each leaf node contains a subset of datapoints that lie in the region of space defined by the leaf's ancestors. To find the nearest neighbor, the search algorithm only considers the subspaces where the query fall in. Annoy incorporates a forest of such trees to increase the probability of collision between query and its nearest neighbor in at least one leaf node.

Experimental setting

In all experiments, for single-probe crosspolytope (SP-CP) LSH, KGraph and Annoy, we use the parameter settings of *ann-benchmark* [6] which is a tool of standardizing benchmarking for approximate nearest neighbor search algorithms. The SP-CP setting in ann-benchmark incorporates a fixed value for the number of hash functions per hash tables, $k = 16$, and let l vary from 1 to 1416. For multiprobe crosspolytope (MP-CP) LSH (which is absent in ann-benchmark), we follow the parameter setting of [3]. In particular, for MP-CP, we use only 10 hash tables ($l = 10$) in each experiment. As stated in [3], the goal of this choice is to keep the additional memory occupied by LSH comparable to the amount of memory needed for storing dataset. This is perhaps the most practical and interesting scheme since large memory overheads are impossible for massive datasets. To set k (number of hash functions per hash table), we try different values for this parameter and select the one with the minimum query time. To do that, following [105], for each value of $k \in \{10 \ldots, 30\}$, we use binary search to find the minimum number of probes that results in a near-perfect recall rate (≥ 0.9), meaning that 10 percent of returned neighbors are

not exact. After fixing k and l, the number of probes per hash table is gradually increased (which results in higher recall rates) and for each value the average query time is reported.

In the following, we compare the performance of different techniques in terms of the average query time and memory requirement. Note that the memory cost reported here is the additional memory required by each technique to build its data structure (memory required to store the raw dataset is not included). The experiments of this section are executed on a single core 3.0 GHz CPU with 32 GB of memory.

Before discussing the results, we would like to note that the ann-benchmark basically is not designed for scenarios with low memory budget. We observed that the settings used for techniques such as KGraph and Annoy require an amount of memory that is much larger than the memory required to store the dataset. Also for LSH, the benchmark only uses single-probe LSH. The main reason for this choice is that, in comparison to multiprobe LSH, single-probe LSH achieves better query time when RAM budget is not a matter of concern. The memory cost of ann-benchmark is perhaps the main reason why larger datasets such as 1 billion SIFT vectors (that we used in the first experiments) are absent in the benchmark (all datasets in ann-benchmark have around 1 million points). The authors of [3] have also explicitly mentioned that the experiments of ann-benchmark are not efficient for low RAM budget scenarios [105].

Nearest Neighbor Search in Binary Space

Here we use the ANN_SIFT1M [59] dataset which consists of 1 million 128D SIFT vectors for the base set and 10000 query items. Similar to section 3.4.2, the dataset is binarized to 64-bit and 128-bit codes by applying AQBC. The binary dataset is then fed to each technique and the average query time as well as memory cost is reported.

Fig. 3.8 and Fig. 3.9 show the average query time as well as the index size (memory overhead) of each technique with respect to the recall rates. Note that AMIH is an exact algorithm in the binary space therefore its recall rate is 1. The results highlight that AMIH is significantly faster than other techniques for near-perfect recall rates. However, for longer codes the difference between AMIH and other techniques reduces. SP-CP has very fast query time for low recall rates especially in 64-bit codes. In particular, SP-CP is the fastest technique for recall rates smaller than 0.3 in 64-bit codes. The results show that LSH based techniques tend to be faster than KGraph and Annoy for both lengths of codes. The only exception is in recall rates very close to 1 for which KGraph performs better than other approximate techniques but still slower than AMIH. Another advantage of AMIH over the other techniques is the memory cost. AMIH achieves perfect recall with memory

cost that is comparable with the dataset. However, Annoy and KGraph index size can take a large amount of memory, even 100 times more than the size of dataset. Therefore, AMIH is particularly interesting when the RAM budget is quite restrictive. In fact, the high memory cost of Annoy and KGraph did not allow us to provide similar comparisons for the ANN_SIFT1B dataset. The memory cost of Annoy and KGraph remains virtually the same for different recall rates but the catch is that they require more preprocessing time to achieve higher recall rates (the preprocessing time of each technique is not shown here due to limited space).

We would like to note that most of the techniques in ann-benchmark are designed to work with real vectors and may not be necessarily optimized for binary data. Therefore, each of these techniques can be potentially implemented more efficiently to achieve better query time for binary data. However, editing the source code of all techniques in ann-benchmark would require a great deal of human effort and is beyond the scope of this study.

Nearest Neighbor Search in Real Space

If the target binary dataset is generated with binary hashing techniques, then the nearest neighbors found by AMIH are approximate with respect to the original space. For instance, the nearest neighbor found in the above experiments is not exact for the original 128D SIFT vectors. One clear advantage of the approximate techniques such as LSH and KGraph over AMIH is that they are far more general techniques that can work with many distance measures, whereas AMIH is specifically designed for binary spaces. The benefit of applying such approximate techniques in the original space is that they could potentially achieve higher recall rates (with respect to the original space). On the other hand, mapping items to binary codes significantly reduces the storage costs as well as the cost of comparing the items. Therefore, the question that arises is: what is the performance of state-of-the-art approximation techniques in the original space in comparison with AMIH applied to a binary dataset generated by a binary hashing technique?

To answer this question, we compare the performance of SP-CP LSH, MP-CP LSH, KGraph and Annoy applied to the original SIFT vectors with the performance of AMIH applied to the binary vectors generated by AQBC. It is clear that, in this setting, the precision of AMIH is highly dependent on how accurate the binary hash function can preserve the similarities. Learning hash functions to increase accuracy is an active line of research, but is not the focus of this study. Still, such a comparison can be helpful in judging the usefulness of AMIH for non-binary datasets.

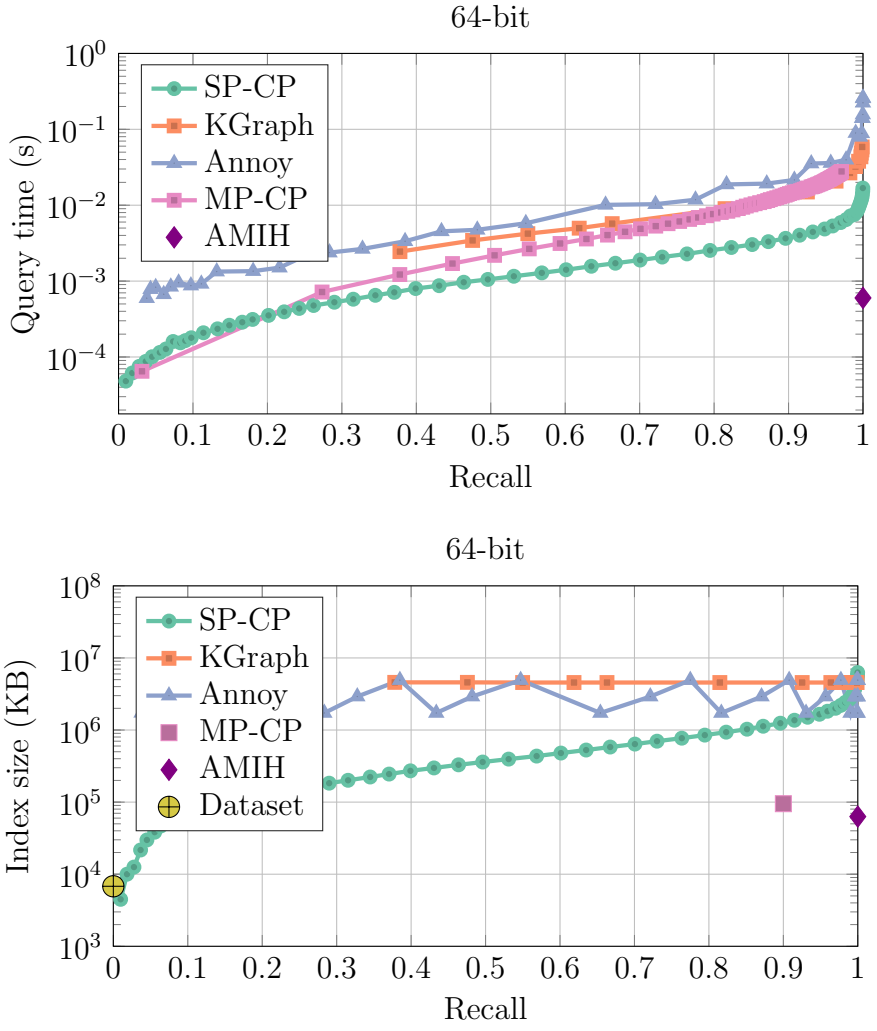


Figure 3.8: Average query time and memory overhead with respect to the recall rate for single-probe crosspolytope (SP-CP) LSH ($k = 16$) , multiprobe crosspolytope (MP-CP) LSH, Annoy, KGraph and AMIH. The memory overhead plots also show the size of dataset (the recall rate of zero for dataset size does not have a meaning). For MP-CP, the optimal value of k is 20.

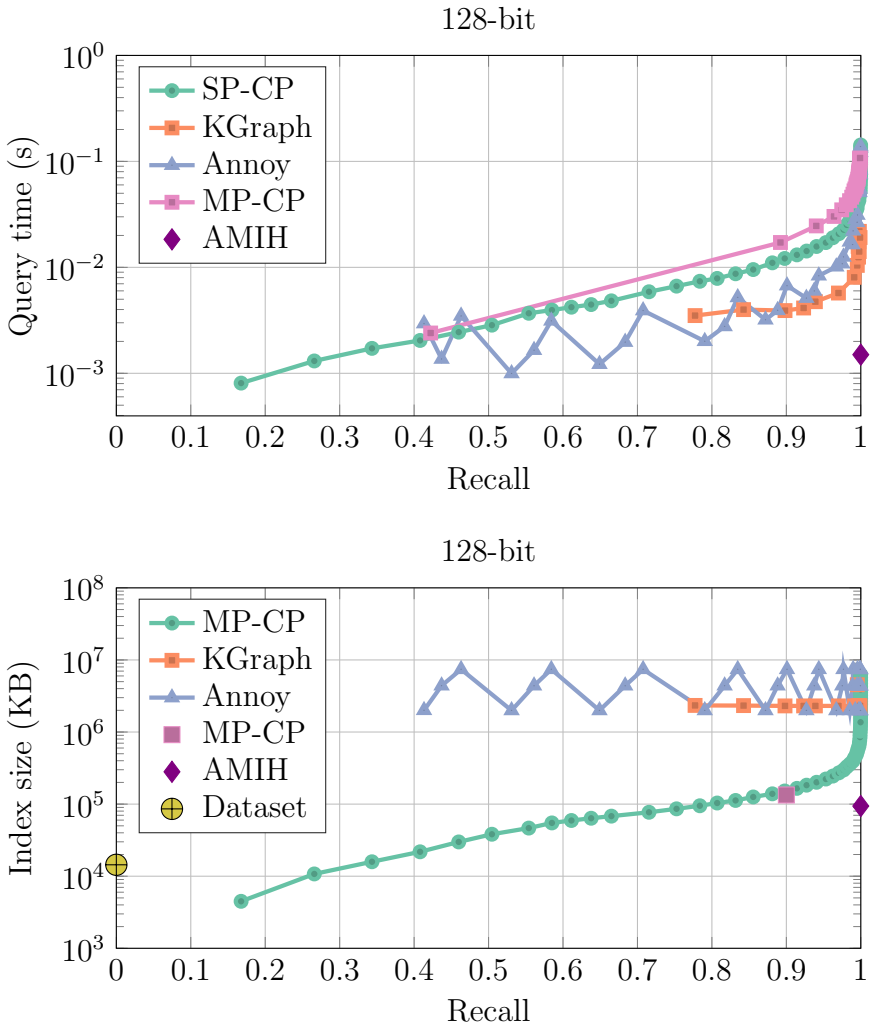


Figure 3.9: Average query time and the memory overhead with respect to the recall rate for single-probe crosspolytope (SP-CP) LSH ($k = 16$) , multiprobe crosspolytope (MP-CP) LSH, Annoy, KGraph and AMIH. The memory overhead plots also show the size of dataset (the recall rate of zero for dataset size does not have a meaning). For MP-CP, the optimal value of k is 20.

Similar to the our previous experiments, we use the ann-benchmark parameter setting for SP-CP, Annoy and KGraph. For MP-CP, we again fix the number of hash tables ($l = 10$) and chose the value of k that corresponds to the minimum query time for recall rates above 0.9. For AMIH, we increase K (the number of nearest neighbor to retrieve) from 1 to 1000 and for each value, the KNN problem is solved for each query in the binary space. Then, the algorithm linearly scans among the retrieved candidates to find the closest point to the query in the original space. Therefore, the AMIH query time reported in this setting is the summation of: i) the time required to hash the real query point into the binary space (using AQBC), ii) the time to solve KNN problem in the binary space with AMIH and iii) the time to perform linear scan among the retrieved points in the original space. This evaluation process of AMIH is similar to the MP-CP. In both, after populating the hash tables, to boost the recall rate, the search algorithm increases the number of probings per hash table in order to retrieve a larger number of candidates. Increasing probings causes better recall rates but also reduces the search speed because we have to probe more buckets and also compare more candidates with the query.

Fig. 3.10 shows the average query time and the memory overhead of each technique for the task of angular nearest neighbor search in the real space. In this case, the KGraph clearly outperforms other techniques for all tested recall rates. Among the others, AMIH 64-bit and Annoy show better average query time for many of the recall rate values. For recall rates very close to one, after KGraph, AMIH-128 consistently exhibits the fastest query time. In terms of the memory requirement, AMIH and MP-SP require constant memory budget in all experiments as the number of hash tables remains fixed. The memory footprint of SP-CP increases with recall rate due to the higher number of hash tables. Similar to previous experiments, at low recall rates, SP-CP imposes small memory cost but for recall rates greater than 0.62 AMIH 64-bit has the smallest memory overhead while achieving slightly better recall rates than SP-CP.

We would like to note that the applications of binary hashing or any other approach for compact representation is slightly different from other approximate nearest neighbor search techniques such as KGraph and LSH. Binary hashing techniques are in essence designed for extremely large datasets, too large that we are not even able to store the entire raw dataset in the memory, let alone algorithms that require superlinear storage with large exponents and constants. The goal of binary hashing is to reduce the storage cost of such large datasets in order to fit them in the memory of a single machine while still being faithful to the original metric. Unlike the setting used in ann-benchmark and the experiments of this subsection, in binary hashing applications, not only high memory overheads are not tolerated, but also the datasets itself is often absent in the memory. Moreover, the performance of AMIH with respect to the real space can be improved if more accurate

hash functions (other than AQBC) are applied to the dataset. Nevertheless, the empirical results of this section show that some approximation techniques with non-compact index structures such as KGraph, perform better when significant memory overhead is not a matter of concern.

3.5 Summary

This chapter proposes a new algorithm for solving the angular KNN problem on large-scale datasets of binary codes. By treating binary codes as memory addresses, our proposed algorithm can find similar binary codes in terms of cosine similarity in a time that grows sublinearly with the size of dataset. To achieve this, we have first established a relationship between the Hamming distance and the cosine similarity. This relationship is in turn used to solve the angular KNN problem for applications where binary codes are used as the memory addresses of a hash table. However, using a hash table for long codes is often inferior to the linear scan due to the large number of empty buckets. To tackle this issue, as the second contribution, we have proposed the AMIH technique; a multi-indexing approach to reducing both computational and storage costs in comparison to using a single hash table. We have empirically shown that the AMIH technique can increase the search speed up to orders of magnitude when applied to large-scale datasets.

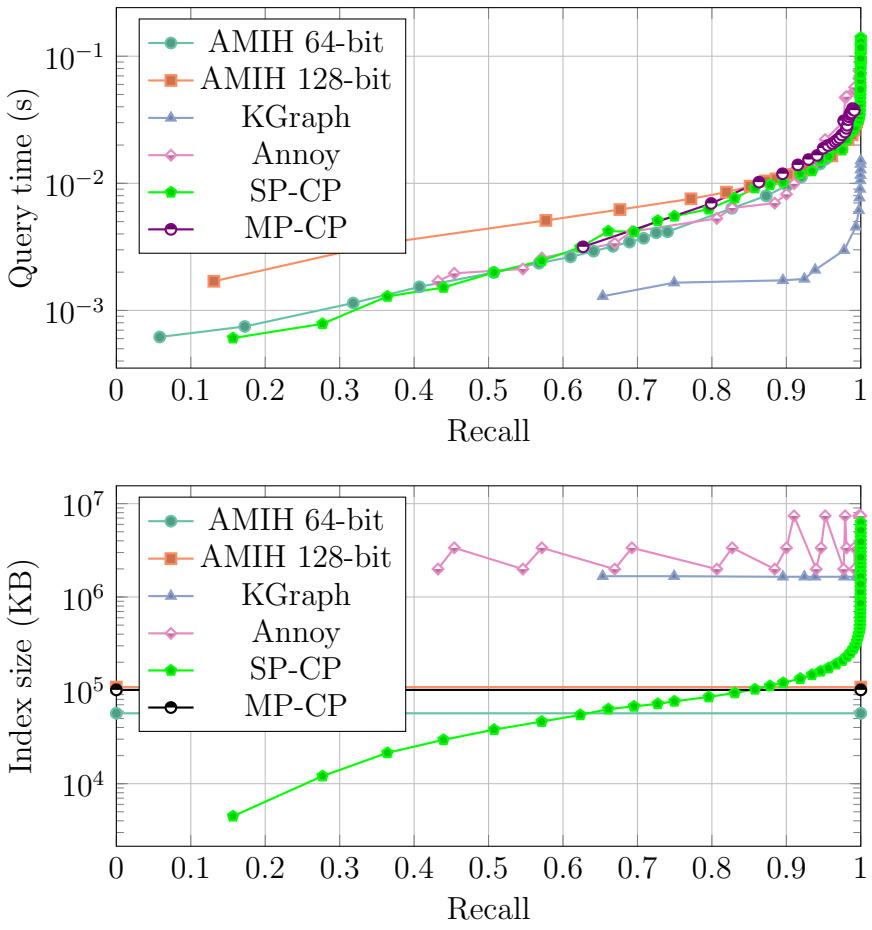


Figure 3.10: Average query time and the memory overhead with respect to the recall rate for single-probe crosspolytope (SP-CP) LSH , multiprobe crosspolytope (MP-CP) LSH for $k = 20$, Annoy, KGraph and AMIH.

Chapter 4

Online Nearest Neighbor Search Using Hamming Weight Trees

Modern real-life datasets are not only large in the number of points, but also are open-ended and dynamic; new items appear over time. For example, a search engine often has numerous new web pages containing images and textual data, that are continuously arriving at the data center everyday. In online NNS therefore, the nearest neighbor queries must be answered based on the total data that has been gathered so far. This leads to a natural question: *can we perform better than linear scan (search) for the task of large-scale KNN search in dynamic binary datasets?*

This question has not been answered adequately in the literature. In the context of CDR, some recent studies have addressed the problem of learning compact binary codes in online settings [12, 53, 54]. They have shown that it is possible to gradually refine the hash function parameters, as data points become available, such that the resulting binary codes better preserve the similarities. However, the problem of efficiently searching among the so-far collected binary codes seems to have remained unchallenged. In practice, researchers resort to linear scan to find nearest neighbors in online applications[53].

It is worth noting that MIH and AMIH do not suit online nearest neighbor search as the optimal number of hash tables relies heavily on the number of database items. In particular, both theory and practice suggest using $\frac{p}{\log n}$ hash tables to elicit the best query performance. Norouzi *et al.* [95] empirically showed that deviating from this value can incur extra work, even significantly more than what linear scan requires. Consequently, both techniques are mostly applicable for batch data in which the number of items remains constant and known.

4.1 Contributions

This chapter revisits the exact nearest neighbor search for compact binary codes in online settings. We propose a data structure, called *Hamming Weight Tree* (HWT), that enables fast and exact NNS under two different distance functions, namely Hamming distance and angular distance (cosine similarity). Equally importantly, HWT supports insertion of new items which is imperative for dealing with dynamic datasets. Our empirical experiments show that HWT achieves orders of magnitude speedup in comparison with linear scan and outperforms several best known solutions for the static setting.

This chapter is concerned with the following research questions:

RQ 4.1: How can we design data structure that supports fast Hamming nearest neighbor search for dynamic data?

RQ 4.2: How can we extend the data structure to support angular distance?

RQ 4.3: What is the empirical performance of the data structure on real-world datasets?

To address these research questions, which are related to the high-level research question RQ1 of Chapter 1, we first propose our tree-based data structure for solving the Hamming NNS and then extend its search algorithm to support angular NNS, finally we report its empirical results on large-scale binary dataset and compare it with the state-of-the-art.

4.2 Hamming Weight Tree

We start off describing our data structure by focusing on the Hamming NNS problem (to answer RQ 4.1), meaning that binary codes are compared in terms of the Hamming distance. Then, in Section 4.3, we show how the same proposed data structure can be applied to the angular NNS by modifying the search algorithm.

We address two closely related problems. Given a dataset of p-dimensional binary codes $\mathcal{B} = \{\mathbf{b}^i \in \{0,1\}^p\}_{i=1}^n$, and a binary query vector $\mathbf{q} \in \{0,1\}^p$, the first problem is the *r-neighbor* problem or *range query*, whose goal is to report all codes in $\mathcal{B}$ that are within a given distance r from $\mathbf{q}$. The second problem, K nearest neighbor, aims at finding the K codes in $\mathcal{B}$ that are closest to $\mathbf{q}$ in terms of the Hamming distance. We address both problems in their online settings; that is, the items in $\mathcal{B}$ become available sequentially and the size of dataset is unknown.

4.2.1 Depth One Tree

We first propose a data structure for solving the r-neighbor problem and then apply the data structure to solve the KNN problem. One of key ideas of this study rests on the following proposition which intuitively states that when two binary codes $\mathbf{h}$ and $\mathbf{g}$ differ by at most r bits then the difference between their *Hamming weights* is at most r where the Hamming weight of a binary code is the number non-zero entries in the code.

Proposition 4.1. *If* $\|\mathbf{h}-\mathbf{g}\|_H = r$*, then we have:*

$$|\|\mathbf{h}\|_H - \|\mathbf{g}\|_H| \in \{0, 2, \ldots, r-2, r\}. \tag{4.1}$$

where $\|.\|_H$ denotes the Hamming weight.

Proof. By definition of Hamming distance, r bits of $\mathbf{h}$ are flipped in $\mathbf{g}$. Such flips can be of types 1) zero to one, or 2) one to zero. Let r_1, r_2 denote the number of type 1 and type 2 flips respectively, thus we have $r_1 + r_2 = r$. We have:

$$\|\mathbf{h}\|_H + r_1 - r_2 = \|\mathbf{g}\|_H. \tag{4.2}$$

Using $r_1 + r_2 = r$, we have:

$$\|\mathbf{h}\|_H - \|\mathbf{g}\|_H = 2r_2 - r. \tag{4.3}$$

The proof is concluded considering the fact that $r_2 \leq r$.

■

It is easy to see that based on Proposition 4.1, for two binary codes with Hamming distance of at most r ($\|\mathbf{h}-\mathbf{g}\|_H \leq r$), the difference of Hamming weights is also at most r. Computing the Hamming weight is an extremely fast operation as many of the modern CPUs provide *popcnt* (population count) instruction which implements the Hamming weight function at the hardware level.

The significance of (4.1) stems from the fact that to solve the r-neighbor search problem for the given query $\mathbf{q}$, one needs to retrieve binary codes with Hamming weights in the set $\{\|\mathbf{q}\|_H - r, \ldots, \|\mathbf{q}\|_H + r\}$ and ignore the rest of the points. Unfortunately, the retrieved codes are not restricted to the Hamming radius of interest around the query. Hence, not all items in the target sets are r-neighbors of the query, so we need to cull any candidate that is not a true r-neighbor. For example, to answer a 2-neighbor problem for the query code $\mathbf{q}$ on a dataset of 128-bit binary codes, we can create a tree, which we call the HWT,

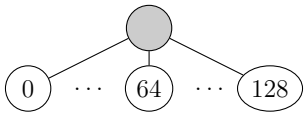


Figure 4.1: Hamming weight tree with depth one for 128-bit codes.

with 129 leaves (one for each possible Hamming weight), and assign the codes of dataset to their corresponding leaf node based on their Hamming weights (see Fig.4.1). Assuming that we have $\|\mathbf{q}\|_H = 64$, to answer the 2-neighbor query, the algorithm linearly searches among the codes belonging to nodes 62,63,64,65,66 and ignores the other 124 nodes. More generally, to solve 2-neighbor query for any query point, the algorithm needs to check at most five leaf nodes.

To create a depth-one HWT, the pre-processing step of our algorithm partitions the binary codes of $\mathcal{B}$ into $p + 1$ sets, each corresponding to one of the possible Hamming weights. Then, during the query phase, the algorithm retrieves the points in the nodes whose Hamming weight difference from $\mathbf{q}$ is at most r. Pleasingly, inserting new items to the HWT is easy as we only need to compute the Hamming weight of the new code and add it to the corresponding leaf.

An ideal scenario for solving the nearest neighbor problem using the HWT occurs when the algorithms only needs to check a few leaf nodes and such nodes contain a small portion of the dataset points. However, the pruning power of a depth-one HWT is limited in real applications, mainly due to the fact that the codes are not distributed uniformly among the nodes. While a depth-one HWT can potentially prune the search space of the r-neighbor problem and consequently use fewer Hamming distance computations compared to the linear scan, it is only beneficial for small radii of search or very long code lengths. Some problems restrict the search to exact matches [87] or small search radius, but in most cases of interest the desired search radius is large and binary codes are compact. The following two facts limit the performance of a depth-one HWT: (1) Concentration of Hamming weights: since the number of possible binary codes with Hamming weight c is $\binom{p}{c}$, Hamming weights of binary codes (both query and dataset) are highly concentrated around $p/2$. This means that the leaf nodes with Hamming weights around $p/2$ are assigned with a great portion of the points. (2) Large radii of search: solving the KNN problem often requires a not-so-small radius and thus we have to check several nodes in such cases. Because of these two observations, we often need to search among several nodes with weights around $p/2$ which unfortunately constitute a great portion of the codes, thus not much pruning can be done in such cases and the query is virtually compared with all the dataset codes.

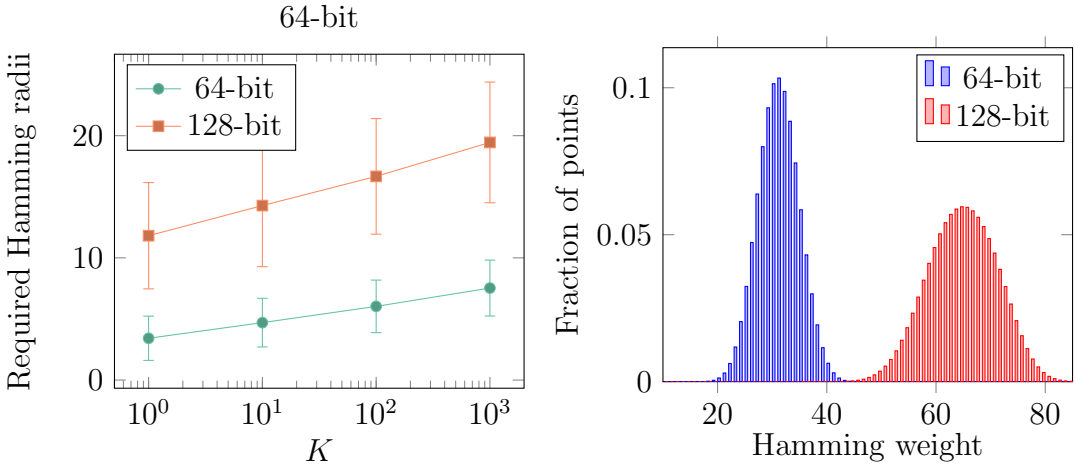


Figure 4.2: (a) Average radius of search for solving the KNN problem for different values of K. (b) Average Hamming weight of 1 billion SIFT vectors that are mapped to binary space using hyperplane LSH.

To further illustrate this problem in a real application, Fig. 4.2a shows the required radius for solving the KNN problem in the Hamming space with different values of K for a dataset of 1 billion binary codes. Fig. 4.2b shows the distribution of Hamming weights for the same dataset which clearly shows the concentration of Hamming weights around $p/2$. For instance, to solve the 10NN problem for 64-bit codes, the required search radius is 5 in average. This indicates that to search for nearest neighbors of a query with $\|\mathbf{q}\|_H = 32$ we have to look among the nodes with Hamming weights $\{27, \ldots, 37\}$ which (based on Fig. 4.2b) contain 80% of the points. The problem is that in a vast majority of cases the algorithm requires to compare the query with all of the points in several leaf nodes each storing a relatively large number of points.

4.2.2 Hamming Weight Tree on Substrings

Our approach for enhancing the pruning is to put a limit on the number of binary codes that a leaf node stores. If a node is assigned with more than τ number of points, it is split by creating multiple children and moving each binary code to its corresponding child. The children of a node are labeled based on the Hamming weights of *substrings* of the binary codes.

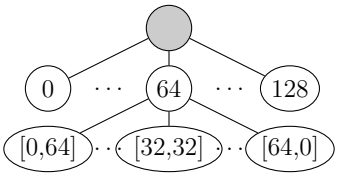


Figure 4.3: A possible configuration of Hamming weight tree with depth 2 for 128-bit binary codes.

For example, each code that belongs to the node 64 in depth-one tree of Fig. 4.1, can be partitioned into two substrings with equal lengths (the left and right 64 bits). We know that for each code that belongs to this node, the sum of the Hamming weights of the left and right substrings is 64, Therefore, we create 65 children for this node (see Fig. 4.3)—where each child is labeled with one of the possible combinations of Hamming weights for left and right substrings—and then move the codes from node 64 to their corresponding children.

In general, each binary code $\mathbf{h} \in \{0,1\}^p$, can be partitioned into d disjoints substrings, $\{\mathbf{h}_d^{(1)}, \ldots, \mathbf{h}_d^{(d)}\}$ each of length $\lfloor p/d \rfloor$ or $\lceil p/d \rceil$. For convenience, in what follows we assume that the substrings contain contiguous bits, $p = 2^t$, and p is divisible by d.

Instead of just considering the Hamming weight of the whole string, we let the tree also incorporate the Hamming weight of the substrings. To that aim, we define the vector transformation $Q_d : \{0,1\}^p \to \mathbb{N}_0^d$ as follows:

$$Q_d(\mathbf{h}) = [\|\mathbf{h}_d^{(1)}\|_H, \ldots, \|\mathbf{h}_d^{(d)}\|_H], \tag{4.4}$$

where $\mathbb{N}_0$ denotes the set of non-negative integers. Therefore, $Q_d(\mathbf{h})$ is a vector of length d with entries denoting the Hamming weights of the $\mathbf{h}$'s substrings. We call the output of this transformation the *d-Hamming weight pattern* of $\mathbf{h}$, in which the i-th entry denotes the Hamming weight of the i-th substring. For example, for the binary code $\mathbf{b} = [1,1,0,0]$, we have $Q_2(\mathbf{b}) = [2,0]$. The insight is that if two binary codes are close to each other, then their Hamming weight patterns must also be similar.

To measure the similarities between the patterns, one can use the ℓ_p norms since patterns lie in a vector space. In particular, we use the ℓ_1 distance as the measure of similarity between two patterns which corresponds to the sum of the Hamming weight differences. Formally, two binary codes $\mathbf{h}$ and $\mathbf{g}$ are said to be (r,d)-neighbor pattern of each other if we have:

$$\|Q_d(\mathbf{h}) - Q_d(\mathbf{g})\|_1 \leq r. \tag{4.5}$$

A special case is when $d = p$ for which we have that $\mathbf{h}$ and $\mathbf{g}$ are (r, p)-neighbor pattern of each other if and only if they are r-neighbors of each other.

We can now apply (4.1) to the substrings of binary codes. Formally, if $\|\mathbf{h} - \mathbf{g}\|_H = r$, we have that $\|\mathbf{h}_d^{(1)} - \mathbf{g}_d^{(1)}\|_H + \ldots + \|\mathbf{h}_d^{(d)} - \mathbf{g}_d^{(d)}\|_H = r$, therefore by applying proposition 4.1 to each of the substrings we have:

$$r - \|Q_d(\mathbf{h}) - Q_d(\mathbf{g})\|_1 \in \{0, 2, \ldots, r\}. \tag{4.6}$$

Now, reconsider the example of solving the 2-neighbor problem for the query point $\mathbf{q}$, with $\|\mathbf{q}\|_H = 64$, in the HWT shown in Figure 4.4. As mentioned, only nodes 62,. . . , 66 can contain such a neighbor. When the algorithm recurses on node 64, it descends down the tree, as it is not a leaf node. Lets assume that for the query code we have that $Q_2(\mathbf{q}) = [32, 32]$. Now, based on (4.6) it suffices to only search among the nodes [31,33], [33,31], and [32,32] while the remaining 62 children of this node can be ignored. Similarly, if the node [32,32] is later assigned with more than τ number of points, the algorithm splits it by partitioning the two substrings, $\mathbf{h}_2^{(1)}, \mathbf{h}_2^{(2)}$, into four smaller substrings, $\mathbf{h}_4^{(1)}, \mathbf{h}_4^{(2)}, \mathbf{h}_4^{(3)}, \mathbf{h}_4^{(4)}$. Fig. 4.4 shows an example of the nodes that must be visited for finding the codes lying at distance r from the query.

Formally, a HWT consists of multiple levels from -1 to l for $l \leq \log_2 p$ (for the sake of simplicity in the calculations, we assume that the depth of root is -1). Each binary code of dataset is stored in exactly one leaf node and each node at level s ($s \geq 0$) is labeled with vector $\mathbf{\Phi} = [\phi_1, \ldots, \phi_w]$ where $\phi_i \in \mathbb{N}_0$ and $w = 2^s$. The label of a node specifies the Hamming weight pattern of the codes that belong to its subtree. In other words, for each code $\mathbf{h}$ that belongs to a node $\mathbf{\Phi}$ we have that $\mathbf{\Phi} = Q_d(\mathbf{h})$.

Based on (4.1), to solve the r-neighbor problem at depth s of the tree, the algorithm only needs to recurse on the nodes with labels such as $\mathbf{\Phi} = [\phi_1, \ldots, \phi_w]$ that satisfy the following equations:

$$\|Q_{2^s}(\mathbf{q}) - \mathbf{\Phi}\|_1 \leq r \tag{4.7}$$

which is similar to (4.4). The only difference is that in (4.7), we are searching for labels of nodes (instead of binary codes) that are (r, w)-neighbors of the query. A node at depth s is called a *promising* node if its label is a $(r, 2^s)$-neighbor pattern of query.

Note that as we descend the tree, more constraints are imposed on the neighbor patterns since the algorithm incorporates piecewise Hamming weights of increasingly finer partitions of the codes. Therefore, not only the Hamming weight of the whole string must be close to the query but also the Hamming weights of the substrings cannot deviate by more than r from those of the query.

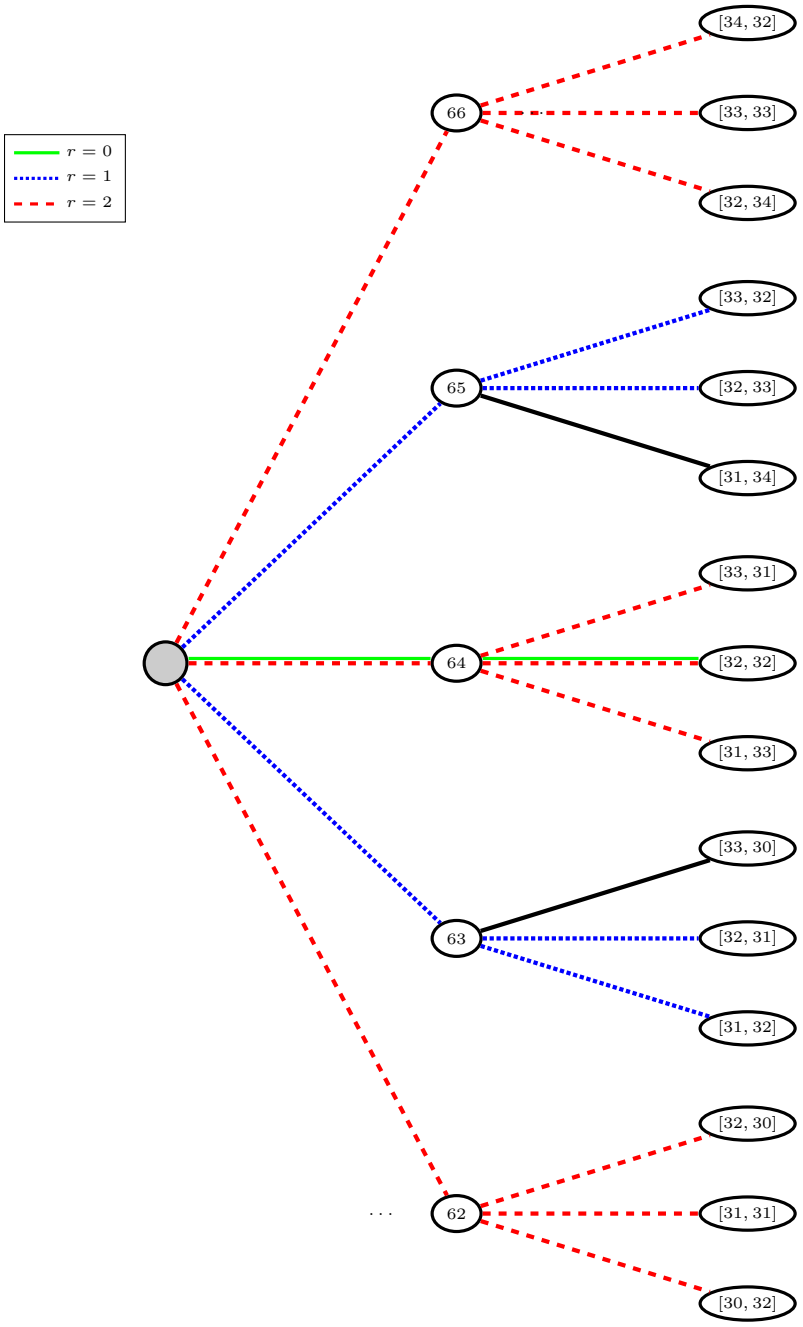


Figure 4.4: The paths that must be traversed for finding the codes lying at distance r from the query $\mathbf{q}$ with $\|\mathbf{q}\|_H = 64$, $\|\mathbf{q}_2^{(1)}\|_H = 32$ and $\|\mathbf{q}_2^{(2)}\|_H = 32$. Note that to solve the r-neighbor problem, we need to check all nodes lying at distance $r' \leq r$ from the query. For example, to solve the 2-neighbor problem in the above tree, the search algorithm must traverse all the red dashed paths.

In the following, we describe the insert and search operations of HWT in more details.

Insert. On the arrival of a new binary code such as **h**, based on the Hamming weights of **h** and its substring, we descend the tree until a leaf node is reached. To descend from a node at level s to a node at level $s+1$, the 2^{s+1}-Hamming weight pattern of **h** is computed and the child whose label matches the tuple is selected. Therefore, each particular point only participates in one branch of recursion during insertion. Upon reaching a leaf node, the code **h** is added to the node if the leaf node is not full. Otherwise, to split a full leaf node at depth s, the algorithm computes the 2^{s+1}-Hamming weight pattern of the binary codes stored at this node, and then moves each of the codes to its corresponding child based on the pattern. Finally, the code **h** is similarly added to its corresponding child.

The branching factor of an internal node at depth s is $(\phi_1+1)\times\ldots\times(\phi_{2^s}+1)$. Although the branching factor can get quite large for deep nodes, in high depths, most of children do not store any code. Consequently, instead of initializing all children of a node at once, we use *lazy* initialization to avoid memory allocation for empty children. To do that, rather than storing all children (empty and non-empty), we define an ordering for the children's labels and assign an index to each one. Then, for each node, a dynamic hash table is used to store key-value pairs where indices of non-empty children serve as keys, and the values are the pointers to the children. This reduces the storage cost as we only need to store the non-empty children. Meanwhile, insertion, deletion and searching for a child still can be performed in amortized constant time.

Search. r-neighbor search on a HWT can be answered by proceeding recursively, starting at the root. The search procedure descends through the tree level by level, keeping track of the subset of nodes that may contain the r-neighbors of the query. When visiting an internal node, the algorithm only recurses on the children whose label satisfy (4.7). Thus, starting from the root node, at each depth such as s, the search procedure only investigates the non-empty children whose labels are $(r, 2^{s+1})$-neighbor patterns of the query. There are two options for finding the non-empty children that satisfy (4.7):

- Option 1: Simply iterate through all children and recurse on those whose pattern satisfy (4.7).
- Option 2: First find the index of all children that satisfy (4.7) and then recurse on those that exist in the hash table.

For the second approach, the algorithm must enumerate over all promising children of the current node. To that aim, at node j labeled $[\phi_1, \ldots, \phi_w]$, we find all solutions of the

following system of equations:

$$\begin{cases} x_1 + x_2 = \phi_1 \\ \dots \\ x_{2w-1} + x_{2w} = \phi_w \\ \sum_{i=1}^{2w} |\|\mathbf{q}_{2w}^{(i)}\|_H - x_i| \leq r \\ x_i \in \mathbb{N}_0, \end{cases} \tag{4.8}$$

Each solution vector, $[x_1, \dots, x_{2w}]$, denotes a label of a child that we need to recurse on (provided that it exists in the tree). The equations of the form $x_{2i-1} + x_{2i} = \phi_i$ are necessary to make sure that the solutions are the labels of j's children. The inequality, on the other hand, is necessary to ensure that the solutions are the r-neighbor patterns of $\mathbf{q}$. Proposition 4.2 describes an efficient procedure for solving (4.8).

Not surprisingly, there is a natural trade-off between the two options. For small radii of search and in low depths, the number of solutions to (4.8) is small and therefore it is computationally more efficient to use option two. On the other hand, if a node has a small number of children then it is often more efficient to use option 1.

In our implementation, we use the following lower bound on the number of children to decide between the two options:

Proposition 4.2. *The number of solutions for* (4.8) *is greater than:*

$$\sum_{r'=0}^{\lfloor \frac{r}{2} \rfloor} \binom{w + r' - \sum_{i=1}^{w} |\phi_i| - 1}{w - 1}. \tag{4.9}$$

Proof. The unknowns of (4.8) come in pairs, tied together only by the last inequality. To solve it, one can consider $r + 1$ systems of equations, one for each possible value of $\sum_{i=1}^{2w} |\|\mathbf{q}_{2w}^{(i)}\|_H - x_i| = r', r' \in \{0, \dots r\}$, and solve them independently.

Now consider one of such systems in which $\sum_{i=1}^{2w} \|\mathbf{q}_{2w}^{(i)}\|_H - x_i| = r'$. Given r'_k and ϕ_k $(0 \leq k \leq w)$, the following system of equations can be solved easily:

$$\begin{cases} x_{2k-1} + x_{2k} = \phi_k \\ |\|\mathbf{q}_{2w}^{(2k-1)}\|_H - x_{2k-1}| + |\|\mathbf{q}_{2w}^{(2k)}\|_H - x_{2k}| = r'_k. \end{cases} \tag{4.10}$$

which has a set of solutions only if i) $r'_k \geq |\phi_k|$ and ii) $r'_k \equiv \phi_k \bmod 2$. For each k, it can be solved by considering four cases according to whether x_{2k-1} and x_{2k} are positive

or negative. To recover all solutions for a specific value of r', we need to iterate through all possible values of r'_ks and for each iteration we must solve an instance of (4.10). In other words, we need to generate all possible distributions of r' among the w equations. The total number of distributions for a specific r' is essentially the number of partitions of parameter r' into w non-negative integers, $r_1 + \ldots + r_w = r'$, given by the *multiset coefficient*:

$$\left(\!\binom{r'+1}{w-1}\!\right) \triangleq \binom{w+r'-1}{w-1}. \tag{4.11}$$

Thus, the total number of instances of (4.10) that we need to solve is:

$$\sum_{r'=0}^{r} \left(\!\binom{r'+1}{w-1}\!\right). \tag{4.12}$$

However, many of the instances do not yield a solution as they can violate constraints (i) or (ii). What we show is that we can skip those instances with a simple approach. To skip the instances that violate constraint (i), we can first find all partitions of $r_1 + \ldots + r_w = r' - \sum_{i=1}^{w} |\phi_i|$ and then for each partition add the $|\phi_i|$ to its corresponding r_i. This reduces the number of instances to $\sum_{r'=0}^{r} \left(\!\binom{r'+1-\sum_{i=1}^{w}\phi_i}{w-1}\!\right)$.

Now, to also ensure that all partitions satisfy constraint (ii), we can limit the radius to $r/2$ which would result in 4.9. In simple terms, instead of finding all $\sum_{r'=0}^{r} \left(\!\binom{r'+1-\sum_{i=1}^{w}\phi_i}{w-1}\!\right)$ solutions and then discarding those not satisfying the $r'_k \equiv \phi_k \bmod 2$ condition, one can first find all solutions of $r'_1 + \ldots + r'_w = r'/2 - \sum_{i=1}^{w} |\phi_i|$ and then for each solution multiply all of the r'_is ($1 \leq i \leq w$) by two. Finally, the entries that must be odd are added by one. Going from (4.12) to (4.9) saves us a lot of unnecessary computation as the first quantity is much bigger.

Using this approach, each generated instance of (4.10) has at least one solution which directly translate to that (4.9) is a lower bound for the number of children that must be checked.

■

We use this lower bound such that, at node j, if the number of non-empty children is less than (4.9), then the algorithm proceeds with option 1 otherwise, it proceeds with option 2.

4.2.3 Storage and Computational Costs

We next analyze the storage and computational costs of HWT. Storing the dataset of binary codes requires $O(np)$ bits. The storage cost of the tree comprises the number of nodes in the tree plus the storage cost of the hash table per node. For each code in a leaf node, we need an identifier that refers to the code in the dataset. This allows one to store the identifier of a code in its corresponding leaf and fetch the full code when necessary. Thus, the total cost of storing identifiers would be $n \log_2 n$. The maximum number of nodes happens when $\tau = 1$ which forms a tree with n leaves in which each leaf stores only one code (provided that there is no duplicate). Assuming that each internal node has at least two children, the number of internal nodes is at most $n-1$. Therefore, the tree has at most $2n-1$ nodes. The total cost of hash tables is also bounded by the number of nodes in tree, since each hash table only stores non-empty children. Therefore, the number of nodes in the tree is linear in the number of points.

The storage cost of hash tables depend on the length of keys which in our application represent the index of the nodes. In general, the required length of indices gets longer as we descend in the tree. The number of possible children of a node at a certain depth depends on both the depth itself and the label of node. It is easy to see that for a node at depth s, the maximum number of children belongs to the node with pattern $[\frac{p}{2^{s+1}}, \ldots, \frac{p}{2^{s+1}}]$, and the number of children for this pattern is:

$$I(s) = (\frac{p}{2^{s+1}} + 1)^{2^s} \tag{4.13}$$

Considering the fact that for an internal node we have $s < \log p$, the maximum of $I(s)$ occurs at $s = \log p - 1$. Therefore, assuming $p = 2^t$, the number of bits required to index a child of a node is upper bounded by:

$$\log(\max(I(s))) = 2^{t-1} \log(\frac{p}{2^t} + 1) = \frac{p}{2}, \tag{4.14}$$

which is of $O(p)$. Since the number of node indices in the hash tables is n, the total storage complexity of HWT is of $O(np + n \log n) = O(np)$.

Interestingly, the storage cost of HWT is the same as linear scan and better than multi-index hashing technique proposed in [49] (with cost of $O(rn^{1+r/p} \log n)$ for solving r-neighbor search) and the same as those in [95, 40].

The insertion time of HWT is also appealing. Starting from the root, at each depth, we just need to compute the pattern of the code at that depth which can done in $O(p)$. Retrieving a pointer to a specific child at a node can be done in amortized $O(1)$ as we are

using hash tables. Since the maximum depth of tree is $O(\log p)$, the total cost of inserting a new item is $O(p \log p)$. Finally, each insertion in the worst case can trigger reinsertions of τ other items but for a fixed τ the cost is still of $O(p \log p)$.

We also show that, for uniformly distributed binary points, the computational cost of r-neighbor search is logarithmic in the number data points.

Theorem 4.1. *[Search Complexity for Uniform Data] Let X_n be a set of n points, generated independently from the uniform distribution over the p-dimensional binary cube $\{0,1\}^p$. Then, the expected cost of a single r-neighbour search over the Hamming Weight Tree built on X_n is $O(p \log p (\log n)^{4r})$.*

Therefore, the query cost is logarithmic in the number of data points for small radii. The exponential dependency on r can be reduced by drawing upon similar techniques used in [49, 95, 144]. The idea is that, given a data structure that solves the r-neighbor problem in sublinear time, one can create several such data structures (say m of them) on the substrings of binary codes (in our case it would be a forest of Hamming weight trees with m trees on the substrings). Based on the pigeonhole principle, instead of solving the r-neighbor problem on the whole binary code, one can solve m number of $\frac{r}{m}$-neighbor problems, one per substring, and then aggregate the results to retrieve the neighbors. While our current implementation of HWT supports search on multiple trees, theoretical and empirical analysis of this idea is out of scope of this study and we postpone it to future works.

4.2.4 K Nearest Neighbors Search

HWT is inherently designed to answer r-neighbor queries. Consequently, each search query to this data structure should contain the query vector and the radius of search, however, the nearest neighbor search does not provide the radius in the query, making it a harder problem than the r-neighbor problem. It turns out that for the nearest neighbor of query, the required radius of search for two different query points may vary significantly, depending on how dense the area around the query is populated. Even for a single dataset, the required radius of search for different queries may vary dramatically [43, 94]. If the search radius is set too small, then the algorithm may return no points. On the contrary, large values of radius can result in non-informative neighbors. Moreover, for a large radius of search, the needed time to retrieve the neighbors would be high. Therefore, it is natural for many tasks to fix the number of neighbors and let the radius depend on the query and dataset distribution. Fortunately, a careful implementation of the proposed HWT can

be adapted to accommodate the nearest neighbor search queries. Given a query point, starting from radius search of zero ($r = 0$), one can progressively increase r until the nearest neighbors are retrieved.

In a naive implementation of HWT, when the radius increases, the new r-neighbor search starts from scratch and we have to check all the nodes that may contain the r-neighbors in their subtrees. However, many of such nodes overlap with those checked for smaller values of r. In fact, when r increases, the algorithm have already checked all the nodes that can contain codes with any distance less than r from query. Therefore, we just need to search for the codes that lie at exact distance of r from the query. More specifically, it is easy to see that, all the nodes that must be visited for solving the r-neighbor problem, must be also visited for solving the $(r + 2)$-neighbor problem (see Fig. 4.4 for $r = 0$ and $r = 2$). To avoid such extra checking, one can store a list of identifiers to the so far internal visited nodes along with their radius of search for which the specific node was visited. Then, to solve the $(r + 2)$-neighbor problem, the algorithm iterates through the list and for each node recurse on children that may contain the codes with distance $r + 2$ from the query (refer to 4.6). By doing so, the algorithm can skip many of edge traversals when the radius increases.

4.3 Angular Nearest Neighbor Search

Although most of studies on the binary codes adopt Hamming distance as the measure of similarity between binary codes, in some applications the angle between the code arises as a more effective alternative [13, 46, 117]. For example, AQBC [46] is a binary hashing technique in which the appropriate similarity measure is the cosine of the angle between the binary codes. In its basic form, AQBC maps real valued feature vectors onto the vertex of binary hypercube with which it has the smallest angle. The distance between the resulting codes are then defined as the cosine of angle between them. Recall that the Hamming NNS search on HWT was performed by solving multiple instance of r-neighbor query. Pleasingly, HWT can be used to carry out angular KNN without modifying the data structure and again by only solving r-neighbor queries during the query phase using the sequential algorithm developed in Chapter 3. The only difference from the Hamming distance case is that for the angular case the search executes r-neighbor queries on some particular nodes and in an altered order discussed in the following. This will answer RQ 4.2.

The cosine of the angle between two binary vectors can be computed as follows:

$$sim(\mathbf{q}, \mathbf{b}) = \cos(\theta(\mathbf{q}, \mathbf{b})) = \frac{\mathbf{q}^T \mathbf{b}}{\|\mathbf{q}\|_2 \|\mathbf{b}\|_2} \tag{4.15}$$

Computing the cosine similarity between two binary codes is marginally slower than computing Hamming distance, however it is still fast compared to computing similarity of the original vectors. As shown in [40], for binary vectors, we can rewrite (4.15) with:

$$sim(\mathbf{q}, \mathbf{b}) = \frac{\|\mathbf{q}\|_1 - r_1(\mathbf{q}, \mathbf{b})}{\sqrt{\|\mathbf{q}\|_1} \times \sqrt{\|\mathbf{q}\|_1 - r_1(\mathbf{q}, \mathbf{b}) + r_2(\mathbf{q}, \mathbf{b})}} \tag{4.16}$$

where $r_1(\mathbf{q}, \mathbf{b}) \triangleq \|\mathbf{q} \wedge \neg \mathbf{b}\|_H$ denotes the number of bits that are 1 in $\mathbf{q}$ and 0 in $\mathbf{b}$ and $r_2(\mathbf{q}, \mathbf{b}) \triangleq \|\neg \mathbf{q} \wedge \mathbf{b}\|_H$ denotes the number of bits that are 0 in $\mathbf{q}$ and 1 in $\mathbf{b}$ (where $\neg$ and $\wedge$ are bitwise negation and logical AND operators). Therefore, all codes with the same value of $r_1(\mathbf{q}, \mathbf{b})$ and $r_2(\mathbf{q}, \mathbf{b})$ lie at the same angle from the query. We say that the code $\mathbf{b}$ *lies at the distance tuple* (e, f) *from* $\mathbf{q}$ or equivalently $\mathbf{b}$ *is a* (e, f)*-neighbor of* $\mathbf{q}$, if we have $(e, f) = (r_1(\mathbf{q}, \mathbf{b}), r_2(\mathbf{q}, \mathbf{b}))$.

As discussed previously, to carry out KNN search in the Hamming space, one can increase the search radius until K neighbors are retrieved. To employ a similar approach for solving the angular KNN, we must first find the correct sequence of tuples that corresponds to the decreasing values of sim and then for each tuple such as (e, f) the HWT must be searched to retrieve the binary codes lying at distance tuple (e, f). The correct sequence of tuples can be found efficiently using the sequential algorithm proposed in [40]. We next show how to search the HWT in order to find codes lying at a specific distance tuple.

Note that the binary codes that correspond to the tuple (e, f) lie at the Hamming distance $e + f$ from the query. Therefore, a simple solution for retrieving desired codes involves solving the r-neighbor problem with $r = a+b$ and then linearly scanning retrieved candidates to find the codes corresponding with tuple (e, f). However, this approach may search some unnecessary nodes. For example, suppose we are looking for codes lying at distance tuple $(2, 2)$ from the query with $\|\mathbf{q}\|_1 = 64$ in a depth-one Hamming weight tree for 128-bit codes. Our algorithm would search among the nodes with Hamming weights 60,62,64,66,68 to find the codes lying at Hamming distance of 4 from the query. However, it is easy to see that only node 64 must be searched because codes lying at distance tuple (2,2), have equal Hamming weights to that of the query.

To find all (e, f) neighbors of $\mathbf{q}$, we need to determine the nodes that the search algorithm must recurse on. To this aim, we first show the relationship between the Hamming weight patterns of two codes lying at a specific distance tuple.

Proposition 4.3. *For binary code* $\mathbf{b}$ *lying at distance tuple* (e, f) *from* $\mathbf{q}$*, we have:*

$$\sum_{i=1}^{d} \left[\|\mathbf{q}_d^{(i)}\|_H - \|\mathbf{b}_d^{(i)}\|_H \right]_+ \leq e, \tag{4.17}$$

$$\sum_{i=1}^{d} \left[\|\mathbf{b}_d^{(i)}\|_H - \|\mathbf{q}_d^{(i)}\|_H \right]_+ \leq f, \tag{4.18}$$

$$\|\mathbf{q}\|_H + (f - e) = \|\mathbf{b}\|_H, \tag{4.19}$$

where $[.]_+ = \max(0, .)$ *is the standard hinge loss function.*

Proof. If (4.17) is not satisfied then at least $e + 1$ set bits in $\mathbf{q}$ are flipped to zero in $\mathbf{b}$ which contradicts the fact that $\mathbf{b}$ lies at distance tuple (e, f) from $\mathbf{q}$. Also, if (4.18) is not satisfied, at least $f + 1$ clear bits in $\mathbf{q}$ are flipped to one in $\mathbf{b}$. (4.19) can be easily derived from the fact each distance tuple uniquely specifies the Hamming weights of the codes of interest. ■

To find all the points that lie at the distance tuple (e, f) in a Hamming weight tree, we again have two options: i) scan all children and recurse on nodes satisfying conditions (4.17)- (4.19), ii) compute all possible Hamming weight patterns that satisfy (4.17)-(4.19) and directly recurse on those nodes.

To use the second option, at a HWT node with Hamming weight pattern of $[\phi_1, \ldots, \phi_w]$, the algorithm finds the promising children by solving the following system of equations and the recurse on the children with the Hamming weight pattern of $[x_1, \ldots, x_{2w}]$.

$$\begin{cases} x_1 + x_2 = \phi_1 \\ \ldots \\ x_{2w-1} + x_{2w} = \phi_w \\ \sum_{i=1}^{2w} \left[\|\mathbf{q}_d^{(i)}\|_H - x_i \right]_+ \leq e \\ \sum_{i=1}^{2w} \left[x_i - \|\mathbf{q}_d^{(i)}\|_H \right]_+ \leq f \\ x_i \in \mathbb{N}_0 \\ \|\mathbf{q}\|_H + (f - e) = \sum_i^{2w} x_i \end{cases} \tag{4.20}$$

The last condition in (4.20) can be easily satisfied by simply forcing the search algorithm to recurse on only one node at the first level which has the pattern of $\|\mathbf{q}\|_H+(f-e)$. Doing that, (4.20) can be rewritten as:

$$\begin{cases} \sum_{i=1}^{w} \left[\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right]_+ + \left[\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right]_+ \leq e \\ \sum_{i=1}^{w} \left[x_{2i-1} - \|\mathbf{q}_d^{(2i-1)}\|_H\right]_+ + \left[\phi_i - x_{2i-1} - \|\mathbf{q}_d^{(2i)}\|_H\right]_+ \leq f \\ x_i \in \mathbb{N}_0 \end{cases} \tag{4.21}$$

By using the fact that $[a]_+ = \frac{1}{2}(a+|a|)$, we have:

$$\begin{cases} \sum_{i=1}^{w} \left|\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right| + \left|\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right| \leq \\ \qquad 2e - \sum_{i=1}^{w} \left(\|\mathbf{q}_d^{(2i-1)}\|_H + \|\mathbf{q}_d^{(2i)}\|_H - \phi_i\right) \\ \sum_{i=1}^{w} \left|\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right| + \left|\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right| \leq \\ \qquad 2f + \sum_{i=1}^{w} \left(\|\mathbf{q}_d^{(2i-1)}\|_H + \|\mathbf{q}_d^{(2i)}\|_H - \phi_i\right) \\ x_i \in \mathbb{N}_0 \end{cases} \tag{4.22}$$

Considering that $\sum_{i=1}^{w} \left(\|\mathbf{q}_d^{(2i-1)}\|_H + \|\mathbf{q}_d^{(2i)}\|_H - \phi_i\right) = e - f$, solving (4.20) reduces to finding all solutions of the following inequality:

$$\sum_{i=1}^{w} \left|\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right| + \left|\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right| \leq e + f \tag{4.23}$$

which is same as (4.8) with $r = e + f$. Thus, the search algorithm recurses on the nodes satisfying two conditions. First, they should satisfy the weight condition, $\|\mathbf{q}\|_H+(f-e) = \sum_{i=1}^{w} \phi_w$. Second, they must be a $(e + f, w)$-neighbor pattern of the query. The first condition is satisfied by making sure that at the first level the algorithm selects the node with weight $\|\mathbf{q}\|_H + (f - e)$. Then, to satisfy the second condition, the algorithm solves the $(e + f)$-near neighbor problem on the subtree of the selected node. Therefore, as in the Hamming KNN, angular KNN search can performed by only solving instances of the r-neighbor problem.

4.4 Experiments

To answer RQ 4.3, in this section we empirically gauge the performance of HWT in comparison with the linear scan baseline, MIH of [95], AMIH of [40], and four popular tree-based search algorithms namely Annoy, kd tree, ball tree and RP forest. The following experiments are run on a single core 2.0 GHz CPU with 128 GB of RAM. Linear scan and HWT are both coded in C++ and compiled with GCC 4.4.4 using the same flags. We used the publicly available implementation of MIH and AMIH in our experiments. Our implementation of HWT is also available at https://github.com/sepehr3pehr/hwt.

4.4.1 Datasets

We evaluate the performance of HWT on two well-known real-world datasets which are publicly available: (1) ANN_1B [60] with 1 billion 128D SIFT vectors, and (2) 80 millions 384D GIST descriptors from the 80 million tiny images [123]. Each experiment requires two sets of items; base set for populating HWT and the query set that comprises the query points. For 80M Gist descriptors, we randomly select 1000 points to form the query set and use the remaining as the base set. The ANN_1B corpus is already divided into 1 billion base data points and 10^4 query points from which we randomly select 1000 query points. Therefore, each experiment involves 1000 queries for which the average run-time is reported.

To map real-valued SIFT and GIST vectors to binary codes, for the Hamming distance experiments, we use the well-known hyperplane LSH [25] which utilizes sign-random projection. More specifically, after zero-centering the data, to encode each bit, first a random hyperplane is selected where each component of the direction is generated from a normal density, then the value of the bit is specified depending on which side of the hyperplane the point lies. For the angular distance, we implemented AQBC [46] and applied on the real-valued vectors to produce angular preserving binary codes. We also make our implementation of AQBC publicly available (https://github.com/sepehr3pehr/AQBC). As opposed to the hyperplane LSH which is a randomized technique with no learning phase, AQBC is data-depedent and requires the parameters of the hash function to be learned. The ANN_1B dataset comes with a predefined *learning* set of 100 million SIFT vectors from which we randomly select one million points. Similarly, for 80M Gist descriptors dataset, we randomly select 300K points from the dataset to form the learning set.

For each dataset and similarity measure, we generate 32, 64 and 128 bit binary datasets. With two datasets, three different code lengths, and two distance measures, we obtain 12

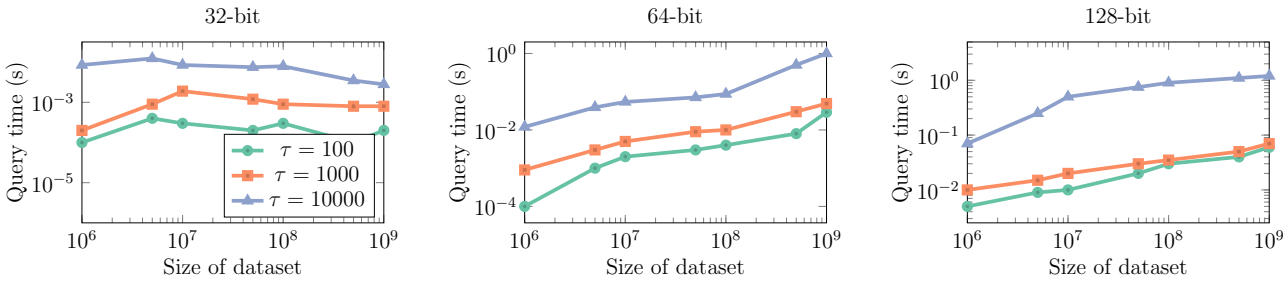


Figure 4.5: Average query time of the nearest neighbor search for τ=100, 1000 and 10000 on ANN_1B dataset.

binary datasets.

4.4.2 Results

Effect of Threshold Value

We first investigate the effect of parameter τ on the average query time. This parameter determines the maximum number of binary codes that can be assigned to a leaf node. We have a natural trade-off for different settings of this parameter. Large values of τ form shallow trees and therefore less pruning takes place which increases the required number of distance computations. Meanwhile, for each query, fewer node traversals and child checkings are required. In the extreme case, we can create a depth-one tree by setting τ to be sufficiently large. Such a tree exhibits a performance similar to linear scan. On the other hand, small values of τ create trees with higher depths which causes further pruning but more node traversals and higher storage cost.

Fig. 4.5 shows the average query time for different values of τ. The figure indicates that smaller values of τ result in a faster query time. This shows that further pruning of the search space often results in a better average query time, even with the overhead imposed for finding the promising children of nodes. Nonetheless, since we create a hash table for each internal node, we observed that the memory footprint increases as we decrease τ. Interestingly, this parameter can be set based on the available memory of the target platform to balance the query time and memory requirement. We observed similar patterns for the angular distance but the results are omitted due to the space limit.

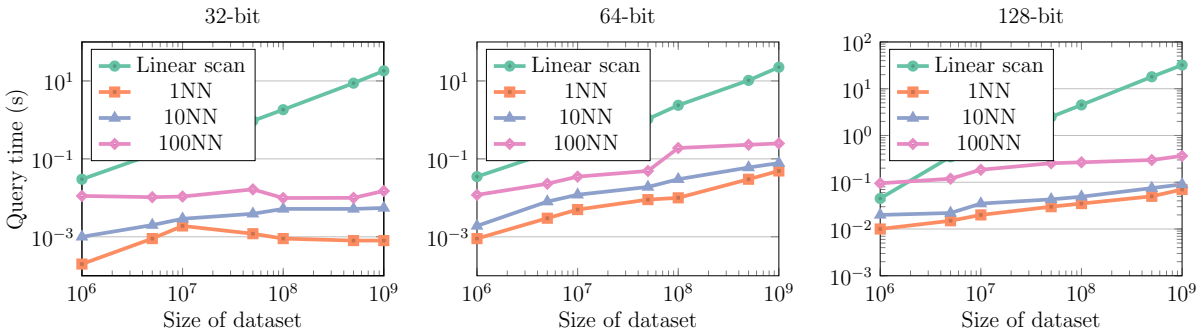


Figure 4.6: Average query time of HWT and linear scan on the ANN_1B dataset for solving the Hamming NNS.

Note that in some limited cases the query time decreases for larger sizes of dataset. This is mainly due to the fact that increasing the number of points often makes the required search radius for retrieving the nearest neighbor smaller. This in turn lets the tree to search among a fewer number of nodes. However, the dominant trend is that the query time increases with the size of dataset.

For the following experiments, we set $\tau = 1000$. For this choice, our current implementation of HWT requires 50 GB, 62 GB, and 73 GB of memory to index 1 billion 32-bit, 64-bit, and 128-bit codes, respectively, which is comparable with that of MIH [95].

HWT vs Linear Scan

In this experiment, we focus on comparing the average query time of HWT and linear scan on all datasets. First, we report the average query time when all items are inserted in the tree (batch data), and then we illustrate the query time when the data is inserted sequentially (online data). Table 4.1 reports the average query time of the linear scan baseline and HWT along with speed up factors gained by using HWT for different KNN problems. For a large range of code lengths and different values of K, HWT can achieve orders of magnitude speed up in comparison with the linear scan. Note that the running time of linear scan neither depends on K nor on the underlying distribution of points, however, both factors affect HWT. As K increases, the required search radius also increases which causes longer query time. This is reflected in the reduction of speed up factors when the value of K increases. Also for longer codes, the difference between HWT and linear scan becomes smaller.

Table 4.1: Average running time of nearest neighbor search with HWT and linear scan (LS) algorithms on ANN_1B and GIST 80M.

				Hamming		**Angular**	
	#bits	**Method**	K	**Time**	**Speedup**	**Time**	**Speedup**
ANN_1B	32	LS	-	18.14	1×	30.65	1×
		HWT	1	0.0008	22675×	0.0084	3649×
		HWT	10	0.0055	3088×	0.0099	3095×
		HWT	100	0.015	1029×	0.034	901×
	64	LS	-	22.47	1×	31.78	1×
		HWT	1	0.049	458×	0.029	1095×
		HWT	10	0.078	150×	0.084	378×
		HWT	100	0.249	90×	0.39	81×
	128	LS	-	32.11	1×	49.34	1×
		HWT	1	0.07	459×	0.25	197×
		HWT	10	0.09	356×	0.38	129×
		HWT	100	0.366	88×	0.79	62×
GIST 80M	32	LS	-	1.02	1×	3.05	1×
		HWT	1	0.003	340×	0.006	508×
		HWT	10	0.005	204×	0.009	338×
		HWT	100	0.003	340×	0.012	254×
	64	LS	-	1.22	1×	3.79	1×
		HWT	1	0.009	113×	0.019	199×
		HWT	10	0.015	81×	0.037	102×
		HWT	100	0.053	23×	0.077	49×
	128	LS	-	2.5	1×	4.93	1×
		HWT	1	0.08	31×	0.15	32×
		HWT	10	0.180	16×	0.39	12×
		HWT	100	0.5	5×	0.71	7 ×

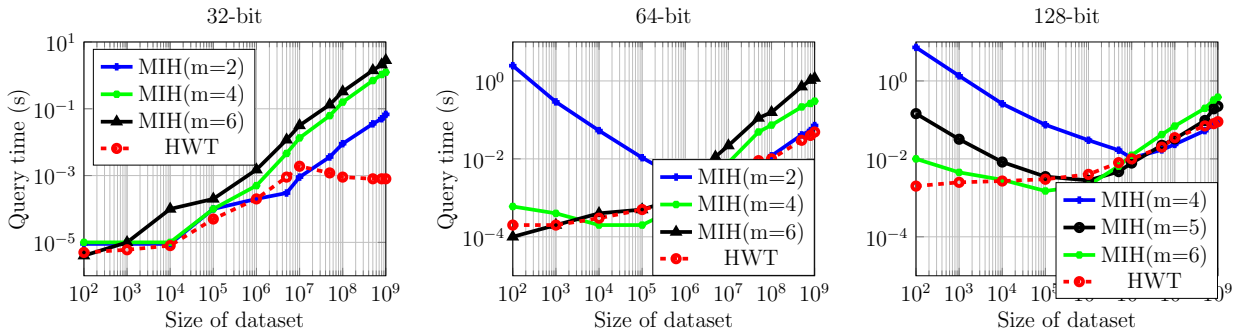


Figure 4.7: Average query time of HWT and MIH for the task of Hamming nearest neighbor search. The value of m denotes the number of hash tables used in MIH.

HWT vs MIH

We also compare the performance of HWT with MIH [95] and angular MIH (AMIH) [40] which to the best of our knowledge have the best query time for solving the exact NN problem for Hamming and angular distances. To set the number of hash tables for MIH, Norouzi et. al [94] used a hold-out validation set of the dataset entries. From that set, the running time of the algorithm for different values of m (number of hash tables) is estimated, and the one with the best result is selected. They empirically observed that the optimal value for m is typically close to $p/\log_2 n$. However, in online settings the data points become available sequentially thus the value of n varies over time and the items of dataset are not available in advance.

In our experiments, we execute MIH and AMIH with different values of m for different sizes of the dataset to investigate the relative performance of these two techniques. Not surprisingly, there is a natural trade-off between large and small values of m. Too large values result in assigning fewer number of bits to each table. Therefore, each bucket of hash table can be assigned with several codes and consequently the query must be compared with more codes. In the extreme case if one bit is assigned to each hash table then we have $m = p$ and the query must be compared with all points. On the other hand, too small values lead to forming hash tables with many empty buckets. In this case, to retrieve K nearest neighbors, the required radius of search per hash table must be increased. This translates to checking many empty buckets.

Fig. 4.7 and Fig. 4.8 show the average query time of MIH vs HWT for Hamming distance and HWT vs AMIH for the angular distance applied to the task of nearest neighbor search (1NN). For MIH and AMIH, we tried all values of $m \in \{1, \ldots, 10\}$ and measured

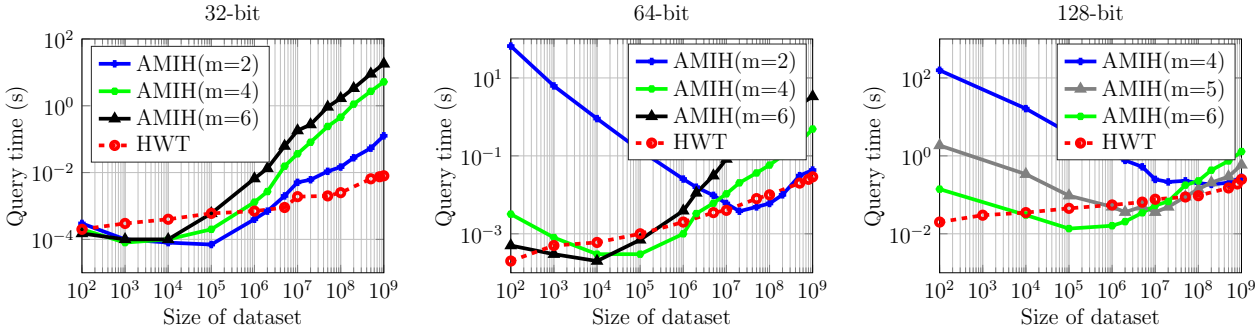


Figure 4.8: Average query time of HWT and AMIH for the task of angular nearest neighbor search.

the average query time (some values resulted in segmentation fault due to high memory overhead) but here we only report those that exhibit better performance than HWT for at least one of the dataset sizes. The figure shows that, for each value of m, there is often a range of dataset size in which MIH and AMIH outperforms HWT (often when m is close to $p/\log_2 n$). This trend can be seen more clearly in 64-bit and 128-bit codes, nevertheless outside of this range the HWT performs better. Moreover, for large number of codes HWT often exhibit superior performance. Due to the lack of space, the average performance of techniques over different sizes of dataset is omitted from this section, but it indicates that for all lengths of code the average query time of HWT (averaged over different sizes of dataset) is the smallest. In general, MIH and AMIH in their optimal parameter setting have a marginally better performance than HWT but when the number of hash table deviates from its optimal value, HWT outperforms MIH. Therefore, when all binary codes are available at one time, using HWT is not the best choice as compared with the MIH.

HWT vs tree search algorithms

Next, we compare the performance of HWT with some of the well-known branch and bound nearest neighbor search techniques. For the experiments of this section we use the *ann-benchmark* [6], a tool for evaluating the performance of in-memory approximate nearest neighbor search. This tool provides a standard interface for measuring the performance and quality achieved by nearest neighbor algorithms on different standard datasets. To make our algorithm usable by ann-benchmark, we implemented a python wrapper for our code and added the parameters to be tested to the configuration file. We compare HWT with tree-based search algorithms of ann-benchmark that support Hamming distance, namely

Annoy, KD tree, Ball Tree and RP forest. Discussing the details of these techniques is beyond the scope of this study but we briefly introduce them here.

Annoy [15] decomposes the search space using multiple trees to achieve sublinear search time. Each intermediate node splits the space into two half-spaces by sampling two points from the subset and taking the hyperplane equidistant from them. Each leaf node stores a subset of data points that lie in the region of space defined by its ancestors. Given the indexed dataset, the search algorithm prunes the search by only considering the points in the subspace where the query falls. Annoy incorporates a forest of such trees to increase the chance of colliding query with the nearest neighbor in at least one tree.

KD tree [14] is one of the long-standing nearest neighbor search algorithms. Each intermediate node in KD tree selects one of the input dimensions as the discriminators to partition the space using axis-aligned hyper-planes. Therefore, each leaf node represents a subset of point lying in the hypercube described by its ancestors.

Ball tree [96], as the name suggests, partitions the input space into disjoint hyperspheres each represented with a center and a diameter. While hyperspheres may intersect, each point is assigned to one ball per level according to its distance from the centers.

RP forest [41] works by creating a set of binary random projection trees. In each tree, dataset points are recursively partitioned based on the cosine of the angle of the points and a randomly drawn hyper-plane where the median angle is used as the pivot point.

Each of the selected algorithms are called with a set of parameters which directly affect the query-time vs recall rate trade-off. We run all algorithms multiple times each with a different parameter setting using the same configurations suggested by the ann-benchmark. To have a fair comparison with our exact technique, among the different settings we report the results of the one with the highest recall rate, defined as ratio of the number of points in the retrieved items being the true nearest neighbors and the number K of the true nearest neighbors. The dataset used for the experiments is the 1 million 256-bit SIFT dataset provided along with the ann-benchmark.

Table 4.2 compares the performance of HWT against the four selected tree based algorithms. Results indicate that HWT significantly outperforms traditional tree indexing approaches KD tree and Ball tree. The proposed approach also achieves a modest speed-up of 2× speed-up compared to the state-of-the-art Annoy with less memory overhead. For the angular distance, among the four selected techniques only Annoy supports angular nearest neighbor search which achieves average query time of 0.091 seconds whereas HWT takes 0.064 seconds. It is worth mentioning that the tree-based algorithms used in this section are more general techniques capable of working with several distance measures in addition to the Hamming distance. We believe that modifying them to better handle

Table 4.2: Performance of different tree based techniques applied to the 256-bit SIFT dataset with 1 million items to find the Hamming nearest neighbor.

Method	**Time** (ms)	**Memory** (KB)	**Recall**	**Indexing Time** (min)
Annoy	9.32	24169	99.23	32
Ball tree	197.32	1224	1	24
KD tree	181.30	1486	98.12	78
RP forest	34.12	9486	97.23	37
HWT	4.36	14896	1	24

binary data through bitwise manipulations can result in performance boost but doing so requires a considerable amount of human effort and is out of the scope of this research.

4.5 Summary

In this work, we focused on the K nearest neighbors search problem in binary datasets when both the query points and dataset items become available gradually. Based on the branch and bound paradigm, we proposed a tree data structure that solves the nearest neighbor problem for both Hamming and angular distances. The empirical results show that the proposed approach retrieves the nearest neighbor much faster than the linear scan and exhibit superior performance than MIH and AMIH for dynamic applications.

Chapter 5

Deep Spherical Quantization

Not surprisingly, with the dawn of deep learning, most of recent research effort in compact coding has been directed towards using deep networks for producing compact and functional binary codes. Deep hashing methods simultaneously learn the representation and hash coding from raw images. Similarly, deep MCQ has been the topic of study in recent years [21, 57]. Surprisingly enough, although MCQ is a more powerful model as it enables producing many more possible distinct distances, due to the lack of research, its performance in the context of deep supervised compact coding is inferior to state-of-the-art in supervised binary hashing [68]. Most of existing deep supervised MCQ techniques incorporate an unsupervised quantization (usually Product Quantization (PQ) [59]) on top of the features generated by a deep architecture. Nevertheless, the adopted networks often produce deep features with relatively high norm variance which adversely affects the quality of quantization [134].

5.1 Contributions

This chapter reformulates the quantization problem by L_2 normalizing deep features to remove norm variance. By exploiting the fact that resulting features lie on a hypersphere, we propose a novel MCQ algorithm, dubbed Deep Spherical Quantization (DSQ), that drops the hard orthogonality constraint of product quantization to achieve lower quantization error. Furthermore, to encourage better discriminating performance, inspired by the recently proposed center loss [133], we add a supervised quantization loss term to the final objective function to increase inter-class variance. Finally, we propose a sparse extension of our

quantization algorithm which is necessary for dealing with large codebooks [142]. Comprehensive empirical studies on three standard image retrieval benchmarks testify that DSQ generates compact binary codes which outperform many state-of-the-art methods. Our goal in this chapter is to answer the following research questions:

> RQ 5.1: How can we eliminate the negative effect of norm variance on the supervised MCQ?
>
> RQ 5.2: How can we achieve supervised MCQ formulation that is easier to optimize and also support sparse quantization?

These research questions are related to the high-level research question RQ2 discussed in Chapter 1.

5.2 Deep Spherical Quantization

In similarity retrieval, we are given a training set of n points, $\mathcal{X} = \{\mathbf{x}_i \in \mathbb{R}^d\}_{i=1}^n$, with each point associated with a class label, $y_i \in \{1, \ldots, l\}$. The goal, given query point $\mathbf{q} \in \mathbb{R}^d$, entails (approximately) finding items in $\mathcal{X}$ that are semantically closest to $\mathbf{q}$ so that the found neighbors share the same class label as $\mathbf{q}$. This study follows the idea of compact coding techniques that is converting database vectors into compact code and then performing the similarity search in the resulting space which has the advantage of lower memory cost and fast distance computation.

In this work, we propose to use a deep network that maps the input points into a discriminative space, and simultaneously perform a form of a supervised MCQ on the embedded points to achieve fast retrieval with low computational and storage overhead. To this aim, we define a loss function comprising four terms, softmax loss, center loss, quantization loss, and discriminative loss each of which will be discussed in the following.

5.2.1 Softmax and Center loss

In deep retrieval systems, obtaining a robust and discriminative representation is crucial for achieving good performance. Usually, this is achieved by applying the softmax loss to the representation layer of the network. However, the resulting features optimized with the supervision of softmax loss are often not discriminative enough as the softmax loss only focuses on finding a decision boundary that separates different classes without

considering the intra-class compactness which is crucial to the accuracy of nearest neighbor search [51, 133].

To increase the intra-class variations while keeping the features of different classes separable, we adopt the state-of-the-art *center loss* [133] on top of the softmax loss.

Let $f(\cdot;\theta) : \mathbb{R}^d \rightarrow \mathbb{R}^p$, with $p \ll d$, denote the feed-forward network that embed the input vectors into p-dimensional deep features, also let $\mathbf{z}_i$ denote deep feature representation of input $\mathbf{x}_i$, $\mathbf{z}_i = f(\mathbf{x}_i;\theta)$, then, the center loss is defined as:

$$L_C = \sum_{i=1}^{n} \|\mathbf{z}_i - \phi_{y_i}\|_2^2 \tag{5.1}$$

where y_i is the classes label associated with $\mathbf{z}_i$ and ϕ_{y_i} denotes the y_i-th class center of deep features. Intuitively, center loss learns a center for the features of each class and meanwhile aims at pulling the deep features of the same class close to its corresponding center. It has been shown that joint supervision of softmax loss and center loss can produce significantly better discriminative deep features [133].

5.2.2 Quantization Loss

To address RQ 5.1, we constrain the deep features to lie a p-dimensional unit hypersphere, *i.e.*, $\|f(\mathbf{x};\theta)\|_2 = 1$. Other than decreasing the intra-class variability of deep features [126], there are two advantages in normalizing feature vectors: 1) norm variance is strictly zero, and 2) Euclidean nearest neighbor search is equivalent to Maximum Inner Product Search (MIPS) as for unit norm vectors we have $\|\mathbf{q} - \mathbf{x}\|_2^2 = 2 - 2\mathbf{q}^T\mathbf{x}$.

The main benefit of dealing with MIPS is that, unlike Euclidean distance (see (2.11)), inner product naturally satisfies the distributive law, that is $\langle \mathbf{q}, \sum_j \mathbf{t}_j \rangle = \sum_j \langle \mathbf{q}, \mathbf{t}_j \rangle$. MCQ works well in large part due to the fact that it permits the distance between query and a quantized point to be computed as the summation of partial distances between query and selected codewords. Given the query, the distances between query and all codewords are stored in query-specific lookup tables and then used to calculate the distance between query and all quantized points. However, to make Euclidean distance satisfy the distributive law, we either need to enforce strong [59, 92]/weak [141, 142] orthogonality constraints over the codewords of different dictionaries which reduces the fidelity of model and often leads to non-convex optimization, or we have to store the inner product between the all codewords in lookup table [7, 78] which increases storage cost and distance computation time.

To reduce the approximation error of MIPS, we need to minimize the *distance reconstruction error* of MCQ. Since the Euclidean distance on the unit sphere is equal to the negative dot product plus a constant, distance reconstruction error can be rewritten as:

$$\begin{aligned} &\mathbb{E}_{\mathbf{q}\sim P(\mathbf{q})}\Big[\sum_{i=1}^{n}|\langle \mathbf{z}_q, \mathbf{z}_i\rangle - \langle \mathbf{z}_q, \bar{\mathbf{z}}_i\rangle|\Big] = \\ &\mathbb{E}_{\mathbf{q}\sim P(\mathbf{q})}\Big[\sum_{i=1}^{n}\langle \mathbf{z}_q, \mathbf{z}_i - \bar{\mathbf{z}}_i\rangle\Big] \leq \\ &\sum_{i=1}^{n}\|\mathbf{z}_i - \bar{\mathbf{z}}_i\|_2 \end{aligned} \tag{5.2}$$

where $\bar{\mathbf{z}}_i$ denotes the approximation of $\mathbf{z}_i$ using MCQ and $\mathbf{z}_q = f(\mathbf{q};\theta)$.

This suggests that the search accuracy directly depends on the quantization error; low quantization error leads to high search accuracy.

Therefore, the cost function we aim to optimize is the quantization loss:

$$\begin{aligned} L_Q(\{C_j\},\{\mathbf{b}_i\}) = \quad & \sum_{i=1}^{n}\|\mathbf{z}_i - [C_1,\ldots,C_m]\mathbf{b}_i\|_2^2 \\ & \mathbf{b}_i = [\mathbf{b}_{i1}^T,\ldots,\mathbf{b}_{im}^T]^T \\ & \mathbf{b}_{ij}\in\{0,1\}^h, \|\mathbf{b}_{ij}\|_1 = 1 \\ & j = 1,\ldots m \end{aligned} \tag{5.3}$$

The benefit of such a simple formulation, in comparison to those that enforce multiple constraints on the codewords [59, 92, 141] are multi-fold; it causes a straightforward optimization procedure and also less implementation overhead.

5.2.3 Discriminative Dictionary Learning

Finally, we also incorporate the supervisory information during quantization procedure. In particular, we encourage the quantized points to be closer to their centers. To achieve this goal, we use the following loss:

$$L_D = \sum_{i=1}^{n}\|\phi_{y_i} - C\mathbf{b}_i\|_2^2 \tag{5.4}$$

Intuitively, (5.4) penalizes the cases where the point $\bar{\mathbf{z}}_i$ is not assigned to the clusters that are close to ϕ_{y_i}.

The overall loss for training model takes the form:

$$L = L_{softmax} + \alpha L_Q + \lambda L_C + \gamma L_D \tag{5.5}$$

where α, λ and γ are the hyper-parameters that control the effect of each term.

5.2.4 Optimization

The objective function composes of four sets of learnable parameters, the parameters of the deep network θ, the centers ϕ_{y_i}s, the codewords in matrix C, the codeword assignment matrix B. We use alternative optimization to solve the problem with each iteration updating one set of parameters while fixing others.

Updating θ. With C, ϕ_{y_i}s, and B fixed, the parameters of the network are updated through back-propagation as all of the terms in the loss are differentiable.

Updating Φ. We follow a similar procedure to [133] for updating the centers. In particular, to avoid large perturbation caused by few mislabelled instances, we use a learning rate parameter ζ for training the centers:

$$\phi_{y_i}^{t+1} = \phi_{y_i}^{t} - \zeta \Delta \phi_{y_j} \tag{5.6}$$

$$\Delta \phi_{y_j} = \frac{\sum_{i=1}^{n} \mathbb{1}(y_i = j) \cdot [\lambda(\phi_{y_i} - \mathbf{z}_i) + \gamma(\phi_{y_i} - C\mathbf{b}_i)]}{1 + \sum_{i=1}^{n} \mathbb{1}(y_i = j)} \tag{5.7}$$

where $\mathbb{1}(condition)$ equals 1 if the condition is satisfied and 0 otherwise. Ideally, the centers should be updated in each iteration based on the whole training set which would be extremely costly. To reduce the cost, the update is performed on the mini-batches.

Updating C. Given B, ϕ_{y_i}s and θ fixed, the resulting optimization problem is:

$$\alpha \|Z - CB\|_2^2 + \gamma \|\Phi - CB\|_2^2 \tag{5.8}$$

where $Z = [\mathbf{z}_1, \ldots, \mathbf{z}_n]$, $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$, and $\Phi = [\phi_{y_1}, \ldots, \phi_{y_n}]$. This is a quadratic function in C and therefore a closed-form solution exists:

$$C = \frac{1}{\alpha + \gamma}(\alpha Z + \gamma \Phi) B^T (BB^T)^{-1} \tag{5.9}$$

It is easy to observe that the optimization problem decomposes over each of the p dimensions. Thus, we can reduce the computational cost by solving p least square problem each with mh variables.

$$\begin{aligned} \min_{C^{(t)}} \alpha \|Z^{(t)} - C^{(t)} B^{(t)}\|_2^2 + \gamma \|\Phi^{(t)} - C^{(t)} B^{(t)}\|_2^2 \\ \forall \quad t = 1, \ldots, p \end{aligned} \tag{5.10}$$

Each of the p problems is a least squares problem with a closed form solution. Online learning algorithms can also be leveraged for acceleration [76].

Updating B. Given θ, ϕ_{y_i} and C fixed, optimizing binary matrix B, known as encoding phase, has been historically identified as the bottleneck of MCQ [7, 78].

It can be seen that the composition indicator vector $\mathbf{b}_i$ is independent of all other vectors $\{\mathbf{b}_t\}_{t \neq i}$. Thus, the optimization problem with respect to B can be decomposed into n independent subproblems:

$$\begin{aligned} \min_{\mathbf{b}_i} \quad & \alpha \|\mathbf{z}_i - C\mathbf{b}_i\|_2^2 + \gamma \|\phi_{y_i} - C\mathbf{b}_i\|_2^2 \\ & \mathbf{b}_i = [\mathbf{b}_{i1}^T, \ldots, \mathbf{b}_{im}^T]^T \\ & \mathbf{b}_{ij} \in \{0,1\}^h, \|\mathbf{b}_{ij}\|_1 = 1 \\ & i = 1, \ldots, n \quad j = 1, \ldots, m \end{aligned} \tag{5.11}$$

The problem is essentially a high-order Markov Random Field (MRF) problem which is NP-hard. Following [78], we use Stochastic Local Search (SLS) method to optimize $\mathbf{b}_i$. The idea of SLS for escaping local minima is to iteratively alternate between a local search procedure, and a randomized pertubation to the current solution. For the local search, we again use alternative optimization technique. Given $\{\mathbf{b}_{ij}\}_{j \neq t}$ fixed, $\mathbf{b}_{it}$ is updated by exhaustively checking all codewords of C_j and finding the element that minimizes the objective function in (6.9). For the perturbation procedure of SLS, we randomly choose k codes by sampling from the uniform distribution $\mathcal{U}(1, m)$. The selected codes are perturbed by setting each of them to a uniformly selected random value between 1 and h. The resulting perturbed solution is then accepted as the starting point of the next local search procedure. Although this procedure is computationally demanding, it can be accelerated using GPU implementation [79, 80], making encoding even faster than codebook learning.

5.2.5 Asymmetric Distance Computation

Given the query, the search process starts by embedding the query using the trained network, $\mathbf{z}_q = f(\mathbf{q}; \theta)$. Then, the inner product between $\mathbf{z}_q$ and all codewords are stored in

$m \times h$ query-specific lookup table. Finally, inner product between query and all database vector is approximated with:

$$\langle \mathbf{z}_q, \mathbf{z}_i \rangle \approx \sum_{j=1}^{m} \langle \mathbf{z}_q, C_j \mathbf{b}_{ij} \rangle \tag{5.12}$$

Therefore, computing the inner product between query and each database item takes $O(m)$ lookups and $O(m)$ addition operations (same as PQ), plus the time required to embed the query into the deep feature space.

5.2.6 Sparse Codebook Learning

In sparse codebook learning, the optimization problem is augmented with sparsity constraint on the codewords. The key advantage of sparse coodebooks is that the distance between the query and every codeword can be computed efficiently using sparse vector manipulations. This is practically important as for large codebooks, with many codewords, the time required for online construction of lookup tables become non-negligible. Zhang *et al.* [142] have shown that sparse codewords can increase the search speed up to 30%. As the name suggests, the Sparse Composite Quantization (SCQ) technique proposed in [142] adds sparsity constraint to the CQ [141] formulation and uses coordinate descent to solve the optimization problem. However, CQ itself involves a hard optimization problem and adding the sparsity constraint makes the problem even harder.

In contrast, DSQ formulation reduces codebook optimization to a linear regression problem, thus adding the sparsity constraint changes the objective to a regularized quadratic problem. In particular, using straightforward algebraic manipulations (5.8) can be rewritten as:

$$(\alpha + \gamma) \| \frac{\alpha Z + \gamma \Phi}{\alpha + \gamma} - CB \|_2^2 - \frac{\| \alpha Z + \gamma \Phi \|_2^2}{\alpha + \gamma} + \alpha \| Z \|_2^2 + \gamma \| \Phi \|_2^2 \tag{5.13}$$

Since only the first term depends on C, we can write the objective function of sparse quantization as:

$$\min_{C} \| \frac{\alpha Z + \gamma \Phi}{\alpha + \gamma} - CB \|_2^2 \quad s.t. \quad \| C \|_0 \leq \epsilon \tag{5.14}$$

The resulting optimization is non-convex because of L_0 regularization term. Commonly, such problems are relaxed by replacing L_0 norm with convex L_1 norm. Therefore, our final

objective function for learning sparse codebooks is defined as:

$$\min_{C} \| \frac{\alpha Z^T + \gamma \Phi^T}{\alpha + \gamma} - B^T C^T \|_2^2 \quad s.t. \quad \|C\|_1 \leq \epsilon \tag{5.15}$$

which is essentially a linear regression problem with L_1 norm regularization on the coefficients, known as Lasso in the statistical literature. It can be efficiently solved using a wide range of heavily-optimized off-the-shelf Lasso solvers such SPGL1 solver [125]. This answers RQ 5.2.

5.3 Experiments

In this section, we gauge the performance of the proposed supervised quantization approach by comparing it with the state-of-the-art against three different datasets.

5.3.1 Datasets and Evaluation

We conduct experiments on three standard datasets: CIFAR-10 [65], NUS-WIDE [28] and ImageNet [33].

CIFAR-10 dataset consists of 60,000 32×32 color images evenly divided into 10 categories. We follow the official split of the datasets and use 50K images as the training set and 10k images as the query set.

NUS-WIDE is a set of 269,648 images collected from Flickr. This is a multi-label dataset where each image is associated with one or multiple labels from a given 81 concepts. Following [112, 129], we collect 193,752 images that are from the 21 most frequent labels for evaluation, including *sky, clouds, person, water, animal, grass, building, window, plants, lake, ocean, road, flowers, sunset, relocation, rocks, vehicles, snow, tree, beach, and mountain*. For each label, we randomly sample 100 images as query points and the remaining images form the training set.

The dataset ILSVRC 2012, named as ImageNet here, contains over 1.2 million images covering 1,000 categories. Following the settings in [22, 27], we select 100 categories and use images associated with them in the provided training set and the validation set as the training and the query sets, respectively.

Parameter setting. There are trade-off parameters in the objective function (5.5): α for quantization loss, λ for center loss and γ for discriminative loss. We select parameters

via validation. In particular, we choose a subset of the training set (same size as the query set), and the best parameters are chosen so that the average performance in terms of MAP is maximized against the validation set. We fix ζ to 0.5 and k to 4.

Following almost all MCQ techniques [7, 92, 142], we choose $k = 256$ to be the codebook size, so that each subindex fits into one byte of memory. This let us store B as a $m \times n$ `uint8` matrix. We vary $m = \{2, 4, 6, 8\}$ such that $m \log_2 k$ is equal to the desired bit-rates which are $\{16, 32, 48, 64\}$.

Experimental settings. Raw images are used as the input for all deep methods, but the images are resized to fit the input of the adopted model. For fairness of comparison, for all deep compact coding methods here, Alexnet is adopted as the core architecture. To reduce the size of deep features, a fully connected layer is added to the network which transforms the output of the network into a 256-dimensional feature space, thus $p = 256$. The size of feature space is not tuned for saving time while we think that tuning it might yield better performance. The L_2 normalization is performed on the 256-dimensional deep features using a L_2 normalize layer [103].

We fine-tune layers conv1fc7 copied from the AlexNet model pre-trained on ImageNet and train the last layer which maps the feature layer via back-propagation. As the last layer is trained from scratch, we set its learning rate to be 10 times that of the other layers. We use mini-batch stochastic gradient descent (SGD) with 0.9 momentum as the solver, and cross-validate the learning rate from 10^{-5} to 10^{-2} with a multiplicative step-size $\sqrt{10}$. We also fix the mini-batch size of images as 128 and the weight decay parameter as 0.0005. Following [78], we use SPGL1 as the lasso solver for the sparse extension of our algorithm [125]. For non-deep methods, we extract the outputs of the layer fc7 in the deep model [34] as input features.

Methods. DSQ is compared with a wide range of supervised compact coding methods including binary hashing methods: KSH [74], ITQ [47], SDH [112], CNNH [135], DPSH [69], DSH [72], HashNet [22], and supervised quantization techniques: SQ [129], SUBIC [57], DQN [21] and DTQ [71]. We implemented SQ in Python as its source code is not available at the time of writing this chapter. We tried our best to be faithful to the experimental settings of the paper [129]. Other techniques are executed using the implementation generously provided by the authors.

5.3.2 Results

Single domain retrieval. Single-domain retrieval is the main experimental benchmark in the supervised binary hashing literature in which the query and training items belong

	CIFAR-10				NUS-WIDE				ImageNet			
Method	16	32	48	64	16	32	48	64	16	32	48	64
KSQ	0.3216	0.3285	0.3371	0.3384	0.4061	0.4182	0.4264	0.4436	0.1620	0.2818	0.3422	0.3934
ITQ	0.2412	0.2432	0.2482	0.2531	0.5573	0.5932	0.6128	0.6166	0.3115	0.4632	0.5223	0.5446
SDH	0.4199	0.4301	0.4392	0.4465	0.5342	0.6282	0.6298	0.6335	0.2729	0.4521	0.5329	0.5893
CNNH	0.5373	0.5421	0.5765	0.5780	0.6221	0.6233	0.6321	0.6372	0.2888	0.4472	0.5328	0.5436
DPSH	0.6367	0.6412	0.6573	0.6676	0.7015	0.7126	0.7418	0.7423	0.3226	0.5436	0.6217	0.6534
DSH	0.6192	0.6565	0.6624	0.6713	0.7181	0.7221	0.7521	0.7531	0.3428	0.5500	0.6329	0.6645
HashNet	0.6857	0.6923	0.7183	0.7187	0.7331	0.7551	0.7622	0.7762	0.5016	0.6219	0.6613	0.6824
DTQ	0.7037	0.7191	0.7319	0.7373	0.7511	0.7812	0.7886	0.7892	0.5128	0.6123	0.6727	0.6916
SUBIC	0.6555	0.6789	0.6854	0.7014	0.7021	0.7131	0.7555	0.7568	0.5547	0.5597	0.6462	0.6622
SQ	0.6212	0.6438	0.6545	0.6578	0.7126	0.7138	0.7303	0.7423	0.3865	0.5586	0.6279	0.6618
DQN	0.5979	0.6097	0.6099	0.6133	0.6913	0.7121	0.7471	0.7562	0.5065	0.6205	0.6669	0.6912
DSQ	**0.7212**	**0.7346**	**0.7418**	**0.7589**	**0.7785**	**0.7899**	**0.7918**	**0.7988**	**0.5769**	**0.6541**	**0.6800**	**0.6940**

Table 5.1: Single-domain category retrieval performance of DSQ versus the state-of-the-art with 16, 32, 48 and 64 bit codes.

to the same set of class labels. To evaluate performance of different techniques, we adopt the widely used Mean Average Precision (MAP). We report the results of MAP@5000 and MAP@1000 for NUS-WIDE and ImageNet datasets respectively. Table 5.1 shows the single-domain retrieval performance of DSQ against a wide-range of techniques. The observation is that our proposed method consistently delivers the best performance for different length of codes. We attribute the performance improvement to the proposed loss that aims at jointly preserving similarity information and controlling the quantization error. Also, dropping the orthogonality constraint increases the fidelity of codebooks which in turn reduces the approximation error of nearest neighbor search. Finally, back-proping the proposed supervised quantization loss can remarkably enhance the quantizibilty of the deep representation.

Figure 5.1 also shows the performance of different techniques in terms of the precision-recall curves for 64-bit codes. From the curves, we can observe that DSQ delivers higher precision than the state-of-the-art compact coding methods at the same recall rate. This shows that DSQ is also favourable for precision-oriented retrieval systems. Although the query time comparison is not presented here due to space limit, we observed that all deep MCQ techniques in this study exhibit similar query time mainly because they adopt the same core architecture (AlexNet). However, binary hashing techniques are often faster than deep MCQ as they incorporate Hamming distance to compare binary codes.

Sparse coding. We also show the performance of sparse extension of DSQ. To the best of our knowledge, sparse DSQ is the first attempt to explore supervised sparse multi-codebook quantization for semantic similarity search. Nevertheless, we compare our technique with two unsupervised sparse quantization techniques, SCQ [142] and SLSQ [78]

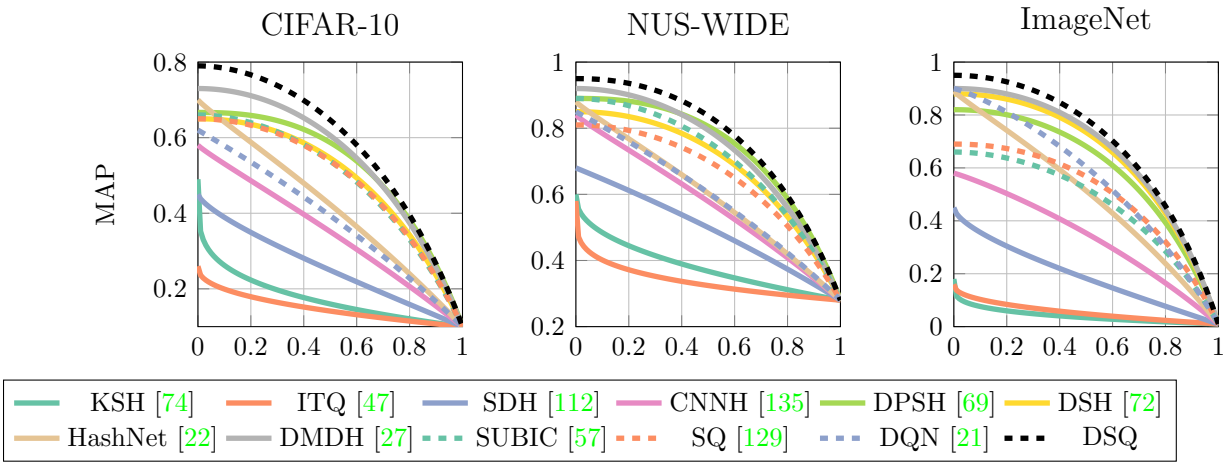


Figure 5.1: Precision-recall curves on the CIFAR-10, NUS-WIDE and ImageNet datasets for 64-bit codes.

applied to the deep features of the fc7 layer of the deep model in [34].

Following [141], we evaluate the sparse version of our algorithm, Sparse Deep Spherical Quantization (SDSQ) , using two degrees of sparsity: SDSQ1 with $\|C\|_0 \leq \epsilon = h \cdot p$ and SDSQ2 with $\|C\|_0 \leq \epsilon = h \cdot p + p^2$. Since the former criterion imposes a harder sparsity constraint on the codebooks, we would naturally expect to achieve lower search accuracy but better query time. We compare against SCQ1 and SCQ2 from [142] and SLSQ1 and SLSQ2 from [78].

Figure 5.2 shows the performance of different techniques against three different datasets. Again, in this scenario, we observe that SDSQ comfortably outperforms the baselines with a large margin mainly because sparse DSQ jointly optimizes the quantization error while preserving the semantic similarity and satisfying the sparsity constraint, whereas the other benchmarks separately apply unsupervised sparse quantization, which merely minimizes the quantization error.

Cross-domain retrieval. To further evaluate our supervised quantization method, we follow an alternative evaluation protocol from [107] wherein the model learned on a given set of training classes is tested on a new, disjoint set of test classes. This protocol is used to show how each method is capable of preserving the semantic information of certain classes implicitly even if the class samples are not included in the training set.

Toward this aim, we partition the samples based on their class labels such that 70% of the labels belong to the training and the remaining labels are used to form the base and

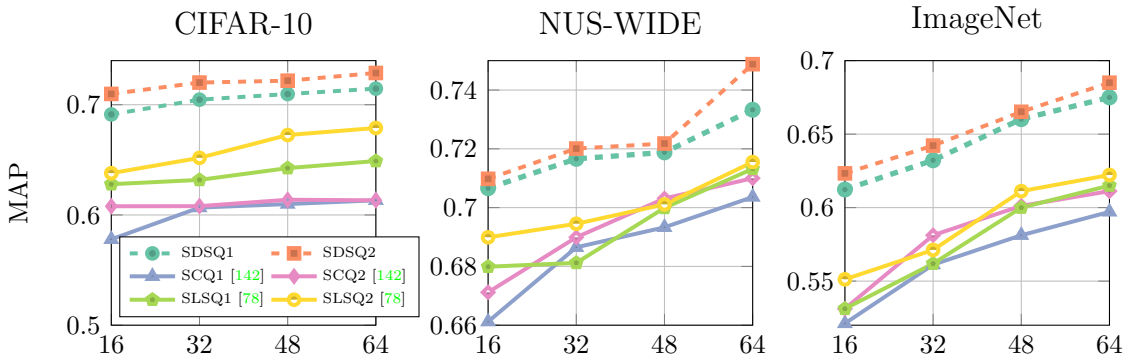


Figure 5.2: Mean Average Precision performance of different sparse quantization techniques against three datasets.

query set. Note that in this scenario the training set is used to optimize the parameters of the model. Once learning is completed, the training set is removed and the items of base set are mapped into compact codes using the trained model. Finally, the average performance over the query set is reported. We use 80% of the samples with unseen classes as the training set and the rest as the query set. This process is repeated 5 times with random class splits and the average results is reported. For this setting, during the encoding phase, we drop the L_C term from loss because the trained centers do not correspond to any of the labels in the base set. Similarly, the regression loss term in SQ [129] is dropped during encoding as it directly depends on the class labels of training set.

Table 5.2 demonstrates the results of this experiment which shows the superiority of DSQ for different lengths of code. We also observe that the MAP performance of methods are generally higher than that of the previous protocol since there is less variation in the base set consisting of only 3 classes and fewer samples to retrieve from. Also the rank of techniques is different from the single domain experiments. For example, SUBIC exhibits the closest performance to DSQ whereas in single-domain setting DTQ is the closest.

5.3.3 Ablation Study

We also perform an ablation study to showcase the contribution and importance of loss function components on the final performance of the model by empirically comparing different variants of DSQ. We evaluate this experiment across different models to understand the sensitivity of DSQ to different terms: 1) $L_{softmax} + L_Q$, 2) $L_{softmax} + L_Q + L_C$, 3)

Method	16	32	48	64
CNNH	0.6241	0.6456	0.6478	0.6491
DPSH	0.6894	0.7134	0.7198	0.7256
HashNet	0.7826	0.7941	0.7989	0.8010
SUBIC	0.7832	0.7931	0.8032	0.8077
DSH	0.7316	0.7388	0.7437	0.7456
SQ	0.7112	0.7126	0.7319	0.7389
DQN	0.7562	0.7612	0.7649	0.7655
DTQ	0.7525	0.7685	0.7700	0.7895
DSQ	**0.7944**	**0.8165**	**0.8195**	**0.8218**

Table 5.2: Mean Average Precision performance of different techniques for the task of cross domain performance on CIFAR-10.

$L_{softmax} + L_Q + L_D$, and 4) $L_C + L_D$. For each model, the coefficients of different terms are again tuned using cross validation and the average performance of model for 64-bit codes against CIFAR-10 dataset is reported in Figure 5.3.

The first observation is that all of the loss components contribute in improving MAP. Also, the plot indicates the importance of softmax loss. This is due to the fact that the softmax loss is the only term in the objective function that uses that class labels to force the deep features of different classes staying apart, without it, the resulting loss function degrades all inputs points to be projected onto a single point. The figure also demonstrates considerable contribution of discriminative loss, L_D, showing the effectiveness of our framework in incorporating semantic information during quantization.

5.4 Summary

In this study, we propose a deep supervised quantization technique for efficient and fast image retrieval. By incorporating L_2 normalized features, we propose a simple yet efficient multi-supervised MCQ algorithm for encoding large-scale datasets with similarity preserving binary codes. We also show that our algorithm can be easily extended to accommodate sparsity constraint in the codebooks which is necessary for learning large-scale codebooks.

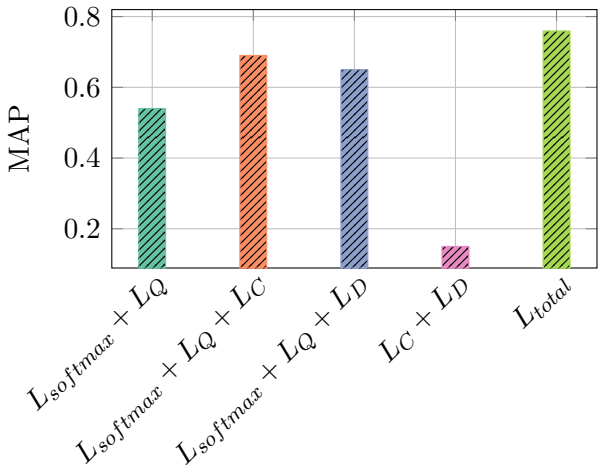


Figure 5.3: Difference in MAP, when different loss components are excluded from DSQ objective function. The experiments are conducted on 64-bit codes of CIFAR-10 dataset.

Comprehensive experiments justify that DSQ and its sparse extension generate compact binary codes that yield state-of-the-art retrieval performance on three standard benchmarks, CIFAR-10, NUS-WIDE, and ImageNet.

Chapter 6

Augmented Vector Quantization

6.1 Contributions

As discussed in 2.3.2, with higher number of parameters, non-orthogonal MCQ techniques tend to outperform their orthogonal counterparts, however, this comes at the cost of either a higher storage cost to encode the norms, or longer query time to calculate the codebook inner product term. In this chapter, we first put forward a non-orthogonal MCQ technique, which in contrast to its predecessors, does not require a separate norm quantizer and also benefits from efficient codebook learning as well as fast query time. This is achieved through applying two distinct vector transformations on database and query vectors which allows casting the Euclidean NNS to Maximum Inner Product Search (MIPS) – a problem that can be solved more efficiently with MCQ. We gauge the performance of our quantization approach on the task of nearest neighbor search against three large-scale datasets. The results indicate that the proposed approach can achieve better recall rates and faster optimization without increasing the memory cost.

Second, we provide a generalization bound for MCQ. As the number of potential centers grows exponentially with the number of codebooks, MCQ is at risk of over-fitting: the codebooks optimized on a training set may fail to achieve a low quantization error for unseen test points. We show, however, that the sample complexity grows only polynomially with the number of codebooks. To the best of our knowledge, this is the first distribution-independent sample complexity bound for MCQ techniques. The goal of this this chapter is to answer the following research questions:

RQ 6.1: How can we reformulate unsupervised MCQ in order to learn non-orthogonal

codebooks and without the need for a separate norm encoder?

RQ 6.2: What is the sample complexity of MCQ family?

6.2 Augmented Vector Quantization

To answer RQ 6.1, this section details our non-orthogonal MCQ technique, dubbed Augmented Vector Quantization (AVQ).

To attain our objective function, we first introduce a pair of vector transformations 1) $f : \mathbb{R}^d \to \mathbb{R}^{d+1}$ for transforming database vectors, and 2) $g : \mathbb{R}^d \to \mathbb{R}^{d+1}$ for transforming queries, as:

$$f(\mathbf{x}) = [\mathbf{x}, \frac{\|\mathbf{x}\|_2^2}{2}] \quad \text{and} \quad g(\mathbf{q}) = [\mathbf{q}, -1]. \tag{6.1}$$

Now we have:

$$\|\mathbf{q} - \mathbf{x}\|_2^2 = \|\mathbf{q}\|_2^2 - 2\langle \mathbf{q}, \mathbf{x}\rangle + \|\mathbf{x}\|_2^2 = \|\mathbf{q}\|_2^2 - 2\langle g(\mathbf{q}), f(\mathbf{x})\rangle. \tag{6.2}$$

Considering the fact that $\|\mathbf{q}\|_2^2$ is constant for a fixed query, we obtain the following identity which establishes the connection between distance minimization and inner product maximization:

$$\arg\min_{\mathbf{x}\in X} \|\mathbf{q} - \mathbf{x}\|_2^2 = \arg\max_{\mathbf{x}\in X} \langle g(\mathbf{q}), f(\mathbf{x})\rangle. \tag{6.3}$$

This indicates that NNS in Euclidean space is equivalent to solving the Maximum Inner Product Search (MIPS) among the augmented vectors. Next, we show how we can use MCQ to solve MIPS.

6.2.1 MCQ for MIPS

To reduce the approximation error of MIPS using MCQ, we need to minimize the distance reconstruction error for the transformed query $\mathbf{y} = g(\mathbf{q})$ and the transformed database vector $\mathbf{z} = f(\mathbf{x})$:

$$\sum_{\mathbf{z}} |\langle \mathbf{y}, \mathbf{z}\rangle - \langle \mathbf{y}, \bar{\mathbf{z}}\rangle| = \sum_{\mathbf{z}} |\langle \mathbf{y}, \mathbf{z} - \bar{\mathbf{z}}\rangle| \leq \sum_{\mathbf{z}} \|\mathbf{y}\|_2 \|\mathbf{z} - \bar{\mathbf{z}}\|_2 = \|\mathbf{y}\|_2 \sum_{\mathbf{z}} \|\mathbf{z} - \bar{\mathbf{z}}\|_2. \tag{6.4}$$

The upper bound depends on the ℓ_2 norm of the query, but for a fixed query, the solution of MIPS is independent of the query norm. This shows that, similar to Euclidean

NNS, the search accuracy of MCQ for MIPS directly depends on the quantization loss; low quantization loss leads to high search accuracy.

Therefore, the cost function we aim to optimize is the quantization loss:

$$\begin{gathered}\min_{\{C_j\},\{\mathbf{b}_i\}} \sum_{i=1}^{n} \|\mathbf{z}_i - [C_1, \ldots, C_m]\mathbf{b}_i\|_2^2 \\ \mathbf{b}_i^T = [\mathbf{b}_{i1}^T, \ldots, \mathbf{b}_{im}^T], \mathbf{b}_{ij} \in \{0,1\}^k, \|\mathbf{b}_{ij}\|_1 = 1 \\ i = 1, \ldots, n \quad j = 1, \ldots m.\end{gathered} \tag{6.5}$$

It it easy to observe that our objective function is hyper-parameter-free with no codebook constraints, and as we will see in the following, does not require additional codebooks for encoding the cross codebook term.

6.2.2 Querying

Given $\mathbf{q}, C$ and $\mathbf{b}_i$'s, after applying the transformation $\mathbf{y} = g(\mathbf{q})$, the distance between query and a database vector is approximated with:

$$\langle \mathbf{y}, \mathbf{z}_i \rangle \approx \langle \mathbf{y}, \bar{\mathbf{z}}_i \rangle = \langle \mathbf{y}, \sum_{j=1}^{m} C_j \mathbf{b}_{ij} \rangle = \sum_{j=1}^{m} \langle \mathbf{y}, C_j \mathbf{b}_{ij} \rangle. \tag{6.6}$$

Note that inner product naturally satisfies the distributive law (unlike Euclidean distance), and as a result the cross codebook inner product term of (2.11) does not appear in inner product computation. This allows us to efficiently compute (6.6) using an $m \times k$ lookup table in $O(m)$ time. The only additional storage cost here is a single extra dimension of codebooks which is constant with respect to the number of database vectors.

6.2.3 Optimization

The quantization problem in (6.5) is a mixed-integer programming consisting of two groups of unknowns, codebooks $\{C_j\}_{j=1}^m$ and assignment vectors $\{\mathbf{b}_i\}_{i=1}^n$. We use an alternating optimization method, where each step updates one group of variables.

Updating codebooks, C. Let $C = [C_1, \ldots, C_m] \in \mathbb{R}^{(d+1)\times mk}$ denote the matrix of codebooks. Given B fixed, (6.5) is quadratic in C, thus a closed-form least squares solution exists:

$$C = ZB^T(BB^T)^{-1}, \tag{6.7}$$

where $Z^{(d+1)\times n} = [\mathbf{z}_1, \ldots, \mathbf{z}_n]$. It is easy to observe that the optimization problem decomposes over each of the $d+1$ dimensions. Thus, we can reduce the computational cost by solving $d+1$ least square problems each with mh variables.

$$\min_{C^{(t)}} \|Z^{(t)} - C^{(t)}B^{(t)}\|_2^2 \quad \forall t = 1, \ldots, d+1. \tag{6.8}$$

Each of the $d+1$ problems is a least squares problem with a closed form solution.

Updating assignment vectors, B. Given C fixed, optimizing the assignment matrix $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ is in general NP-hard [7, 78]. It can be seen that the assignment vector $\mathbf{b}_i$ is independent of all other vectors $\{\mathbf{b}_{t\neq i}\}$. Thus, the optimization problem with respect to B can be decomposed into n independent sub-problems:

$$\begin{aligned} &\min_{\mathbf{b}_i} \|\mathbf{z}_i - C\mathbf{b}_i\|_2^2 \\ &\mathbf{b}_i \in \{0,1\}^k, \ \|\mathbf{b}_i\|_1 = 1, \ i = 1, \ldots, n \end{aligned} \tag{6.9}$$

The problem amounts to a high-order Markov Random Field (MRF) inference problem which is NP-hard [29]. Following [78], we use Iterated Local Search (ILS) to optimize $\mathbf{b}_i$. The idea of ILS for escaping local minima is to iteratively alternate between a local search procedure, and a randomized perturbation to the current solution. For the local search, we again use an alternative optimization technique. Given $\{\mathbf{b}_{ij}\}_{j\neq t}$ fixed, $\mathbf{b}_{it}$ is updated by exhaustively checking all codewords in the C_j and finding the element that minimizes the objective function in (6.9). For the perturbation procedure of ILS, we randomly choose e (set to 4 in our experiments) segments of $\mathbf{b}_i$ by sampling from the uniform distribution $\mathcal{U}(1, m)$. The selected codes are perturbed by setting each of them to a uniformly selected random value between 1 and k. The resulting perturbed solution is then accepted as the starting point of the next local search procedure.

Every update step in the algorithm assures that the objective function value weakly decreases after each iteration.

6.2.4 Implementation Details

Instead of using (6.7) for updating the codebooks, we use *diagonal loading* which adds a constant λ to the diagonal elements of BB^T, thus we have:

$$C = ZB^T(BB^T + \lambda I)^{-1} \tag{6.10}$$

which is in fact the solution to the least squares problem with ℓ_2 regularization. Although we are not looking for regularized formulation in our problem, this will render the solution numerically stable as $BB^T + \lambda I$ is guaranteed to be full-rank even if BB^T is not. In addition, the solution (6.10) can be used both when B is tall and when B is wide. The matrix inversion can thus be performed directly with the help of Cholesky decomposition in $O(m^3h^3)$ time. By setting λ to a very small value ($\lambda = 10^{-5}$ in our experiments), one can minimize the effect of the regularization on the learned cobebooks.

6.3 A Generalization Bound for MCQ

One of the benefits of using MCQ compared to VQ is reducing the quantization error. In fact, with using m codebooks of size k, we can have k^m possible assignments; therefore, the effective number of codewords/centers can be exponential in m. This flexibility, however, comes with the risk of over-fitting. In other words, a codebook with low distortion on the training data may have high expected distortion on the test data (*e.g.*, on the base set). The following uniform convergence result shows that, fortunately, the sample complexity grows polynomially with m.

Theorem 6.1. *Let $\mathcal{D}$ be an arbitrary distribution supported on the unit ball of $\mathbb{R}^d$, and let $\{\mathbf{x}_i\}_{i=1}^n$ be an i.i.d. sample of size n generated from $\mathcal{D}$. Then with probability at least $1-\delta$ we have*

$$\forall C \in \mathcal{C}, \quad \left| \frac{1}{n} \sum_{i=1}^{n} \min_{\mathbf{b}_i} \|\mathbf{x}_i - C\mathbf{b}_i\|_2^2 - \mathbb{E}_{\mathbf{x}\sim\mathcal{D}} \left(\min_{\mathbf{b}} \|\mathbf{x} - C\mathbf{b}\|_2^2 \right) \right| \leq \frac{(m+1)^2}{2\sqrt{n}} \left(28k + (m+1)k\sqrt{\ln(16nm^2)} + \sqrt{2\ln(1/\delta)} \right)$$

where $\mathcal{C}$ corresponds to the set of possibilities for m codebooks each having k codewords, where each codeword is in the unit ball of $\mathbb{R}^d$.

As it can be seen, Theorem 6.1 establishes a distribution-free (worst-case) bound on the distortion, and holds uniformly for all C. Therefore, it works for both orthogonal and non-orthogonal choices of C. The proof is based on the work of Maurer and Pontil [84] which establishes a framework for bounding the Radermacher complexity of functions corresponding to coding schemes. A nice feature of this bound is that it is independent on d (as long as the distribution is supported on the unit ball of Hilbert space). The proof as well as more details can be found in Section B. This answers the RQ 6.2.

6.4 Experiments

6.4.1 Setup

Datasets. We demonstrate the performance of our technique on SIFT1M [59], ConvNet1M [80], and Deep1M [10] datasets which are often used in benchmarking the performance of nearest neighbor search. These datasets have three partitions: *train, base* and *query*. We follow the standard protocol and use the train set to learn the codebooks C. After learning is completed, we remove the training data and use the resulting codebooks to encode the base set and obtain B. Finally, we use the query set to find the approximate nearest neighbors in the compressed base set and report the average performance. SIFT1M, ConvNet1M, Deep1M contain 128, 128 and 96 dimensional vectors respectively. All datasets have 100K training points, 1M base points and 10K query points.

Baselines. We compare our approach, AVQ, with several state-of-the-art methods which include orthogonal techniques: PQ [59], OPQ [44, 92], as well as non-orthogonal techniques: CQ [141, 128], Residual Vector Quantization (RVQ) [26], and also enhanced version of LSQ++ (with SR_D perturbation method) [80]. For PQ and OPQ, we use the implementation provided by Norouzi *et al.* [92]. Likewise, for CQ we adopt the C++ implementation due to Zhang *et al.* [141]. Finally, for RVQ and LSQ++ (referred here as LSQ), we use Rayuela.jl which is a highly optimized MCQ library written in Julia with C++ and CUDA bindings to enhance encoding. We also implemented our technique on top of Rayuela.jl and adopted similar GPU enhancements to speed up manipulations. Following most of previous studies, we use $k = 256$ in all experiments so that each sub-index fits into one byte. This let us store B as a $m \times n$ `UINT8` matrix. All algorithms are executed for $m = \{8, 16\}$ codebooks which results in index lengths of $m \times \log k = \{64, 128\}$ bits. We use 20 optimization iterations for all techniques to train the codebooks. Other than that, we use the default hyper-parameters as provided in their respective code releases. Finally, following [80], the number of ILS iterations for the encoding step of LSQ and AVQ is set to 32.

Initialization. Like AQ, CQ, and LSQ, we use an auxiliary quantization method to initialize B and C in our optimization procedure. We run OPQ, followed by a simple method similar to optimized tree quantization [9] but simplified to assume that the dimension assignments are given by a natural partition of adjacent dimensions.

Evaluation metrics. We follow previous works and evaluate the performance of our technique in terms of Recall@R which is defined as percentage of queries for which the actual nearest neighbor lies among the R estimated nearest neighbors. The goal is to

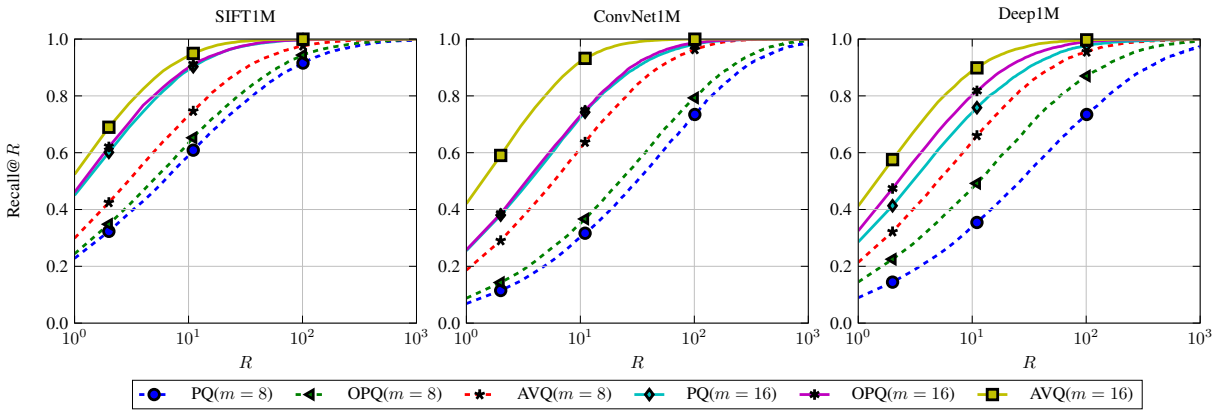


Figure 6.1: Recall@R curves for AVQ and orthogonal techniques.

obtain the highest recall for a given R. We also compare different techniques in terms of the optimization time. We do not provide the query time comparison here as for a fixed number of codebooks all selected techniques perform the same number of lookups and additions thus exhibit similar query time. All techniques are executed on a desktop with a 12-core i7-5930K CPU @ 3.50 GHz, 64 GB of RAM and a GeForce RTX 2080 Ti GPU.

6.4.2 Results

We report the Recall@R results for all three datasets in Figure 6.1 where it is immediately clear that AVQ outperforms orthogonal techniques for all values of R with a large margin. Tables 6.1 and 6.2 also show the comparison between AVQ and non-orthogonal techniques for two length codes, where AVQ consistently exhibits superior recall performance. Not surprisingly, some of the non-orthogonal techniques are closer competitors to AVQ as they learn full-dimensional codebooks with more trainable parameters. CQ, however, tends to perform even worse than orthogonal techniques. This is in line with other studies [80] which argue that often with CQ, the learned codebooks on the train set do not generalize well to the base set. Particularly, CQ performs well on the non-standard *train/query* protocol in which there are only two partitions of data; the codebook learning and encoding is run directly on the train set and the recall performance is also evaluated on the queries with respect to the train set [80].

Using the highly optimized CUDA bindings of Rayuela.jl library, our AVQ implementation manages to achieve similar optimization time as LSQ (the optimization time reported here is for SIFT1M and ConvNet 1M which have identical sizes and dimensions). Both

techniques are significantly faster than other benchmarks which are all executed on CPU and do not benefit from parallelism.

6.5 Comparisons

LSQ vs AVQ. There is a close connection between our closest competitor, LSQ, and our technique. LSQ is an extension of AQ [7] that adopts iterated local search to boost the performance of encoding step. LSQ and AQ expand the Euclidean distance between query and a compressed point as:

$$\|\mathbf{q} - \bar{\mathbf{x}}_i\|_2^2 = \|\mathbf{q}\|_2^2 - 2.\sum_{j=1}^{m}\langle \mathbf{q}, C_j\mathbf{b}_{ij}\rangle + \|\bar{\mathbf{x}}\|_2^2. \tag{6.11}$$

Similar to AVQ, both techniques use dot-product lookup tables to accelerate the computation of the second term of (6.11). However, to estimate the last term, AQ first proposes using a second $mh \times mh$ lookup table that stores the inner products between codebooks (note that we have, $\|\bar{\mathbf{x}}_i\|_2^2 = \|\sum_{j=1}^{m} C_j\mathbf{b}_{ij}\|_2^2 = \sum_{j=1}^{m}\sum_{t=1}^{m}\langle C_j\mathbf{b}_{ij}, C_t\mathbf{b}_{it}\rangle$). However, the authors noticed that this increases the distance computation time from $O(m)$ to $O(m^2)$ and also makes encoding harder. Alternatively, they suggest using a separate scalar quantizer to approximate the norms of the compressed vectors. This approach, also adopted by LSQ, increases the memory cost linearly with respect to the number of database vectors. Moreover, for a fixed length of code, the best split of memory budget between the two quantizers is not clear. In practice, LSQ assigns $m-1$ codebooks to quantize the database vectors and the remaining single codebook to quantize the norm. This *ad hoc* approach, however,

Table 6.1: Detailed recall@R rates and optimization time for $m = 8$ (64-bit codes).

	SIFT1M			ConvNet1M			Deep1M			
	@1	@10	@100	@1	@10	@100	@1	@10	@100	Time
CQ	17.18	59.12	90.12	12.32	52.11	91.19	15.49	51.10	88.41	∼40 mins
RVQ	22.38	61.29	92.40	14.09	53.39	92.73	16.43	54.81	89.65	∼3 mins
LSQ	29.79	72.54	96.27	18.60	62.46	96.17	20.15	62.33	94.89	∼1 min
AVQ	**31.98**	**74.25**	**98.69**	**19.62**	**63.74**	**97.55**	**21.29**	**64.12**	**95.47**	∼1 min

Table 6.2: Detailed recall@R rates and optimization time for $m = 16$ (128-bit codes).

	SIFT1M			ConvNet1M			Deep1M			
	@1	@10	@100	@1	@10	@100	@1	@10	@100	Time
CQ	34.25	81.25	92.58	30.25	83.25	98.72	30.63	79.14	92.18	∼70 mins
RVQ	42.95	89.32	99.74	31.48	84.14	99.62	36.14	86.09	99.53	∼6 mins
LSQ	51.33	94.49	99.95	41.44	93.21	99.97	41.18	88.86	99.72	∼2 mins
AVQ	**53.29**	**96.51**	**100**	**42.89**	**94.25**	**100**	**43.24**	**91.22**	**100**	∼2 mins

may not necessarily be the best split of the budget. On the other hand, AVQ treats the ℓ_2 norms as one of the input features and lets the optimization algorithm automatically decide on the split of quantization budget. Furthermore, unlike AQ and LSQ which adopt k-means to quantize the norms, AVQ leverages MCQ; allowing the number of potential centers for compressing the norm to grow exponentially in the number of parameters. The only additional cost of AVQ is a single extra dimension of the codebooks which is constant with respect to the number database vectors.

Asymmetric mapping. The general idea of using distinct mappings, one for database objects and the other for queries, has been investigated in other studies as well [48, 61, 113]. For example, Shrivastava and Li [116] argued that there is no symmetric LSH family for solving MIPS and proposed asymmetric mappings to cast inner product search into Euclidean NNS (reverse of what AVQ does) for which there exist efficient LSH functions. Neyshabour *et al.* [88] also studied the effectiveness of asymmetric mappings in the context of binary hashing. They showed that even if the target similarity function is symmetric, using asymmetric binary hashes can be more powerful and allow better approximation of the target similarity with shorter codes.

6.6 Summary

This chapter introduces a compact coding approach, Augmented Vector Quantization, to perform fast nearest neighbor search among high-dimensional items. The new formulation allows unconstrained codebooks learning without increasing the length of compressed codes. The first distribution and dimension independent generalization bound for the family of MCQ techniques is also introduced in this chapter.

Chapter 7

Conclusion and Future Directions

Here we recap the contributions and discuss possible directions of research to advance this work.

7.1 Contributions

This dissertation develops several algorithms with different flavors for solving large-scale nearest neighbor search, all of which are centred around using CDR to avoid exhaustive search and accelerate distance computation. Here, we revisit the research questions introduced in the first chapter and provide short answers based on our findings.

7.1.1 RQ1: How can we preprocess the compact discrete codes to achieve fast nearest neighbor search?

To answer this question, we propose two non-exhaustive search algorithms for solving nearest neighbor search among large-scale binary datasets:

- *Angular Multi-Index Hashing* (AMIH), a method for building multiple hash tables to enable exact angular NNS among binary codes, is introduced in Chapter 3. The approach is based on pigeon-hole principle, is simple, and easy to implement. The theoretical analysis on uniformly distributed data shows sublinear search time. Also, the empirical results on non-uniform large-scale benchmarks indicate substantial speedup in comparison to linear scan and other standard techniques.

- *Hamming Weight Tree* (HWT), introduced in Chapter 4, is another technique for solving exact Hamming and angular nearest neighbor search. The proposed data structure is simple and intuitive yet enables fast nearest neighbor search as well as efficient insertion of new items which is necessary for dealing with dynamic datasets.

7.1.2 RQ2: How can we design efficient compact code that are more faithful to the given notion of similarity and are easy to optimize?

As answers to this question, the last two chapters propose supervised and unsupervised quantization-based techniques in order to achieve fast distance computation:

- Chapter 5 proposes *Deep Spherical Quantization* (DSQ), a deep supervised quantization technique for efficient and fast image retrieval. By incorporating ℓ_2 normalized features, DSQ achieves a simple yet efficient supervised MCQ algorithm for encoding unit normalized data points with similarity preserving compact codes. Due to its simple formulation, it can be easily extended to accommodate sparsity constraint in the codebooks which is necessary for learning large-scale codebooks. Extensive empirical evaluation shows the superiority of the proposed techniques in comparison to the state-of-the-art in supervised hashing.

- Finally, Chapter 6 discusses *Augmented Vector Quantization* (AVQ), an unsupervised multi-codebook quantization technique for minimizing the approximation error of nearest neighbor search. By augmenting the database vectors with their ℓ_2 norms and query vectors with a constant, AVQ maps Euclidean nearest neighbor search into maximum inner product search which can be solved more efficiently with MCQ. The new formulation enables unconstrained codebooks learning without increasing the length of compressed codes. Along with that, the first distribution and dimension independent generalization bound for the family of MCQ techniques is provided.

The contributions of this thesis, linked with the publications, are summarized in Table 7.1.

7.2 Future Works

The proposed techniques laid the groundwork for several interesting future directions of research, some of which are proposed here:

	Binary hashing	Quantization
Learning	–	DSQ [39], AVQ
Search	AMIH [40], HWT [37, 38]	–

Table 7.1: Relationship between proposed techniques.

Adapting data to index. The space partitioning data structures developed in Chapter 3, Chapter 4, and the multi-index hashing based proposed by Norouzi *et al.* [95] all can provably support sublinear search time for uniformly distributed datapoints but it may not be true for other distributions. This can be problematic if we want to responsibly utilize these algorithms in scenarios with serious impact, where it is important to delineate regimes with reliable answers. Very recently, Sablayrolles *et al.* [106] discussed the idea of adapting data to index. In a nutshell, they propose to learn a mapping such that the output follows a specific distribution under which the subsequent indexing algorithm performs better. Similar ideas can be used in binary hashing techniques to make sure that the output of binary hashing technique not only preserves the neighborhood but also favors a uniform distribution to enhance search among the binary codes.

Non-exhaustive search among quantized vectors. The MCQ framework, such as techniques proposed in Chapters 5 and 6, in general provides better recall performance in comparison to the binary hashing techniques mainly because it allows many more possible distances. However, the problem of non-exhaustive search among the quantized points seems unsettled. Adopting a two-step quantization in which a coarse quantizer (such as k-means) partitions the database vectors and then encoding the residual error with a high resolution quantizer of MCQ family has been shown to be effective in avoiding retrieval. In this setting, the query is only compared with vectors that belong to the nearest bucket(s) of the coarse quantizer. However, such inverted indexing systems are unsupervised, and are built by a process of carefully designed manual parameter tuning. Further research in this direction can adopt simultaneous quantizer learning (for better recall) and partitioning of the space (for non-exhaustive search). The work by Jian *et al.* [58] takes the initial steps in this direction where the quantization error is back-propagated to the indexing structure, however, their study is basically based on binary hashing as they use Hamming distance to compute the distances.

Learning mixture of Gaussians with MCQ. There are strong connections between k-means and mixture of Gaussians. We also developed MCQ models that are in fact a generalization k-means. An interesting avenue for future research concerns formulations

of MCQ mixture models. Such models can make use of multiple codebooks, each of which is assigned to a subset of datapoints. At training and test time, one considers all of the codebooks and their codewords and picks the ones that minimize quantization error. The main research question is whether such mixture models will lead to a sufficient reduction in quantization error to justify the increase in encoding and storage cost.

References

[1] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

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

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. *Advances in Neural Information Processing Systems*, pages 1225–1233, 2015.

[4] Alexandr Andoni and Ilya Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. *Symposium on Theory of Computing*, pages 793–801, 2015.

[5] Alexandr Andoni and Ilya Razenshteyn. Tight lower bounds for data-dependent locality-sensitive hashing. *arXiv preprint arXiv:1507.04299*, 2015.

[6] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *International Conference on Similarity Search and Applications*, pages 34–49, 2017.

[7] Artem Babenko and Victor Lempitsky. Additive quantization for extreme vector compression. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 931–938, 2014.

[8] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260, 2014.

[9] Artem Babenko and Victor Lempitsky. Tree quantization for large-scale similarity search and classification. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4240–4248, 2015.

[10] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, 2016.

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

[12] Vassileios Balntas, Lilian Tang, and Krystian Mikolajczyk. Binary online learned descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(3):555–567, 2018.

[13] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *International Conference on World Wide Web*, pages 131–140. ACM, 2007.

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

[15] Erik Bernhardsson. Annoy https://github.com/spotify/annoy.

[16] Andrei Z Broder. On the resemblance and containment of documents. *Compression and Complexity of Sequences*, pages 21–29, 1997.

[17] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[18] Jeremy Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[19] Yue Cao, Bin Liu, Mingsheng Long, and Jianmin Wang. Hashgan: Deep learning to hash with pair conditional wasserstein gan. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1287–1296, 2018.

[20] Yue Cao, Mingsheng Long, Bin Liu, and Jianmin Wang. Deep cauchy hashing for hamming space retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1229–1237, 2018.

[21] Yue Cao, Mingsheng Long, Jianmin Wang, Han Zhu, and Qingfu Wen. Deep quantization network for efficient image retrieval. In *AAAI Conference on Artificial Intelligence*, 2016.

[22] Zhangjie Cao, Mingsheng Long, Jianmin Wang, and Philip S Yu. Hashnet: Deep learning to hash by continuation. In *IEEE International Conference on Computer Vision*, pages 5608–5617, 2017.

[23] Miguel A Carreira-Perpinán and Ramin Raziperchikolaei. Hashing with binary autoencoders. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 557–566, 2015.

[24] Amit Chakrabarti and Oded Regev. An optimal randomised cell probe lower bound for approximate nearest neighbour searching. In *IEEE Symposium on Foundations of Computer Science*, pages 473–482, 2004.

[25] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *ACM Symposium on Theory of Computing*, pages 380–388. ACM, 2002.

[26] Yongjian Chen, Tao Guan, and Cheng Wang. Approximate nearest neighbor search by residual vector quantization. *Sensors*, 10(12):11259–11273, 2010.

[27] Zhixiang Chen, Xin Yuan, Jiwen Lu, Qi Tian, and Jie Zhou. Deep hashing via discrepancy minimization. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 6838–6847, 2018.

[28] Tat-Seng Chua, Jinhui Tang, Richang Hong, Haojie Li, Zhiping Luo, and Yantao Zheng. Nus-wide: a real-world web image database from national university of singapore. In *International Conference on Image and Video Retrieval*, page 48. ACM, 2009.

[29] Gregory F Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2-3):393–405, 1990.

[30] Hui Cui, Lei Zhu, Jingjing Li, Yang Yang, and Liqiang Nie. Scalable deep hashing for large-scale social image retrieval. *IEEE Transactions on Image Processing*, 2019.

[31] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. *International Conference on World Wide Web*, pages 271–280, 2007.

[32] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Annual Symposium on Computational Geometry*, pages 253–262. ACM, 2004.

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

[34] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International Conference on Machine Learning*, pages 647–655, 2014.

[35] Wei Dong. Kgraph https://github.com/aaalgo/kgraph.

[36] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling LSH for performance tuning. In *Conference on Information and knowledge management*, pages 669–678. ACM, 2008.

[37] Sepehr Eghbali, Hassan Ashtiani, and Ladan Tahvildari. Online nearest neighbor search in binary space. In *International Conference on Data Mining*, pages 853–858, 2017.

[38] Sepehr Eghbali, Hassan Ashtiani, and Ladan Tahvildari. Online nearest neighbor search using hamming weight trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[39] Sepehr Eghbali and Ladan Tahvildari. Deep spherical quantization for image search. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 11690–11699, 2019.

[40] Sepehr Eghbali and Ladan Tahvildari. Fast cosine similarity search in binary space with angular multi-index hashing. *IEEE Transactions on Knowledge and Data Engineering*, 31(2):329–342, 2019.

[41] Lyst Engineering. rpforest https://github.com/lyst/rpforest.

[42] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer-Verlag Berlin Heidelberg, 2006.

[43] Jinyang Gao, H.V. Jagadish, Beng Chin Ooi, and Sheng Wang. Selective hashing: Closing the gap between radius search and k-nn search. In *International Conference on Knowledge Discovery and Data Mining*, pages 349–358, 2015.

[44] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, 2013.

[45] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Very Large Data Bases*, pages 518–529, 1999.

[46] Yunchao Gong, Sanjiv Kumar, Vishal Verma, and Svetlana Lazebnik. Angular quantization-based binary codes for fast similarity search. In *Advances in Neural Information Processing Systems*, pages 1196–1204, 2012.

[47] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(12):2916–2929, 2013.

[48] Albert Gordo, Florent Perronnin, Yunchao Gong, and Svetlana Lazebnik. Asymmetric distances for binary embeddings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(1):33–47, 2013.

[49] Dan Greene, Michal Parnas, and Frances Yao. Multi-index hashing for information retrieval. *Annual Symposium on Foundations of Computer Science*, pages 722–731, 1994.

[50] Junfeng He, Wei Liu, and Shih-Fu Chang. Scalable similarity search with optimized kernel hashing. In *International Conference on Knowledge Discovery and Data Mining*, pages 1129–1138. ACM, 2010.

[51] Xinwei He, Yang Zhou, Zhichao Zhou, Song Bai, and Xiang Bai. Triplet-center loss for multi-view 3d object retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1945–1954, 2018.

[52] Jae-Pil Heo, Youngwoon Lee, Junfeng He, Shih-Fu Chang, and Sung-Eui Yoon. Spherical hashing. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2957–2964, 2012.

[53] L. K. Huang, Q. Yang, and W. S. Zheng. Online hashing. *IEEE Transactions on Neural Networks and Learning Systems*, 2017.

[54] Long-Kai Huang, Qiang Yang, and Wei-Shi Zheng. Online hashing. *International Joint Conference on Artificial Intelligence*, pages 1442–1428, 2013.

[55] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[56] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[57] Himalaya Jain, Joaquin Zepeda, Patrick Pérez, and Rémi Gribonval. Subic: A supervised, structured binary code for image search. In *IEEE International Conference on Computer Vision*, pages 833–842, 2017.

[58] Himalaya Jain, Joaquin Zepeda, Patrick Pérez, and Rémi Gribonval. Learning a complete image indexing pipeline. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4933–4941, 2018.

[59] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[60] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *International Conference on Acoustics, Speech and Signal Processing*, pages 861–864, 2011.

[61] Qing-Yuan Jiang and Wu-Jun Li. Asymmetric deep supervised hashing. In *AAAI Conference on Artificial Intelligence*, 2018.

[62] Yushi Jing and Shumeet Baluja. Visualrank: Applying pagerank to large-scale image search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1877–1890, 2008.

[63] Benjamin Klein and Lior Wolf. End-to-end supervised product quantization for image search and retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[64] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowledge and Information Systems*, 12(1):25–53, 2007.

[65] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Technical report, 2009.

[66] Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *Advances in neural information processing systems*, pages 1042–1050, 2009.

[67] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5(Apr):361–397, 2004.

[68] Qi Li, Zhenan Sun, Ran He, and Tieniu Tan. Deep supervised discrete hashing. In *Advances in Neural Information Processing Systems*, pages 2482–2491, 2017.

[69] Wu-Jun Li, Sheng Wang, and Wang-Cheng Kang. Feature learning based deep supervised hashing with pairwise labels. In *International Joint Conference on Artificial Intelligence*, pages 1711–1717. AAAI Press, 2016.

[70] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. Deep learning of binary hash codes for fast image retrieval. In *IEEE conference on computer vision and pattern recognition workshops*, pages 27–35, 2015.

[71] Bin Liu, Yue Cao, Mingsheng Long, Jianmin Wang, and Jingdong Wang. Deep triplet quantization. In *ACM Multimedia Conference on Multimedia Conference*, pages 755–763. ACM, 2018.

[72] Haomiao Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. Deep supervised hashing for fast image retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2064–2072, 2016.

[73] Wei Liu, Cun Mu, Sanjiv Kumar, and Shih-Fu Chang. Discrete graph hashing. In *Advances in Neural Information Processing Systems*, pages 3419–3427, 2014.

[74] Wei Liu, Jun Wang, Rongrong Ji, Yu-Gang Jiang, and Shih-Fu Chang. Supervised hashing with kernels. In *Conference on Computer Vision and Pattern Recognition*, pages 2074–2081. IEEE, 2012.

[75] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. *International Conference on Very Large Data Bases*, pages 950–961, 2007.

[76] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online dictionary learning for sparse coding. In *International Conference on Machine Learning*, pages 689–696, 2009.

[77] Minqi Mao, Zhonglong Zheng, Zhongyu Chen, Huawen Liu, Xiaowei He, and Ronghua Ye. Two-dimensional pca hashing and its extension. *International Conference on Pattern Recognition*, pages 1624–1629, 2016.

[78] Julieta Martinez, Joris Clement, Holger H Hoos, and James J Little. Revisiting additive quantization. In *European Conference on Computer Vision*, pages 137–153. Springer, 2016.

[79] Julieta Martinez, Holger H Hoos, and James J Little. Solving multi-codebook quantization in the gpu. In *European Conference on Computer Vision*, pages 638–650. Springer, 2016.

[80] Julieta Martinez, Shobhit Zakhmi, Holger H Hoos, and James J Little. Lsq++: Lower running time and higher recall in multi-codebook quantization. In *European Conference on Computer Vision*, pages 491–506, 2018.

[81] Julieta Martinez-Covarrubias. *Algorithms for large-scale multi-codebook quantization*. PhD thesis, University of British Columbia, 2018.

[82] Yusuke Matsui, Toshihiko Yamasaki, and Kiyoharu Aizawa. Pqtable: Nonexhaustive fast search for product-quantized codes using hash tables. *IEEE Transactions on Multimedia*, 20(7):1809–1822, 2017.

[83] Yusuke Matusi, Toshihiko Yamasaki, and Kiyoharu Aziawa. Pqtable: Fast exact asymmetric distance neighbor search for product quantization using hash tables. In *International Conference on Computer Vision*, 2015.

[84] Andreas Maurer and Massimiliano Pontil. k-dimensional coding schemes in hilbert spaces. *IEEE Transactions on Information Theory*, 56(11):5839–5846, 2010.

[85] Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge*, 19(88):2, 1969.

[86] Stanislav Morozov and Artem Babenko. Unsupervised neural quantization for compressed-domain similarity search. *International Conference on Computer Vision*, 2019.

[87] Marius Muja and David G Lowe. Fast matching of binary features. In *IEEE Conference on Computer and Robot Vision*, pages 404–410, 2012.

[88] Behnam Neyshabur, Nati Srebro, Ruslan R Salakhutdinov, Yury Makarychev, and Payman Yadollahpour. The power of asymmetry in binary hashing. In *Advances in Neural Information Processing Systems*, pages 2823–2831, 2013.

[89] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[90] Mohammad Norouzi. *Compact Discrete Representations for Scalable Similarity Search*. PhD thesis, University of Toronto, 2016.

[91] Mohammad Norouzi and David J Fleet. Minimal loss hashing for compact binary codes. In *International Conference on Machine Learning*, pages 353–360, 2011.

[92] Mohammad Norouzi and David J Fleet. Cartesian k-means. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 3017–3024, 2013.

[93] Mohammad Norouzi, David J Fleet, and Ruslan R Salakhutdinov. Hamming distance metric learning. In *Advances in neural information processing systems*, pages 1061–1069, 2012.

[94] Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast search in hamming space with multi-index hashing. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 3108–3115, 2012.

[95] Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast exact search in hamming space with multi-index hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(6):1107–1119, 2014.

[96] Stephen M Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[97] Eng-Jon Ong and Miroslaw Bober. Improved hamming distance search using variable length substrings. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2000–2008, 2016.

[98] Ezgi Can Ozan, Serkan Kiranyaz, and Moncef Gabbouj. Competitive quantization for approximate nearest neighbor search. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2884–2894, 2016.

[99] Ryan ODonnell, Yi Wu, and Yuan Zhou. Optimal lower bounds for locality-sensitive hashing (except when q is tiny). *ACM Transactions on Computation Theory*, 6(1):5, 2014.

[100] Jia Pan and Dinesh Manocha. Bi-level locality sensitive hashing for k-nearest neighbor computation. In *IEEE International Conference on Data Engineering*, pages 378–389, 2012.

[101] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. *Annual ACM-SIAM symposium on Discrete algorithm*, pages 1186–1195, 2006.

[102] Loc Paulev, Herv Jgou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348 – 1358, 2010.

[103] Rajeev Ranjan, Carlos D Castillo, and Rama Chellappa. L2-constrained softmax loss for discriminative face verification. *arXiv preprint arXiv:1703.09507*, 2017.

[104] Ilya Razenshteyn. *High-dimensional similarity search and sketching: algorithms and hardness*. PhD thesis, Massachusetts Institute of Technology, 2017.

[105] Ilya Razenshteyn and Ludwig Schmidt. Fast lookups of cosine and other nearest neighbors https://github.com/FALCONN-LIB/FALCONN.

[106] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. Spreading vectors for similarity search. In *International Conference on Learning Representations*, pages 689–696, 2019.

[107] Alexandre Sablayrolles, Matthijs Douze, Nicolas Usunier, and Hervé Jégou. How should we evaluate supervised hashing? In *International Conference on Acoustics, Speech and Signal Processing*, pages 1732–1736. IEEE, 2017.

[108] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.

[109] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[110] Venu Satuluri and Srinivasan Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *Very Large Data Bases*, 5(5):430–441, 2012.

[111] Jian Shao, Fei Wu, Chuanfei Ouyang, and Xiao Zhang. Sparse spectral hashing. *Pattern recognition letters*, 33(3):271–277, 2012.

[112] Fumin Shen, Chunhua Shen, Wei Liu, and Heng Tao Shen. Supervised discrete hashing. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 37–45, 2015.

[113] Fumin Shen, Yang Yang, Li Liu, Wei Liu, Dacheng Tao, and Heng Tao Shen. Asymmetric binary coding for image search. *IEEE Transactions on Multimedia*, 19(9):2022–2032, 2017.

[114] Fumin Shen, Xiang Zhou, Yang Yang, Jingkuan Song, Heng Tao Shen, and Dacheng Tao. A fast optimization method for general binary code learning. *IEEE Transactions on Image Processing*, 25(12):5610–5621, 2016.

[115] Rajendra Shinde, Ashish Goel, Pankaj Gupta, and Debojyoti Dutta. Similarity search and locality sensitive hashing using ternary content addressable memories. In *International Conference on Management of data*, pages 375–386. ACM, 2010.

[116] Anshumali Shrivastava and Ping Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (mips). In *Advances in Neural Information Processing Systems*, pages 2321–2329, 2014.

[117] Anshumali Shrivastava and Ping Li. In defense of minhash over simhash. *International Conference on Artificial Intelligence and Statistics*, pages 886–894, 2014.

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

[119] Shupeng Su, Chao Zhang, Kai Han, and Yonghong Tian. Greedy hash: Towards fast optimization for accurate hash coding in cnn. In *Advances in Neural Information Processing Systems*, pages 798–807, 2018.

[120] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *International Conference on Computer Vision*, pages 843–852, 2017.

[121] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Ver Large Data Bases*, 6(14):1930–1941, 2013.

[122] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Conference on Computer Vision and Pattern Cecognition*, pages 1701–1708, 2014.

[123] Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008.

[124] Antonio Torralba, Robert Fergus, Yair Weiss, et al. Small codes and large image databases for recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, page 2, 2008.

[125] E. van den Berg and M. P. Friedlander. Probing the pareto frontier for basis pursuit solutions. *SIAM Journal on Scientific Computing*, 31(2):890–912, 2008.

[126] Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, and Wei Liu. Cosface: Large margin cosine loss for deep face recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition*, June 2018.

[127] J. Wang, T. Zhang, j. song, N. Sebe, and H. T. Shen. A survey on learning to hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):769–790, 2018.

[128] Jingdong Wang and Ting Zhang. Composite quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[129] Xiaojuan Wang, Ting Zhang, Guo-Jun Qi, Jinhui Tang, and Jingdong Wang. Supervised quantization for similarity search. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2018–2026, 2016.

[130] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

[131] Yair Weiss, Rob Fergus, and Antonio Torralba. Multidimensional spectral hashing. *Computer Vision*, pages 340–353, 2012.

[132] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In *Advances in neural information processing systems*, pages 1753–1760, 2009.

[133] Yandong Wen, Kaipeng Zhang, Zhifeng Li, and Yu Qiao. A discriminative feature learning approach for deep face recognition. In *European conference on computer vision*, pages 499–515. Springer, 2016.

[134] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu. Multiscale quantization for fast similarity search. In *Advances in Neural Information Processing Systems*, pages 5745–5755, 2017.

[135] Rongkai Xia, Yan Pan, Hanjiang Lai, Cong Liu, and Shuicheng Yan. Supervised hashing for image retrieval via image representation learning. In *AAAI Conference on Artificial Intelligence*, 2014.

[136] Erkun Yang, Tongliang Liu, Cheng Deng, Wei Liu, and Dacheng Tao. Distillhash: Unsupervised deep hashing by distilling data pairs. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2955, 2019.

[137] Huei-Fang Yang, Kevin Lin, and Chu-Song Chen. Supervised learning of semantics-preserving hashing via deep neural networks for large-scale image search. *arXiv preprint arXiv:1507.00101*, 2, 2015.

[138] Huei-Fang Yang, Cheng-Hao Tu, and Chu-Song Chen. Adaptive labeling for deep learning to hash. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 10–16, 2019.

[139] Chun-Nam John Yu and Thorsten Joachims. Learning structural svms with latent variables. *International Conference on Machine Learning*, pages 1169–1176, 2009.

[140] Lei Zhang, Yongdong Zhang, Jinhu Tang, Ke Lu, and Qi Tian. Binary code ranking with weighted hamming distance. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1593, 2013.

[141] Ting Zhang, Chao Du, and Jingdong Wang. Composite quantization for approximate nearest neighbor search. In *International Conference on Machine Learning*, volume 2, page 3, 2014.

[142] Ting Zhang, Guo-Jun Qi, Jinhui Tang, and Jingdong Wang. Sparse composite quantization. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 4548–4556, 2015.

[143] Wanqian Zhang, Dayan Wu, Bo Li, Xiaoyan Gu, Weiping Wang, and Dan Meng. Fast and multilevel semantic-preserving discrete hashing. 2019.

[144] Xiaoyang Zhang, Jianbin Qin, Wei Wang, Yifang Sun, and Jiaheng Lu. Hmsearch: An efficient hamming distance query processing algorithm. In *International Conference on Scientific and Statistical Database Management*, pages 567–574, 2013.

[145] Fang Zhao, Yongzhen Huang, Liang Wang, and Tieniu Tan. Deep semantic ranking based hashing for multi-label image retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1556–1564, 2015.

[146] Han Zhu, Mingsheng Long, Jianmin Wang, and Yue Cao. Deep hashing network for efficient similarity retrieval. In *AAAI Conference on Artificial Intelligence*, 2016.

[147] Bohan Zhuang, Guosheng Lin, Chunhua Shen, and Ian Reid. Fast training of triplet-based deep binary embedding networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 5955–5964, 2016.

[148] Hui Zou, Trevor Hastie, and Robert Tibshirani. Sparse principal component analysis. *Journal of Computational and Graphical Statistics*, 15(2):265–286, 2006.

APPENDICES

Appendix A

Proof of Theorem 4.1

We start the proof of Theorem 4.1 by defining the event of *collision* over d-patterns, and then provide an upper bound on the probability of such collision.

Definition A.1 (Collision)**.** *Let $X_n = \{\mathbf{x}^i\}_{i=1}^n$ be a set of p-dimensional binary vectors, i.e., $\mathbf{x}^i \in \{0,1\}^p$. We say that query $\mathbf{q} \in \{0,1\}^p$ d-collides with X_n if*

$$\exists i \in [n],\ s.t.\ Q_d(\mathbf{x}^i) = Q_d(\mathbf{q})$$

Lemma A.1. *Let $X_n = \{\mathbf{x}^i\}_{i=1}^n$ and $\mathbf{q}$ be $n+1$ iid random variables generated from the uniform distribution over the p-dimensional binary cube. Then, for every $0 < d \leq p$, the probability that $\mathbf{q}$ d-collides with X_n is at most $n(\frac{d}{p})^{\frac{d}{2}}$.*

Proof. Note that because $\mathbf{q}$ is uniformly generated, the distribution of its Hamming weight (i.e., distribution of $\|\mathbf{q}\|_H$) is binomial. Similarly, the distribution of the Hamming weight of the all substrings of $\mathbf{q}$ are binomial, i.e., $\|\mathbf{q}_d\|_H \sim Bin(\frac{p}{d}, 1/2)$. Now with an application of union bound, and the fact that the d different patterns are generated independently we

have:

$$
\begin{aligned}
&Pr\left[\exists i \in [n] \text{ s.t. } Q_d(\mathbf{x}^i) = Q_d(\mathbf{q})\right] \leq n.Pr\left[Q_d(\mathbf{x}^1) = Q_d(\mathbf{q})\right] \\
&\leq n\left(Pr\left[\|\mathbf{x}_d^{1(1)}\|_H = \|\mathbf{q}_d\|_H\right]\right)^d \\
&\leq n\left(\sum_{j=0}^{\frac{p}{d}} Pr\left[\|\mathbf{x}_d^{1(1)}\|_H = \|\mathbf{q}_d^{(1)}\|_H = j\right]\right)^d \\
&\leq n.\left(\sum_{j=0}^{\frac{p}{d}} Pr\left[\|\mathbf{x}_d^{1(1)}\|_H = j\right] Pr\left[\|\mathbf{q}_d^{(1)}\|_H = j\right]\right)^d \\
&\leq n.\left(\sum_{j=0}^{\frac{p}{d}} Bin(\frac{p}{d}, 1/2)\Big|_j .Bin(\frac{p}{d}, 1/2)\Big|_j\right)^d \\
&\leq n.\left(\sum_{j=0}^{\frac{p}{d}} Bin(\frac{p}{d}, 1/2)\Big|_j .Bin(\frac{p}{d}, 1/2)\Big|_{\frac{p}{2d}}\right)^d \\
&\leq n.\left(\sqrt{\frac{d}{p}}\sum_{j=0}^{\frac{p}{d}} Bin(\frac{p}{d}, 1/2)\Big|_j\right)^d \leq n\left(\frac{d}{p}\right)^{\frac{d}{2}}
\end{aligned}
$$

where we used the fact that the binomial pmf, $Bin(\frac{p}{d}, 1/2)|_j$, is maximized when $j = \frac{p}{2d}$.

Now we are ready to prove Theorem 4.1. We first bound the computational cost of search associated with a single layer, and then aggregate the cost over all of the layers.

In depth s, each node corresponds to a d-pattern where $d = 2^s$. Also, for any query $\mathbf{q}$, the number of (r, d)-neighbors patterns (r-vicinity) for $\mathbf{q}$ are at most $r\binom{d+r-1}{r} \leq d^r$. In other words, in layer s, there are at most $d^r = 2^{rs}$ different potential nodes that are the (r, d)-neighbor patterns of $\mathbf{q}$. The critical observation is that any operation in layer s is performed on a subset of these nodes, but not all of these potential nodes are actually materialized.

In fact, a node is accessed only it is (i) non-empty, and (ii) d-collides with a point in r-vicinity of $\mathbf{q}$. But based on Lemma A.1, the expected number of such nodes—i.e., non-empty nodes in layer s that d-collide with a point in r-vicinity of $\mathbf{q}$—is bounded by $2^{rs}\min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right)$. For each of these nodes, we check at most 2^{rs} potential solutions to

Equation 4.8. Therefore, in layer s, in total we would have at most $2^{rs}2^{rs}\min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right)$ many patterns to check.

The cost of checking whether two d-dimensional binary vectors have the same d-pattern is $O(p)$. Hence, the total cost associated with layer s is $O\left(p2^{2rs}\min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right)\right)$. Finally, we have at most $\log p$ layers, so the total cost is $O\left(2^{2rs}\min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right)p\log p\right)$, which we claim is in fact $O\left(p\log p(\log n)^{4r}\right)$. This is clear when $s < 1 + \log\log n$. Also, if $s \geq 1 + \log\log n$, we can assume $p > \frac{s}{2}$ (because the last layer will not have any children) so $2^{2rs}\min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right) \leq (\log n)^4$ which completes the proof. ■

Appendix B

Proof of Theorem 6.1

In this section, we provide the proof of Theorem 6.1. Let us recall the definition of quantization error for a set of points $\{\mathbf{x}_i\}_{i=1}^n$ and for the (set of m) codebooks $C = [C_1, \ldots, C_m]$.

$$\sum_{i=1}^{n} \min_{\mathbf{b}_{ij} \in \mathcal{B}} \|\mathbf{x}_i - \sum_{j=1}^{m} C_j \mathbf{b}_{ij}\|_2^2 = \sum_{i=1}^{n} \min_{\mathbf{b}_i \in \mathcal{B}_m} \|\mathbf{x}_i - C\mathbf{b}_i\|_2^2 \tag{B.1}$$

where $\mathcal{B} = \{\mathbf{b} \mid \mathbf{b} \in \{0,1\}^k, \|\mathbf{b}\|_1 = 1\}$ and $\mathcal{B}_m = \{[\mathbf{b}_1, \ldots, \mathbf{b}_m] \mid \forall i \in [m], \mathbf{b}_i \in \mathcal{B}\}$. In other words, $\mathcal{B}$ is the set of possible assignments for VQ and $\mathcal{B}_m$ is the set of possible assignments for MCQ.

We assume that the distribution that is generaing the data is supported on the unit ball of $\mathbb{R}^d$ which we denote by $\mathcal{X}$.[1] It is therefore natural to work with codewords who belong to $\mathcal{X}$ as well. Therefore, we define $\mathcal{C}$, the set of all possible codebooks, to be $\mathcal{C} = \{[\mathbf{c_1}, \ldots, \mathbf{c_{km}}] | \forall i \in [km], \mathbf{c_i} \in \mathbb{R}^d, \|\mathbf{c_i}\| \leq 1\}$ (corresponding to m codebooks each having k codewords). Define

$$\|\mathcal{C}\|_{\mathcal{B}_m} = \sup_{C \in \mathcal{C}} \sup_{b \in \mathcal{B}_m} \|Cb\|_2$$
$$b = \sup_{\mathbf{x} \in \mathcal{X}} \sup_{C \in \mathcal{C}} \min_{\mathbf{b} \in \mathcal{B}_m} \|\mathbf{x} - C\mathbf{b}\|_2^2$$

The following theorem – which is a translation of Theorem 2 in [84] to suit our purposes – establishes a uniform convergence result for Z-dimensional coding schemes.

[1] One can scale the points to work with any compact subset of $\mathbb{R}^d$.

Theorem B.1 ([84]). *Let $\mathcal{D}$ be an arbitrary distribution over $\mathcal{X}$, and let $\{\mathbf{x}_i\}_{i=1}^n$ be an i.i.d. sample of size n generated from $\mathcal{D}$. Then with probability at least $1-\delta$ we have*

$$\forall C \in \mathcal{C}, \quad \left| \frac{1}{n} \sum_{i=1}^{n} \min_{\mathbf{b}_i \in \mathcal{B}_m} \|\mathbf{x}_i - C\mathbf{b}_i\|_2^2 - \mathbb{E}_{\mathbf{x}\sim\mathcal{D}} \left(\min_{\mathbf{b}\in\mathcal{B}_m} \|\mathbf{x} - C\mathbf{b}\|_2^2 \right) \right| \leq \frac{Z}{\sqrt{n}} \left(14\|\mathcal{C}\|_{\mathcal{B}_m} + \frac{b}{2}\sqrt{\ln(16n\|\mathcal{C}\|_{\mathcal{B}_m}^2)} \right) + b\sqrt{\frac{\ln(1/\delta)}{2n}}$$

where in our case, $Z = km$, is the dimensionality of the assignments (*i.e.*, $C \in \mathbb{R}^{d\times Z}$).

In order to use this theorem, we need to find upper bounds for $\|\mathcal{C}\|_{\mathcal{B}_m}$ and b in our MCQ application. The following proposition establishes this for us.

Proposition B.1.

$$\|\mathcal{C}\|_{\mathcal{B}_m} = \sup_{C\in\mathcal{C}} \sup_{b\in\mathcal{B}_m} \|Cb\|_2 \leq m$$
$$b = \sup_{\mathbf{x}\in\mathcal{X}} \sup_{C\in\mathcal{C}} \min_{\mathbf{b}\in\mathcal{B}_m} \|\mathbf{x} - C\mathbf{b}\|_2^2 \leq (m+1)^2$$

Proof. The first part holds because

$$\|\mathcal{C}\|_{\mathcal{B}_m} = \sup_{C\in\mathcal{C}} \sup_{b\in\mathcal{B}_m} \|Cb\|_2 = \|\mathcal{C}\|_{\mathcal{B}_m} = \sup_{\{C_j\}_{j=1}^m} \sup_{b_j\in\{e_1,\ldots,e_k\}} \left\| \sum_{C_j} C_j b_j \right\|_2$$
$$\leq \sup_{\{C_j\}_{j=1}^m} \sum_{C_j} \sup_{b_j\in\{e_1,\ldots,e_k\}} \|C_j b_j\|_2 \leq m \sup_{C_1} \sup_{b_j\in\{e_1,\ldots,e_k\}} \|C_1 b_j\|_2 \leq m$$

where $\{e_1, \ldots, e_k\}$ are the standard unit vectors in $\mathbb{R}^k$. For the second part we have

$$b = \sup_{\mathbf{x}\in\mathcal{X}} \sup_{C\in\mathcal{C}} \min_{\mathbf{b}\in\mathcal{B}_m} \|\mathbf{x} - C\mathbf{b}\|_2^2 \leq \sup_{\mathbf{x}\in\mathcal{X}} \sup_{C\in\mathcal{C}} \min_{\mathbf{b}\in\mathcal{B}_m} \left(\|\mathbf{x}\|_2^2 + \|C\mathbf{b}\|_2^2 \right)$$
$$\leq \sup_{\mathbf{x}\in\mathcal{X}} \left(\|\mathbf{x}\|_2^2 \right) + \|\mathcal{C}\|_{\mathcal{B}_m}^2 \leq m^2 + 1 \leq (m+1)^2$$

■

Using Theorem B.1 together with this proposition we have

$$\forall C \in \mathcal{C}, \quad \left| \frac{1}{n} \sum_{i=1}^{n} \min_{\mathbf{b}_i} \|\mathbf{x}_i - C\mathbf{b}_i\|_2^2 - \mathbb{E}_{\mathbf{x}\sim\mathcal{D}} \left(\min_{\mathbf{b}} \|\mathbf{x} - C\mathbf{b}\|_2^2 \right) \right| \leq \frac{(m+1)^2}{2\sqrt{n}} \left[28k + (m+1)k\sqrt{\ln(16nm^2)} + \sqrt{2\ln(1/\delta)} \right]$$