Hashing Techniques: A Survey and Taxonomy

LIANHUA CHI, IBM Research, Melbourne, Australia
XINGQUAN ZHU, Florida Atlantic University, Boca Raton, FL; Fudan University, Shanghai, China

With the rapid development of information storage and networking technologies, quintillion bytes of data are generated every day from social networks, business transactions, sensors, and many other domains. The increasing data volumes impose significant challenges to traditional data analysis tools in storing, processing, and analyzing these extremely large-scale data. For decades, hashing has been one of the most effective tools commonly used to compress data for fast access and analysis, as well as information integrity verification. Hashing techniques have also evolved from simple randomization approaches to advanced adaptive methods considering locality, structure, label information, and data security, for effective hashing. This survey reviews and categorizes existing hashing techniques as a taxonomy, in order to provide a comprehensive view of mainstream hashing techniques for different types of data and applications. The taxonomy also studies the uniqueness of each method and therefore can serve as technique references in understanding the niche of different hashing mechanisms for future development.

Categories and Subject Descriptors: A.1 [**Introduction and Survey**]: Hashing

General Terms: Design, algorithms

Additional Key Words and Phrases: Hashing, compression, dimension reduction, data coding, cryptographic hashing

ACM Reference Format:
Lianhua Chi and Xingquan Zhu. 2017. Hashing techniques: A survey and taxonomy. ACM Comput. Surv. 50, 1, Article 11 (April 2017), 36 pages.
DOI: http://dx.doi.org/10.1145/3047307

1. INTRODUCTION

Recent advancement in information systems, including storage devices and networking techniques, have resulted in many applications generating large volumes of data and requiring large storage, fast delivery, and quick analysis. Such reality has imposed a fundamental challenge on how to accurately and efficiently retrieve/compare millions of records with different data types, such as text, pictures, graphs, software and so forth. To support fast retrieval and verification, applications, such as database systems, often use a short message "key" to represent a record in a large table, such that users can efficiently retrieve items from a large repository.

Indeed, when collecting data for storing or processing, it makes sense to use less information to represent them. This has motivated hashing techniques to transform a data record into shorter fixed-length values or bucket addresses that can represent original data with significantly reduced runtime or storage consumption. Such a transformation

Authors' addresses: L. Chi, IBM Research Australia, 60 City Road, Southbank, VIC 3006, Australia; email: lianhuac@au1.ibm.com; X. Zhu (corresponding author), Dept. of Computer & Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, USA and the School of Computer Science, Fudan University, Shanghai, China; email: xzhu3@fau.edu.

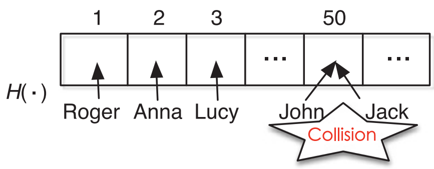


Fig. 1. A hashing example: Each string represents a name and the number represents the spot in the hash space corresponding to the hash values of each string. A hash function $\mathbf{h}(\cdot)$ maps each black string to a corresponding number. The red "Collision" indicates that two different strings are mapped to the same spot and therefore collide.

can be achieved by using a hashing function $\hbar()$ to map original data records to a lower dimensional space, under conditions that (1) each item is randomly/strategically mapped to a spot in the hash space, and (2) two items with the same values will generate the same hash values and are mapped to the same spot. With the help of hash functions, typical data access operations such as insertion, deletion, and lookup on the data can be done in almost constant time with $O(1)$ costs.

When mapping each record to the hash space, data records with different values may have the same hashing output and therefore be mapped to the same spot and result in a collision. Collisions are inevitable if there are more data records than hashing spots, as shown in Figure 1. Accordingly, hashing is mainly challenged by two questions: (1) how to design better hash functions to minimize the collision or increase the accuracy on the basis of high efficiency and (2) when a collision occurs, how to deal with it.

Figure 1 demonstrates the conceptual view of using hashing for efficient data access. A hash function $h(\cdot)$ maps each string as a numeric value. When searching a record, one can hash the query term and directly generate the hashing value corresponding to this record. Because the hashing value can be calculated using constant time, hashing can retrieve a query with $O(1)$ cost, which is much more efficient than searching through all records (typically subject to $O(n)$ or $O(\log n)$ costs for n records).

Hashing can be extremely beneficial for many applications, such as text classification [Chi et al. 2014; Xu et al. 2011a], image retrieval [Chum et al. 2008; Kulis and Grauman 2009; Torralba et al. 2008], multimedia search [Zhu et al. 2013a, 2013b; Song 2015], and data verification [FIPS 1995; Black et al. 1999; Breitinger et al. 2014; Hsieh 2004]. For example, in image retrieval, an essential challenge is to develop efficient similarity measures and fast matching methods with very little memory consumption. In reality, the image database is often very large, and it is difficult or even impossible to store all image information in memory. On the other hand, directly comparing images is time-consuming. Therefore, finding images similar to a query example, from a large image database, is time-consuming. In such cases, it would be beneficial to compress data to speed up the search process. For example, binary-code representation can map similar points in the original feature space to nearby binary codes in the hash code space. The compact representation in hashing can effectively save the storage and achieve fast query for large-scale datasets. In addition, hashing can also help verify whether a large volume of data has been modified by a third party, by comparing hash messages instead of original data, for example, verifying whether software has been maliciously modified to plant virus code [Hsieh 2004].

When using hashing techniques, applications are mainly driven by two distinct motivations: (1) how to retrieve or compare items from a large collection of data or (2) how to verify whether a large volume of information is indeed from its owner, without any change or modification by a third party. The first motivation is data driven, with data access efficiency playing an essential role. On the other hand, the second motivation is security driven, with a data validation purpose. Although they both employ the hashing principle, the focus on either the data or security perspective often results in different hashing techniques and solutions. Data-oriented hashing normally employs two types of approaches, data-independent hashing and data-dependent hashing. If the hashing function set is defined independent of the data to be hashed without involving a training process from the data, we refer to such methods as data-independent hashing. Otherwise, they are classified as data-dependent hashing. For security-oriented hashing, it often uses cryptographically secure hashing or cryptographically insecure hashing approaches, with the former having stricter security properties than the latter.

Many hashing techniques exist for different purposes, and the actual hashing mechanisms vary depending on the data characteristics or the objective of the underlying applications. There is, however, no comprehensive survey providing a complete view of major hashing techniques, the strength/weakness, and the niche of different types of hashing methods. Such limitation has made it technically difficult to design new hashing methods, particularly for data engineers or practitioners.

In this survey, we categorize existing hashing techniques as a hierarchical taxonomy with two major groups: data-oriented hashing versus security-oriented hashing. Our survey will review the uniqueness and strength of mainstream methods in each group, and summarize major domain applications involving hashing. The combined review, from the technique and application perspectives, offers both theoretical and practical views and therefore makes this survey useful for motivating new hashing techniques, as well as serving as a technique reference for real-world implementations.

The remainder of the survey is organized as follows. Section 2 briefly introduces hashing history and defines some terminologies. Section 3 categorizes hashing methods as a hierarchical taxonomy. Section 4 reviews detailed hashing techniques. Section 5 introduces hashing applications, and we conclude the survey in Section 6.

2. HISTORY AND PRELIMINARY

2.1. A Brief History of Hashing

The term “hash” is originated from the physical world, where the standard meaning of “hash” is “chop and mix,” which intuitively means that the hashing function “chops and mixes” information and derives hash results. The importance of hashing techniques has been well recognized since the very early stage of computing systems. Soon after the invention of the first true electronic computer in 1950, the concept of hashing was first mentioned in 1953 [Luhn 1953], where a defined general hash function using random keys could achieve the equivalent of the mathematical concept of uniformly distributed random variables. In computer science, it is almost impossible to get a completely even distribution. Creating even distributions can only be achieved by considering the structure of the keys. For any arbitrary set of keys, it is also impossible to create a general hash function that works better because the keys could not be obtained in advance. In this case, a random uniform hash is the best.

In 1957, motivated by the needs of using a random-access system with very large capacity for business applications, Peterson [1957] provided an estimation of the amount of search required for the exact location of a record in several types of storage systems, including the index-table method and the sorted-file method. In 1968, the word “hashing” was first used [Morris 1968].

2.2. Definitions

Definition 2.1 (*Hashing function*). A hashing function is any function $\hbar(\cdot)$ that can be used to map an arbitrary size of data to a fixed interval [0, m]. Given a dataset containing n data points $X = [x_1, x_2, \ldots x_n] \in \mathrm{R}^D$, and a hashing function $\hbar(\cdot)$, the

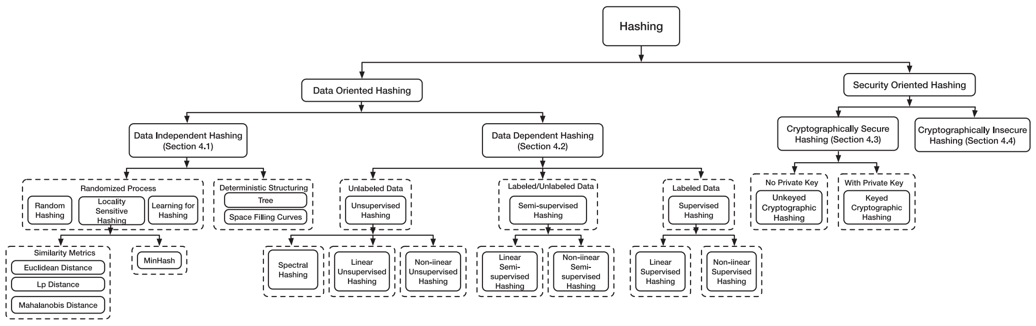

Hashing
Data Oriented Hashing
Security Oriented Hashing
Data Independent Hashing (Section 4.1)
Data Dependent Hashing (Section 4.2)
Cryptographically Secure Hashing (Section 4.3)
Cryptographically Insecure Hashing (Section 4.4)
Randomized Process
Random Hashing
Locality Sensitive Hashing
Learning for Hashing
Deterministic Structuring
Tree
Space Filling Curves
Unlabeled Data
Unsupervised Hashing
Labeled/Unlabeled Data
Semi-supervised Hashing
Labeled Data
Supervised Hashing
No Private Key
Unkeyed Cryptographic Hashing
With Private Key
Keyed Cryptographic Hashing
Similarity Metrics
Euclidean Distance
Lp Distance
Mahalanobis Distance
MinHash
Spectral Hashing
Linear Unsupervised Hashing
Non-iinear Unsupervised Hashing
Linear Semi-supervised Hashing
Non-iinear Semi-supervised Hashing
Linear Supervised Hashing
Non-iinear Supervised Hashing

Fig. 2. The hierarchical taxonomy of hashing techniques. Hashing methods are categorized into two groups: data-oriented hashing versus security-oriented hashing. Each group contains two subgroups, with its content being reviewed in respective subsections.

$\hbar(X) = [\hbar(x_1), \hbar(x_2), \ldots, \hbar(x_n)] \in [0, m]$ can be called the hash values or simply hashes of data points $X = [x_1, x_2, \ldots x_n] \in \mathrm{R}^D$.

One practical use of the hashing function is a data structure called a hash table, which has been widely used for rapid data lookup.

Definition 2.2 (*Nearest neighbor (NN)*). Given a set of n data points $X = [x_1, x_2, \ldots x_n] \in \mathrm{R}^D$, NN represents one or multiple data items in X that are closest to a query point x_q.

Definition 2.3 (*Approximate nearest neighbor (ANN)*). Given a set of n data points $X = [x_1, x_2, \ldots x_n] \in \mathrm{R}^D$, ANN intends to find a data point $x_a \in X$ that is an ε-approximate nearest neighbor of a query point x_q in that for all $x_a \in X$, the distance between x_a and x satisfies $d(x_a, x) \leq (1+\varepsilon)d(x_q, x)$.

3. HASHING TECHNIQUE TAXONOMY

In Figure 2, we categorize hashing techniques as a hierarchical taxonomy from data- and security-oriented perspectives, respectively. From the data-oriented perspective, hashing is primarily used to speed up the data retrieval process, by using data-independent hashing or data-dependent hashing. From the security-oriented perspective, hashing acts as a message digester to generate signatures for verification, where data security is the primary concern. Security-oriented hashing also has two main types, cryptographically insecure hashing and cryptographically secure hashing, considering whether certain security properties are guaranteed, such as collision resistance or preimage resistance [Menezes et al. 1996].

3.1. Data-Oriented Hashing

Data-oriented hashing refers to methods that intend to use hashing to speed up data retrieval or comparison, where a hash table is often maintained for a query.

3.1.1. Data-Independent Hashing. If a hashing function is defined independently of the data to be processed without involving a training process from the data, we refer to such a hashing technique as data-independent hashing. A data-independent hashing method does not have any labeled data/information to help assess the quality of the hashing results. A hashing function is often prespecified, although some of them may learn data distributions to improve hashing results, such as locality-sensitive hashing or learning for hashing. The data-independent hashing functions can be divided into four classes based on the underlying projection modes: random projection, locality-sensitive projection, learning for hashing, and structured projection.

3.1.2. Data-Dependent Hashing. Data-dependent hashing learns hashing functions based on a given set of training data, such that hashing functions can find the best compact codes for all data records. Because data-dependent hashing results are highly sensitive to the underlying data, they are accompanied with faster query time and less memory consumption. In order to better preserve locality information and achieve a better selectivity, data-dependent hashing needs to closely fit the data distributions in the feature space by uniquely defining the hashing function family for a given training dataset. Furthermore, data-dependent hashing usually considers comparing the similarity with the features in the training data.

The data-dependent hashing methods are categorized into three major classes, unsupervised hashing, semisupervised hashing, and supervised hashing, according to the availability of label information in the training data.

Labels provide valuable information to reveal the semantic categorization of each instance (or correlation between instances). Such labels provide additional information, compared to the feature values of each instance, for finding good hash functions suitable for the training data. In addition, even if the actual label of each instance is unknown, one can still specify weak or partial label information, such as pairwise label, to indicate whether some instances are close to each other (e.g., belonging to the same group) or not. Such weak label information is also very helpful for designing hash functions. Methods in this category are commonly referred to as semisupervised hashing. On the other hand, if no label information is provided for training at all, we refer to such hashing as unsupervised hashing.

3.2. Security-Oriented Hashing

Security-oriented hashing refers to methods that use hashing for verification or validation. For example, a user may download software from a public web server but is worried whether the software has been modified by a third party. For verification purposes, the software owner can publish the MD5 [Rivest 1992] hash code of the software. Users can download software from different sources and generate a new MD5 code. If two MD5 codes are identical, it would mean that the downloaded copies are original, and no change has been made to the software. Because MD5 codes are relatively small (e.g., 128 bits), it is much easier than comparing hundreds of megabytes of original data. Meanwhile, because security-oriented hashing codes are often much longer than data-oriented hashing codes, a hash table is often not required or cannot be maintained. Methods in this category are primarily concerned about security properties. For such reasons, they are often computationally expensive and are less efficient, compared to data-oriented hashing methods.

3.2.1. Cryptographically Secure Hashing. Cryptographically secure hashing, or cryptographic hashing in short, refers to methods whose hashing function is designed to be one-way and is very difficult, if not infeasible, to invert. When applying such methods, the length of input (also called "message") is arbitrary, and the size of output (also called "message digest") is fixed. The fixed-size hash results are used as a signature to represent the original message for validation. Due to such a security-sensitive nature, cryptographic hashing has a strict avalanche effect, which requires that a hash output changes significantly (e.g., ,approximately half the output bits change) if there is a change of the input (e.g., changing one single bit in the input).

For cryptographic hashing, three properties are normally enforced: (1) preimage resistance, (2) second preimage resistance, and (3) collision resistance. The preimage resistance means that the input (message) is difficult to find if only output (message digest) is known (i.e., the one-way attribute). The second preimage resistance refers to the property that given a message m_i and its hash output $hash(k,m_i)$, where k is the

Table I. Categorization of Data-Independent Hashing Methods

Random Hashing	Random Projection Hashing; Universal Hashing
Locality Sensitive Hashing	Locality-Sensitive Hashing (LSH)
	MinHash; Weighted MinHash (WMH)
	Shift-Invariant Kernel-Based Hashing (SIKH)
	Nested Subtree Hashing (NSH)
	Discriminative Clique Hashing (DICH)
Learning for Hashing	BoostMap
Structured Projection	Quadtree; Hilbert Curve; Z curve

hash key, it is difficult to find another message m_j satisfying $hash(k,m_i = hash(k,m_j)$ (i.e., ,the second-preimage attacks). The collision resistance requires that two messages m_i and m_j should have different hash results in order to avoid a birthday attack (i.e., attackers can find two input messages with the same hash output).

Applications using cryptographic hashing include digital signature, public key cryptography, and message authentication. The cryptographic hashing can be further divided into two categories, keyed cryptographic hashing and unkeyed cryptographic hashing, depending on whether a secret key is used by the hashing function.

3.2.2. Cryptographically Insecure Hashing. While cryptographically secure hashing has nice security properties, they are often computationally inefficient. For applications without strong security concerns, a simpler hashing mechanism, called cryptographically insecure hashing or noncryptographic hashing, is more practical. For noncryptographic hashing, such as the Fowler-Noll-Vo (FNV) hash function [Fowler 1991], the main objective is still to generate hash output for verification, but the hashing process does not have to consider cryptography. As a result, it becomes possible to have faster processing, a lower collision probability, a higher probability of detecting small errors, and easier collision detection, compared to cryptographically secure hashing. This kind of hashing method is especially popular in applications that require fast searches or processing, such as Twitter, Domain Name Service (DNS) servers, or database implementations.

4. HASHING TECHNIQUE DESCRIPTION

In this section, we first review data-oriented and security-oriented hashing methods by separating them into four subsections, as outlined in Figure 2. After that, we will summarize time complexity of the methods across all four categories.

4.1. Data-Independent Hashing Methods

In the following, we first introduce the most simple projection method, random projection, and then advance to locality-sensitive projection to preserve the locality characteristics of data. After that, we will further introduce learning hashing projection and structured projection. The hashing algorithm category is in Table I.

4.1.1. Random Hashing. Random projection hashing is a general data reduction technique, which randomly projects original high-dimensional data into a lower-dimensional subspace. For example, an original data item with d-dimensional features can be projected to k-dimensional ($k \ll d$) subspace after random projection hashing.

Random projection hashing was first proposed [Donald 1999] to use a random function $h : U \rightarrow V$ to generate a random hash value $h(x)$ in domain V (corresponding to the k-dimensional data) and be associated with data items in the original domain U (corresponding to the d-dimensional data). In random projection, a random function requires $d \times \lg k$ bits to represent, which leads to the infeasibility of storing a randomly chosen function. Accordingly, some researchers began to use fixed functions in the random projection. Carter and Wegman [1977] proposed a universal hashing approach, which randomly chooses hashing functions from a small family of particular functions, instead of from all functions. As a result, it guarantees the provable performance and achieves feasible and succinct storage of hash functions. For example, in Shakhnarovich [2005], a task-specific similarity measure was proposed to choose hash functions uniformly from a family F of hash functions:

$$\{x \to ((ax + b) \bmod p) \bmod v \mid 0 < a < p, 0 \le b < p\}. \tag{1}$$

The whole family is defined by the parameters p and v, and a particular hashing function is defined by the parameters a and b.

In this universal hashing, each set of n elements in U is uniformly projected to random and independent values, and the corresponding family F is *n-wise independent*. Wegman and Carter [1981] proposed such function families where a random function requires $n \lg d$ bits of space to store. For quite a long time, the time complexity of all n-wise independent families to evaluate a hash function was $O(n)$. However, an important breakthrough made in Siegel [2004] proposed extremely random constant-time hash functions, where the hash families are relatively small and highly independent so they can be evaluated in constant time.

Although random projection is technically simple and computationally efficient, its main drawback is the high instability. In other words, different random hash functions may lead to totally different hashing values. On the other hand, if two elements differ in one bit, they will have two different hash values and be projected to two completely different random spots. Therefore, pure random-projection-based hashing inherently discards the characteristics of the original feature space and cannot achieve good performance for certain applications. In order to preserve the data characteristics in the original feature space, locality-sensitive hashing was introduced.

4.1.2. Locality-Sensitive Hashing. The most commonly known data-independent method with randomized projection is locality-sensitive hashing (LSH) [Indyk and Motwani 1998; Gionis et al. 1999]. Due to its wide popularity, we briefly introduce this representative method, along with its advantages and disadvantages.

LSH: Given a dataset containing n data points/items $X = [x_1, x_2, \ldots x_n] \in \mathrm{R}^D$, and hashing functions $H(\cdot)$ containing K hashing functions, LSH maps a data point x_i to a K-bits hash code $\in \{0,1\}$:

$$H(x_i) = [\hbar_1(x_i), \hbar_2(x_i), \ldots, \hbar_k(x_i)]. \tag{2}$$

An important feature of LSH is that two data points within a smaller distance, in the original feature space, are more likely to have similar hash codes. In other words, in the Hamming space, LSH largely preserves the original locality information. Such a locality-preserving property can be elaborated using the following equation between two data points x and y:

$$P\{H(x) = H(y)\} = sim(x, y), \tag{3}$$

where $sim(x, y)$ is the similarity measure and can be represented by a distance function, such as p-norm distance ($p \in (0, 2]$) [Datar et al. 2004], Mahalanobis distance [Shakhnarovich 2005], angular similarity [Charikar 2002; Ji et al. 2012], and kernel similarity [Kulis and Grauman 2009]. For the random projection hashing, LSH has

$$H(x) = sign(w^T x + b), \tag{4}$$

where w represents a random hyperplane from the geometric point of view and b represents a random intercept. The label of each data point is determined by the side of the hyperplane w. Thus, LSH satisfies

$$H(x) = \begin{cases} 0 & if\ w^T x < b \\ 1 & otherwise \end{cases} . \tag{5}$$

Specifically, based on this hash function, for any two points $x, y \in \mathrm{R}^D$, LSH satisfies

$$P\left\{\hbar_m(x) = \hbar_m(y)\right\} = 1 - \frac{1}{\pi}\cos^{-1}(x^T y) . \tag{6}$$

LSH not only preserves data characteristics in the hash space but also guarantees the collision probability between similar data points. Despite its noticeable advantages, LSH still has an unavoidable disadvantage that is the inefficiency of the hash code. First, the random generation of hash functions and independency of data in LSH cannot guarantee the efficiency. Second, LSH usually needs long codes in each hash table to guarantee an acceptable accuracy, which increases the lookup complexity. According to Equation (6), the collision probability decreases exponentially with respect to the code length, so LSH needs long binary codes to achieve good precision although enjoying asymptotic theoretical properties, which also results in low recall when creating a hash lookup table. Accordingly, several recent research works have been focused on generating short compact hash codes. Among all these efforts, data-dependent hashing has drawn significant attentios (as we will soon review in Section 4.2).

With the help of simple random projections, two objects within a smaller distance are more likely to have the same LSH hash codes. For similarity measures in LSH, many methods [Datar et al. 2004; Shakhnarovich 2005; Charikar 2002; Kulis and Grauman 2009; Ji et al. 2012] use p-norm distances for $p \in (0,\ 2]$ [Datar et al. 2004], Mahalanobis distance [Shakhnarovich 2005], angular similarity [Charikar 2002; Ji et al. 2012], and kernel similarity [Kulis and Grauman 2009]. The basic idea of LSH is to choose a random hyperplane at the outset and use the hyperplane to hash input vectors. The hyperplanes are often used to partition data points into two sets with two different binary codes being assigned based on the set each data point is assigned to. For the hyperplane, in order to optimally allocate a variable number of bits per LSH hyperplane, Moran et al. [2013a, 2013b] proposed dubbed Neighborhood Preserving Quantization (NPQ) [Moran et al. 2013a] and dubbed Variable Bit Quantization (VBQ) [Moran et al. 2013b]. The former assigns multiple bits per hyperplane based on adaptively learned thresholds, and the latter provides a data-driven nonuniform bit allocation across hyperplanes. Base on the randomized projection of LSH, Zhang et al. [2013] proposed Distribution-Aware LSH (DALSH), which uses data-adaptive projection to address the problem of lacking adaptation to real data.

MinHash, also known as the min-wise independent permutations locality-sensitive hashing scheme, is another LSH-related hashing technique commonly used for text mining or applications requiring quick similarity assessment between two sets [Broder 1997; Bharat and Broder 1998]. MinHash is efficient to estimate the similarity between two sets, S_1 and S_2, each containing a number of elements (e.g., finding similarity between two documents by using shared keywords). In order to find similarity between S_1 and S_2, K hash functions (random permutations) are applied to elements in S_1 and S_2, respectively. The minimum hash value of S_1, S_2 is the MinHash of S_1, S_2. By checking minimum hash values between two sets, one can quickly assess the similarity between S_1 and S_2 without involving a complicated set operations to check their

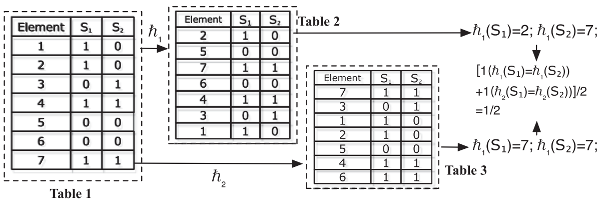


Fig. 3. An example of MinHash process: In the last two columns of the tables, "1" or "0" represents that the corresponding element is in the set (S_1or S_2) or not; the $\hbar_1$ and $\hbar_2$ represent two different hash functions.

member relationships, as defined in Equation (7):

$$\hat{\jmath}(S_1, S_2) = \frac{\sum_{k=1}^{K} 1(\hbar_k(S_1) = \hbar_k(S_2))}{K}, \tag{7}$$

where $1(\hbar_k(S_1) = \hbar_k(S_2)) = 1$ if $\hbar_k(S_1) = \hbar_k(S_2)$, and $1(\hbar_k(S_1) = \hbar_k(S_2)) = 0$ otherwise. As $K \to \infty$, $\hat{\jmath}(S_1, S_2) \to \jmath(S_1, S_2)$.

In addition to its computational efficiency, MinHash also enjoys a very nice property: the probability that S_1 and S_2 can be hashed to the same min-hash value is equal to the Jaccard similarity of S_1, S_2, as given in Equation (8). In other words, min-hash-based similarity is equivalent to Jaccard similarity, a commonly used set similarity measure, if a sufficiently large number of permutations are used:

$$Pr(\hbar(S_1) = \hbar(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \jmath(S_1, S_2). \tag{8}$$

In Figure 3, we demonstrate an example of the Min-wise hashing process. Given two sets $S_1 = \{1,2,4,7\}$ and $S_2 = \{3,4,7\}$, and two independent random element permutations $\hbar_1 = [2,5,7,6,4,3,1]$ and $\hbar_2 = [7,3,1,2,5,4,6]$, for set S_1, the minimum $\hbar_1$ hash value is 2, and the minimum $\hbar_2$ hash value is 1. For set S_2, the minimum $\hbar_1$ hash value is 7, and the minimum $\hbar_2$ hash value is 1. Therefore, the similarity between S_1 and S_2 is 1/2.

The Min-wise hashing scheme was initially introduced in Broder [1997] and later used to detect and eliminate duplicate web pages from searching results in the AltaVista search engine [Bharat and Broder 1998], large-scale document clustering [Broder 1997], near-duplicate image detection [Chum et al. 2008], and large-scale text classification [Chi et al. 2014], which uses a Recursive Min-wise Hashing (RMH) to preserve the context information. Similarly, another kind of similarity hashing function called SimHash [Sadowski and Levin 2007] is normally used to determine file similarity. In order to compare MinHash with SimHash, Shrivastava and Li [2014c] provided the first provable way to compare them under different similarity measures and verified that MinHash is provably better than SimHash even for cosine similarity.

Even though the MinHash was verified to be an efficient hashing method, there is still much space to be improved. In order to save storage space, Li et al. [2010], Li and Konig [2010, 2011], and Li et al. [2011] extended MinHash techniques to a *b*-bit Min-wise hashing by changing the traditional 64 bits used to store each hashed value in Min-wise hashing methods. In order to make Min-wise hashing faster, Shrivastava and Li [2014a, 2014b] made use of the idea of permutation and densification to offer massive savings in computation cost compared to Min-wise hashing. From another perspective inspired by the weighted sampling, Manasse et al. [2010], Ioffe [2010], and Shrivastava [2016] gradually proposed a Weighted Minwise Hashing (WMH) and verified that WMH is much simpler, significantly faster, and more memory efficient.

Another random projection hashing technique includes Shift Invariant Kernel-Based Hashing (SIKH) [Shakhnarovich 2005; Raginsky and Lazebnik 2009], Nested Subtree Hashing (NSH) [Li et al. 2012], and Discriminative Clique Hashing (DICH) [Chi et al.

2013]. Shakhnarovich [2005] and Raginsky and Lazebnik [2009], based on random projections, introduced a simple distribution-free encoding scheme, which could relate the expected Hamming distance between two sets of binary codes corresponding to two vectors to the value of a shift-invariant kernel between the two vectors. For data mining applications involving networked data with dynamically increasing volumes, Li et al. [2012] and Chi et al. [2013] proposed new hashing schemes to address the problem of large-scale graph classification over streams. In Li et al. [2012], we proposed NSH to extract multiresolution subtree patterns from graph streams and hash each pattern to a low-dimensional feature space. In Chi et al. [2013], two random hashing schemes were employed to speed up the clique-pattern mining process and address the unlimited clique-pattern expanding problem.

4.1.3. Learning for Hashing. Learning for hashing represents a set of data-sensitive hashing methods learning a new hash space with respect to the given data. Shakhnarovich et al. [2006] proposed BoostMap, a data-independent machine-learning method for Euclidean embedding construction.

BoostMap intends to learn a new embedding space for each task specified by the underlying data, so the new hash space can optimally reveal the data similarity. It is an efficient approximate nearest-neighbor method and can be used in arbitrary distance measures, metric or nonmetric. Before introducing BoostMap, we first introduce some basic methods for constructing Euclidean embeddings, such as Lipschitz embedding [Johnson and Lindenstrauss 1984], Bourgain embedding [Hjaltason and Samet 2003; Bourgain 1985], FastMap [Faloutsos and Lin 1995], MetricMap [Wang et al. 2000] and BoostMap [Shakhnarovich et al. 2006].

The basic idea of Lipschitz embedding is to embed metric spaces into other spaces with low distortion. In Lipschitz embedding, an object $x \in X$ (a space) is transformed into an n-dimensional vector $V = (v_1, v_2, \dots, v_n)$ such that each element v_i corresponds to the distance of object $o \in X$ to a predefined reference set [Hjaltason and Samet 2003]. The Bourgain embedding is a special type of Lipschitz embedding.

The equation $P_o(x) = D_X(x, o)$ can represent the $1D$ Euclidean embedding P_o in space X. The object o is called a reference object. If D_x is obedient to the triangle inequality, the nearby points in X are intuitively mapped to nearby points on the real line R by P_o. Even though D_x violates the triangle inequality, the nearby points in X may still be mapped to nearby points in R by P_o [Athitsos and Sclaroff 2003]. On the other side, it is also probable for distant points to be mapped to nearby points.

In FastMap, the basic idea is to choose two data objects $x_1, x_2 \in X$ as pivot objects, and then define the embedding $P^{x1,x2}$ of arbitrary x as the projection of x onto the line between x_1 and x_2. FastMap defines the projection according to the distance between x and x_1 and between x and x_2, respectively. The distances are then treated as the sides of a triangle:

$$P^{x_1,x_2}(x) = \frac{D_X(x, x_1)^2 + D_X(x_1, x_2)^2 - D_X(x, x_2)^2}{2D_X(x_1, x_2)}. \tag{9}$$

Figure 4 demonstrates an example of the FastMap projection used in Equation (9).

In Equation (9), $P^{x1,x2}$ projects data objects, which are mutually close in the original space, to nearby locations and preserves the proximity structure of data objects. In FastMap, multiple pairs of pivot objects are used to project a finite set of data objects. MetricMap [Wang et al. 2000] extends the FastMap and maps the finite data object set onto a pseudo-Euclidean space, and outperforms FastMap when using non-Euclidean.

BoostMap [Shakhnarovich et al. 2006] optimizes the quantitative measure and preserves better similarity rankings than using random choices and heuristics. Meanwhile, the learning of BoostMap is independent and does not require an original distance measure, such as Euclidean or metric properties. The key point of the learning

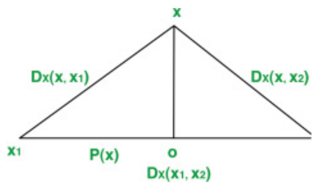


Fig. 4. Example of FastMap projection used in Equation (9). The length of the sides xx_1, xx_2, x_1x_2 are $D_X(x, x_1)$, $D_X(x, x_2)$, and $D_X(x_1, x_2)$, respectively. A line xo from x is perpendicular to x_1x_2, and the length of line segment x_1o is equal to $P^{x1,x2}(x)$.

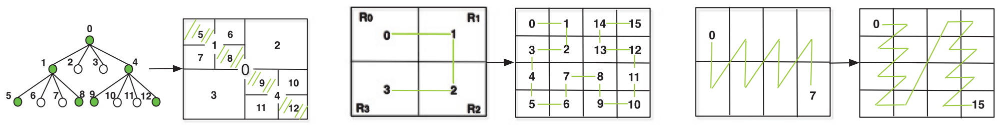
process in BoostMap is to treat embedding as classifiers in order to estimate the distance of any three data objects, and use AdaBoost [Schapire and Singer 1999] to unite all previous lower-dimensional embedding into one higher-dimensional embedding for a higher-accuracy similarity ranking. The main process of BoostMap is as follows: after identifying a big family of $1D$ embeddings P based on a pair of pivot objects or a reference object, each P is treated as a continuous output binary classifier and a *weak classifier* [Schapire and Singer 1999]. After that, BoostMap uses AdaBoost to combine $1D$ embedding P into a multidimensional embedding that serves as a *strong classifier* with relatively higher accuracy than a *weak classifier*. BoostMap makes full use of the advantage of machine-learning techniques to assemble a higher-accuracy embedding from many one-dimensional embeddings.

4.1.4. Structured Projection. Many hashing methods exist, but they are mainly effective for low-dimensional data. When applying such methods to data with a high (or very high) dimensionality, their performance may degrade. For similarity search in High-Dimensional Vector Spaces (HDVSs), a conventional approach is to use a multidimensional index structure that requires data space partitioning.

Structured projection hashing methods represent a set of approaches that partition data space along predefined lines, regardless of data features, where different hashing methods define their unique line partitioning structures. For example, Weber et al. [1998], Joly et al. [2004], and Poullot et al. [2007] proposed data-independent hashing schemes with structured projection, including tree [Weber et al. 1998] and space-filling curves [Joly et al. 2004 and Poullot et al. 2007].

Weber et al. [1998] studied the impact of dimensionality on nearest-neighbor similarity search on HDVSs and demonstrated that any partitioning scheme and clustering technique must degenerate to a sequential scan through all their blocks if the dimensionality is sufficiently high. In the paper, some tree structures are used to partition data space, and their results showed that, for high-dimensional data, tree-based structures outperform sequential scanning by orders of magnitude. An example is shown in Figure 5(a) that uses Quadtree (tree data structure) based data space partitioning [Finkel and Bentley 1974]. In Quadtree, each internal node has exactly four children or no children. It is mainly used to recursively divide and subdivide a two-dimensional space into four quadrants or regions. In the right panel of Figure 5(a), the space is divided into four quadrants, with node 0 being in the center. Node 1 on one side of the plane forms one region, and nodes 2, 3, and 4 on the other side form another region. In this way using planes, recursively partitioning space produces the tree showing on the left panel of Figure 5(a).

Joly et al. [2004] and Poullot et al. [2007] studied the content-based copy identification by space-filling curve projection. More specifically, Joly et al. [2004] proposed a novel strategy dedicated to pseudo-invariant feature retrieval suitable for content-based copy identification. They adopted the Hibert curve as the line of projection and directly mapped an approximate search range onto a Hilbert space-filling curve in order to



(a) Data space partitioning by Quadtree

(b) 2D data space partitioning by using Hilbert curve at depth 3 (left) and 4 (right)

(c) 2D data space partitioning by using Z curve at depth 3 (left) and 4 (right)

Fig. 5. (a) Each node in the tree represents a bounding rectangle (in the right split rectangle) that covers some part of the whole space; root node "0" represents the entire space; the shaded region represents the hash prefixes or ranges corresponding to the nodes "5," "8," "9," and "12" in the left tree. (b) The green line illustrates the order in which a Hilbert curve "visits" each quad. (c): The green line illustrates the order in which the hashing approach "visits" each quad.

establish efficient access to the database. The advantage of the Hilbert curve is that it can guarantee that two cells that are neighbors in the index are also neighbors in the description space [Jagadish 1997].

Figure 5(b) demonstrates a 2D data space partitioning by using the Hilbert curve at depth 3 (left) and 4 (right). In Figure 5(b) (left), the Hilbert curve replicates in four quadrants R_0, R_1, R_2, and R_3. In the process of replicating, the quadrant R_0 is rotated clockwise 90°, and the quadrant R_3 is rotated counter-clockwise 90°. After rotation, the sense of both lower quadrants R_0, R_3 is reversed. The two upper quadrants R_1, R_2 will not be rotated. According to this rule, we can obtain a Hilbert curve of depth 4. All rotation and sense computations are relative to the previously obtained rotation and sense in a particular quadrant. Figure 5(b) (left) shows the basic Hilbert curve of a 2*2 grid. The procedure to derive a deeper depth of this curve is to rotate and reflect the curve at R_0 and R_3. The curve can keep growing recursively by following the same rotation and reflecting pattern at each vertex of the basic curve. Figure 5(b) shows Hilbert curves of depth 3 and 4, respectively.

For the Hilbert curve, one disadvantage is that it is difficult to compute the key in the index starting from the position in the description space for high-dimensional space and higher-order partitioning. In order to simplify the computation of the keys (cell addresses in the index) and to tightly link it to a component-wise search process, Poullot et al. [2007] replaces the Hilbert curve by using a Z space-filling curve and hierarchically partitions the description space into hyperrectangular cells following the Z-curve, as shown in Figure 5(c). The advantage of the Z-curve for space partitioning is clear: regardless of the depth, all the cells are partitioned along the same dimensions.

4.2. Data-Dependent Hashing Methods

The data-dependent hashing can be divided into three categories: unsupervised hashing, semisupervised hashing, and supervised hashing.

4.2.1. Unsupervised Hashing. For unsupervised hashing, no label information, including weak labels such as pairwise labels between instances, is provided. Unsupervised hashing methods use unlabeled data to generate binary codes for given training data and try to preserve the similarity information in the original feature space.

According to the actual forms of functions used for hashing, including eigenfunctions, linear functions, and nonlinear functions, we categorize unsupervised hashing approaches into three types: spectral hashing, linear hashing, and nonlinear hashing. The categories and some important algorithms are summarized in Table II.

Spectral Hashing. Spectral hashing (SH) [Weiss et al. 2009] is one of the most popular data-dependent unsupervised hashing methods. Many methods use spectral hashing to address the problem of learning hashing code with semantics. For example,

Table II. Categorization of Unsupervised Hashing Methods

Unsupervised Hashing	**Spectral Hashing**	
	Linear Unsupervised Hashing	Anchor Graph Hashing (AGH)
		Product Quantization (PQ)
		Angular Quantization-Based Binary Coding (AQBC)
		Spherical Hashing
		Isotropic Hashing
		Manhattan Hashing
		Predictable Dual-View Hashing (PDH)
		Inductive Manifold Hashing (IMH)
		Locally Linear Hashing (LLH)
		Topology Preserving Hashing (TPH)
	Nonlinear Unsupervised Hashing	Kernelized LSH (KLSH)
		Multiple Feature Kernel Hashing (MFKH)

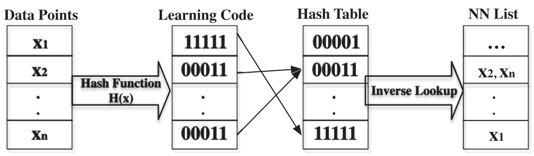


Fig. 6. The process of learning compact codes.

Salakhutdinov and Hinton [2009] proposed to design compact binary codes for a large number of documents, under the objective that documents with a short Hamming distance in terms of the hashing codes are semantically similar to each other. Weiss et al. [2009] defined a hard criterion for a good code that is related to graph partitioning.

Due to the popularity of spectral hashing, we now introduce its detailed procedures. The common process of learning compact codes in SH is shown in Figure 6, where the far left list represents a set of data points. By using hash function $H(x)$, one can learn the corresponding compressed code of each data point. After that, the NN list is obtained by using inverse lookup in the hash table. Therefore, spectral hashing not only maintains the sample similarity in the reduced Hamming space but also seeks the ones where the average Hamming distance between similar data points is minimal.

Spectral hashing assumes that the inputs are embedded in R^D, and $\mathrm{A}_{ij} = \exp(-\|x_i - x_j\|^2/\epsilon^2)$. A_{ij} is the similarity between a pair of data points (x_i, x_j) in the input space. The parameter ϵ defines the distance in R^D, which corresponds to similar items. Based on these settings, the average Hamming distance between similar neighbors is $\sum_{ij} A_{ij}\|y_i - y_j\|^2$.

As a result, spectral hashing codes satisfy the following criteria [Weiss et al. 2009]:

$$Minimize : \frac{1}{2}\sum_{ij} A_{ij}\|y_i - y_j\|^2 = \mathrm{tr}(\mathrm{y^T Ly})$$

$$Subject\,to : (1) y \in \{-1, 1\}^{n*K}; (2) 1^T \mathrm{y} = 0; (3) \frac{1}{n} y^T y = I_{K*K},$$

where L is the graph Laplancian diag $(A\mathbf{1})$- A, the constraint $1^T \mathrm{y} = 0$ defines the bits to be balanced, and the constraint $\frac{1}{n} y^T y = I_{K*K}$ specifies the bits to be uncorrelated.

Spectral hashing includes training and hashing three major steps: (1) building the sparse affinity matrix A of an exact neighborhood graph on n data points, (2) computing and binarizing K eigenvectors of the graph Laplacian L, and (3) generalizing K eigenvectors to the testing data points and binarizing.

In summary, spectral hashing performs Principle Component Analysis (PCA) on training data and fits a multidimensional rectangle. Despite its simplicity, it is in fact effective and comparable to many advanced hashing methods. But its training process is intractable for offline, and hashing process is infeasible for online.

In order to evaluate the spectral hashing results, the quality of the hashing is typically assessed using the following criteria [Salakhutdinov and Hinton 2009]: (1) easy computation for a novel input, (2) need for a smaller number of bits to code the full dataset, and (3) similar data points with similar binary codewords.

Because it is often a time-consuming process to obtain supervised information, as a well-recognized unsupervised codeword generation method, spectral hashing minimizes the codeword distances between similar points to learn short binary code words. Although spectral hashing has shown promising performance by learning the binary codes with a spectral graph partitioning method, its performance may become worse as the number of bits increases, and the construction of the graph Laplacian by the Euclidean distance may not reflect the inherent distribution of the data. Meanwhile, although it is straightforward to do the calculation for inputs in the training data, it is a problem to compute the hash codewords for previously unseen data.

In order to address these problems, many methods [Weiss et al. 2012; Li et al. 2013; Bodó and Csató 2014] exist to extend spectral hashing to solve specific challenges. For example, Multidimensional Spectral Hashing (MSH) [Weiss et al. 2012] is proposed to ensure the stability of performance. MSH seeks to reconstruct the affinity between data points, rather than their distances. In order to reflect the underlying distributions of the data, Li et al. [2013] proposed a method based on the pairwise similarities of image labels/tags to directly optimize the graph Laplacian, and this method can automatically determine the scale factor during the optimization. In order to compute hash codewords for data points that are previously unseen, Bodó and Csató [2014] proposed to use linear scalar products as similarity measures and use different generalization (an inductive generative formula) to find hash codes. Due to the generalization, for a new data point, the codeword generation method and random hyperplane-based LSH are similar.

Linear Unsupervised Hashing. Linear unsupervised hashing refers to a set of methods whose hashing functions are linear functions, although the algorithms may rely on learning-based approaches to derive actual hash functions [Liu et al. 2011; Brandt 2010; Zhang et al. 2010a, 2010b; Gong and Lazebnik 2011; Joly and Buisson 2011; Xu et al. 2011b; Wang et al. 2006, 2010b; Kang et al. 2012; Xu et al. 2012; Zhen and Yeung 2012; He et al. 2013]. Most of these hashing algorithms focus on exploiting the spectral properties of the data affinity matrix for binary coding. Among them, Anchor Graph Hashing (AGH) [Liu et al. 2011] is a popular approach. As a graph-based hashing method, AHG learns appropriate compact codes by automatically discovering the neighborhood structure in the data.

In the following, we briefly describe the anchor graph hashing method:

AGH: In order to make hashing computationally feasible, AGH utilizes anchor graphs to obtain tractable, nonnegative, sparse, and low-rank affinity matrices. Anchor points can be seen as *K-means* clustering centers, and an anchor graph can approximate the exact neighborhood graph when the number of anchors is sufficiently large.

AGH first computes the data-to-anchor similarity and obtains a data-to-anchor similarity matrix $Z \in \mathrm{R}^{n*m}$ (n data points, m anchor points). Based on Z, the data-to-data similarity matrix $\hat{\mathrm{A}} \in \mathrm{R}^{n*n}$ can be obtained. The process is shown in Figure 7, where red data points are anchor points, and $\hat{\mathrm{A}}_{ij} = \sum_{k=1}^{m} Z_{ik} Z_{jk} = Z_i {Z_j}^T$. So the anchor graph

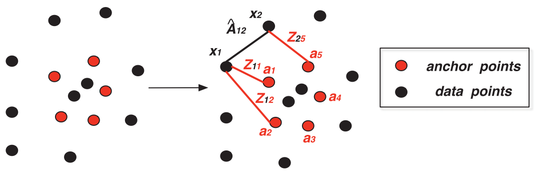


Fig. 7. Matrix Z and matrix $\hat{A}$: the lines between the data points represent the distances between the data points.

affinity matrix $\hat{A} = Z\wedge^{-1}Z$ ($\wedge = \text{diag}(Z^T\mathbf{1}) \in R^{m*m}$), which is different from affinity matrix A in spectral hashing.

AGH seeks a K-bit Hamming embedding $y \in \{1, -1\}^{n*K}$ for n data points in the database by minimizing

$$\max: \ \text{tr}(y^T\hat{A}y)$$

$$\text{Subject to: } (1)1^T y = 0; (2) y^T y = nI_{K*K}.$$

In summary, AGH has five major steps, including training (Steps 1, 2, and 3) and hashing (Steps 4 and 5): (1) building an anchor graph and obtaining a data-to-anchor similarity matrix Z on n data points, (2) computing and binarizing K eigenvectors of the anchor graph Laplacian, (3) building the hash table, (4) generalizing K eigenvectors to the testing data points and binarizing, and (5) inverse lookup in the hash table.

AGH is a scalable graph-based unsupervised hashing approach that considers the underlying manifold structure of the data to search nearest neighbors. AGH ensures linear training time and constant hashing time by extrapolating anchor graph Laplacian eigenvectors to eigenfunctions. In AGH, r anchor graph Laplacian eigenvectors are used to generate r-bit codes, and the higher eigenvectors corresponding to the higher graph Laplacian eigenvalue are of low quality for partitioning. In order to solve this problem, AGH uses a two-layer hashing to revisit the lower graph Laplacian eigenvectors to generate multiple hash bits.

In order to improve the hashing performance, many existing methods propose to use a large number of hash tables (long codewords), which result in significant costs in space consumption. Some solutions [Lu et al. 2006; Kontak et al. 2012; Lin et al. 2013] have been proposed to address this problem. Lu et al. [2006] considered a hardware-friendly scheme for Minimal Perfect Hashing (MPH) via counting Bloom filters to reduce the number of memory accesses to just $O(1)$ and still remains space efficient. In order to perform cost-effective and exact pattern matching, HashMem [Kontak et al. 2012] was proposed to combine hashing and memories by using hashing to generate a distinct address for each candidate pattern stored in memory. Lin et al. [2013] developed a hashing algorithm Compressed Hashing (CH) for high-dimensional nearest-neighbor search by combining the techniques of sparse coding and compressed sensing.

Other important linear unsupervised hashing methods include ANN search algorithm Product Quantization (PQ) [Jegou et al. 2011] and Angular Quantization-Based Binary Coding (AQBC) [Gong et al. 2012] for high-dimensional nonnegative data that commonly exist in vision and text applications. Spherical Hashing [Heo et al. 2012] maps spatially coherent data points into a binary code compared to hyperplane-based hashing functions. Isotropic Hashing (IsoHash) [Kong and Li 2012] learns projection functions that could generate projected dimensions with isotropic variances (equal variances). Manhattan hashing (MH) [Kong et al. 2012] uses Manhattan distance to deal with the destruction of the neighborhood structure in the original feature in Hamming-distance-based hashing. Predictable Dual-View Hashing (PDH) [Rastegari et al. 2013] embeds proximity of data samples in the original spaces, and Inductive

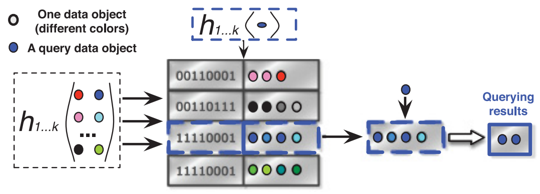


Fig. 8. The core idea of kernelized LSH: the $h(\cdot)$ represents a hash function; the digital code represents the hashing value of the data object.

Manifold Hashing (IMH) [Shen et al. 2013] connects manifold learning methods and hash function learning. Most Recently, Irie et al. [2014] proposed Locally Linear Hashing (LLH) to preserve the locally linear manifold structures of high-dimensional data in a low-dimensional Hamming space. Another latest unsupervised hashing method, Topology Preserving Hashing (TPH), was proposed in Zhang et al. [2014a] for preserving neighborhood relationships and relative neighborhood proximities.

Nonlinear Unsupervised Hashing. Instead of using linear hash functions, nonlinear unsupervised hashing uses nonlinear functions, typically some kernel functions, for unsupervised hashing [Kulis and Grauman 2009; Joly and Buisson 2011; He et al. 2010; Liu et al. 2012a]. Kulis and Grauman [2009] extended the accessibility of LSH to generic kernel space and proposed Kernelized LSH (KLSH). The main idea of KLSH is to construct a random hyperplane hash function in kernel space based on a central limit theorem. According to the central limit theorem, under very mild conditions, the mean of a set of data objects from some underlying distribution will largely follow Gaussian distribution in the limit, as the number of data objects in the set increases. Following this central limit theorem, an approximate random vector will be computed by using data items from the database. Once the random hyperplane hash function is constructed, KLSH computes a small set of candidates approximating nearest neighbors by the method of Charikar. After that, KLSH sorts them to produce a list of hashed nearest neighbors by the kernel function. As a result, the nearest neighbors of a query can be retrieved in sublinear time by using standard LSH techniques. Figure 8 shows the main idea of kernelized LSH.

In Figure 8, the blue point represents the query data object. Hashing functions $\hbar_{1...k}$ are applied to map data objects to a set of compact codes. For the same compact code, there may be more than one data object that tend to be similar to each other. For compact code 11110001, there are four data objects. By mapping the query data point to the same compact code, it narrows the scope of searching down to a small range. As a result, one can easily find the blue object represented by code 11110001.

A noticeable advantage of KLSH is that there is no assumption about the input and the data distributions. As a result, it makes KLSH very suitable for image search and other domains where underlying data distributions are unknown.

In reality, KLSH's performance deteriorates when the number of code bits is small. Motivated by practical requirements in large-scale problems, He et al. [2010] proposed a general hashing algorithm that can work on general data types with any kernel function. This method enjoys a number of key advantages, such as generating efficient and compact codes, fast indexing and search speed, and preserving diverse types of similarities (feature similarity and semantic similarity like label consistency).

Most state-of-the-art hashing methods only utilize a single feature type, whereas for many domains, such as image retrieval, combining multiple features has been proved very helpful for learning. Accordingly, Liu et al. [2012a] proposed a Multiple Feature

Table III. Categorization of Semisupervised Hashing Methods

Semisupervised Hashing	**Linear Semisupervised Hashing**	Semisupervised Hashing (SSH)
		Semisupervised Discriminant Hashing (SSDH)
		Semisupervised Topology-Preserving Hashing (STPH)
	Nonlinear Semisupervised Hashing	LAbel-regularized Max-margin Partition (LAMP)
		Bootstrap-NSPLH

Kernel Hashing (MFKH) method that is compatible with general data types and diverse similarities indicated by different visual features.

Because unsupervised hashing methods do not require any labeled data, the parameters of those methods are typically easy to learn given a prespecified distance metric. However, for some domains, especially vision-related problems, similarity (or distance) between data points might not be easily defined using a simple metric. Meanwhile, a metric similarity learned from a dataset (or a domain) might not work well for another dataset and cannot preserve semantic similarity. In this case, label information including weakly labeled data, such as pairwise instance labels, is useful for hashing.

4.2.2. Semisupervised Hashing. In semisupervised hashing, both labeled data and unlabeled data are used to train hash models. Representative methods include Semisupervised Hashing (SSH) [Wang et al. 2010a, 2012], LAbel-regularized Max-margin Partition [Mu et al. 2010], Semisupervised Discriminant Hashing [Kim and Choi 2011], Bootstrap Sequential Projection Learning for Semisupervised Nonlinear Hashing [Wu et al. 2013], and Semisupervised Topology-Preserving Hashing [Zhang et al. 2014a]. Among these hashing methods, SSH is one of the most popular approaches.

Table III categorizes semisupervised hashing into linear hashing and nonlinear hashing, depending on whether a linear or a nonlinear function is used in the hashing.

Linear Semisupervised Hashing. Due to easy availability of digital cameras and other imaging devices, large-scale image datasets are becoming rapidly available, raising immediate needs of image search from large data repositories. Unfortunately, fast and accurate image retrieval from large databases remains a significant challenge. On one hand, unsupervised hashing methods cannot effectively capture semantic similarity in image search, because no labeled data are provided to help hash functions learn semantic similarities. Although supervised hashing methods can utilize labeled information to learn semantic similarity, they tend to overfit when the number of labeled data is very small or labels are noisy. On the other hand, the training efficiency of supervised methods is rather inefficient. Motivated by these problems, SSH methods were proposed to handle both metric and semantic similarity over labeled and unlabeled data.

In order to handle large-scale image search and solve the constraints of unsupervised hashing in metric and the low efficiency of training in supervised hashing, Wang et al. [2010a, 2012] proposed SSH methods that use simple linear mapping to handle both metric and semantic similarity and dissimilarity in the data. Such SSH methods have been mainly used for large-scale image search.

SSH aims to map n data points $X = [x_1, x_2, \ldots,] \in \mathrm{R}^D$ to a Hamming space and seeks a K-bit Hamming embedding of X given by $\mathrm{y} \in \{1, -1\}^{n*K}$. Given a vector $w_k \in \mathrm{R}^D$ and $W = [w_1, \ldots, w_k, \ldots, w_K] \in \mathrm{R}^{D*K}$, the k^{th} hash function is defined as

$$\hbar_k(x_i) = \mathrm{sign}({w_k}^T x_i). \tag{10}$$

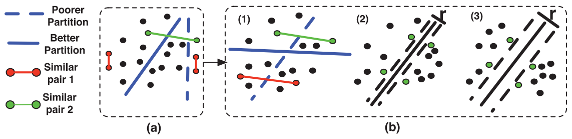


Fig. 9. (a) Comparison between good partition and bad partition: each data point represents an instance. (b) The main idea of the Weakly-Supervised LAbel-regularized Max-margin Partition (LAMP) algorithm: each point represents a data item. Black dotted lines in (2) and (3) represent the partition of data fields. γ denotes the partition margin.

In order to accommodate data labels, SSH allows two categories of label information. Assume $\mathcal{M}$ denotes a neighbor pair and C denotes a nonneighbor pair, and $H = [h_1, \ldots, h_k, \ldots, h_K]$ is a sequence of K hash functions. SSH defines the objective function $J(\cdot)$ to measure the empirical accuracy on the labeled data for H:

$$J(H) = \sum_k \left\{ \sum_{(x_i,x_j)\in\mathcal{M}} \hbar_k(x_i)\,\hbar_k(x_j) - \sum_{(x_i,x_j)\in C} \hbar_k(x_i)\,\hbar_k(x_j) \right\}. \tag{11}$$

Then, SSH defines a matrix $\mathbf{S} \in \mathrm{R}^{L*L}$ to incorporate pairwise labeled information X_l and express $J(H)$:

$$\mathbf{S}_{i,j} = \begin{cases} 1 & : (\mathrm{x_i}, \mathrm{x_j}) \in \mathcal{M} \\ -1 & : (x_i, x_j) \in C \\ 0 & : \text{otherwise} \end{cases}. \tag{12}$$

Suppose $H(X_l) \in \mathrm{R}^{K*L}$ maps data points in X_l to K-bit codes; SSH represents $J(H)$ as

$$J(H) = \frac{1}{2}\,\mathrm{tr}\{H(X_l)\,\mathbf{S}\,H(X_l)^T\} \Rightarrow J(W) = \frac{1}{2}\,\mathrm{tr}\left\{\mathrm{sign}\left(W^T X_l\right)\,\mathbf{S}\,\mathrm{sign}(W^T X_l)^T\right\}. \tag{13}$$

The aim of SSH is to learn optimal projections W that give the same bits for $(x_i, x_j) \in \mathcal{M}$ and different bits for $(x_i, x_j) \in C$:

$$\arg\max_W \frac{1}{2}\ \mathrm{tr}(W^T M W),$$

where $M = X_l\,\mathbf{S}\,X_l^T + \eta X X^T$, $X_l\,\mathbf{S}\,X_l^T$ represents the supervised term, and ηXX^T represents the unsupervised term (η is a positive scalar, which relatively weights the variance-based regularization term). The M is an adjusted covariance matrix. In M, the supervised term aims to greatly reduce the empirical error on the labeled data, and the unsupervised term tries to maximize variance and independence of individual bits in order to provide effective regularization. Furthermore, by relaxing the orthogonality constraints, both orthogonal and nonorthogonal solutions are applied to generate better hash codes at no added computational cost.

Given a large-scale unlabeled dataset with a few pairwise labeled data points, the final goal of SSH is to learn data-dependent hash functions with compact storage and fast retrieval, as well as a better partition of data points. Figure 9(a) demonstrates the difference between good and bad partitioning, where a better partition allows similar pairs to be on the same side and dissimilar pairs to be partitioned at different sides.

Although SSH is efficient and achieves a very simple eigen-decomposition-based solution, it does not consider separability between short binary codes when learning a compact binary code from a set of training data. In order to solve this problem, Kim and Choi [2011] learned discriminative binary codes based on semisupervised hashing on linear discriminant analysis and proposed semisupervised discriminant hashing (SSDH). SSDH uses labeled data to maximize the separability between binary codes in different classes with unlabeled data being used for regularization. Experiments and validations show that for short binary codes, SSDH indeed outperforms SSH.

For the aforementioned methods, hash functions are learned without considering ranking information so their hash codes inherently ignore ranking information, which is important for similarity search, especially for k nearest neighbor kNN search. Moreover, some distance-preserving hashing methods cannot well preserve data topology. Accordingly, Zhang et al. [2014a] proposed a Semisupervised Topology-Preserving Hashing (STPH) method to solve the aforementioned two problems by incorporating neighborhood ranking information with hash function learning. In addition, STPH also leverages semantic labels in training data, so its hash results can accurately search semantic neighbors.

Nonlinear Semisupervised Hashing. For linear semisupervised hashing, existing approaches mainly rely on linear feature representation. In reality, kernel-based feature representation may be more efficient in gauging similarity between data items, especially for visual objects. Accordingly, Mu et al. [2010] developed the Weakly-Supervised LAbel-regularized Max-margin Partition (LAMP) algorithm in kernel space to support kernel-based feature representation. LAMP is specially designed for kernel space and can generate high-quality hash functions with kernel tricks and weak supervision. The random sampling strategy in LAMP makes this method scalable for large-scale datasets. The main motivations and idea of LAMP are illustrated in Figure 9(b).

In Figure 9(b), (1) illustrates the side information that is more reasonable for guiding a hashing scheme, and (2) and (3) show two different hash functions that result in a different margin γ on the same distribution. Figure 9(b) shows that (1) side information or label information can provide useful guidance for more reasonable hashing results; (2) larger margin γ potentially implies a lower error rate in similarity search and can lead to better generalization ability, and (3) for vision applications, if kernel-based feature representation were used, it would be more natural and useful to measure similarity. Similar to KLSH, LAMP can also be applied to any image databases because it does not assume data distributions in input data.

Another nonlinear semisupervised hashing is Bootstrap projection for semisupervised hashing. For SSH, one limitation is that the underlying relationship between data points may not be effectively reflected because of the linear projection accompanied with mean thresholding. Meanwhile, for high-dimensional data points, SSH tends to require high computational costs. To address this limitation, a Semisupervised Nonlinear Hashing (Bootstrap-NSPLH) [Wu et al. 2013] was proposed, by using bootstrap sequential projection learning. In Bootstrap-NSPLH, a nonlinear hash function is employed for reflecting the underlying link between data points that can be used in semisupervised hashing. In addition, Bootstrap-NSPLH can correct the errors accumulated during hashing by holistically considering previous learned bits. In Bootstrap-NSPLH, compared with linear hash functions, the number of dimensions in the matrix for computation is much smaller and does not relate to the number of dimensions in the original data space.

4.2.3. Supervised Hashing. In supervised hashing, labeled data, such as labels of each image or pairwise constraints specifying similar/dissimilar data item pairs, are available to help learn hashing models. The goal of supervised hashing is to utilize label-based similarity or semantic similarity, in addition to the feature values of the data, to train effective hash functions.

Table IV. Summary of Supervised Hashing Methods

Supervised Hashing	**Linear Supervised Hashing**	Supervised Hashing with Binary Reduction
		Boosting Similarity-Sensitive Coding (BoostSSC)
		Binary Reconstructive Embedding (BRE)
		Minimal Loss Hashing (MLH)
		Latent Factor Hashing (LFH)
		Linear Discriminant Analysis-Based Hashing (LDAHash)
	Nonlinear Supervised Hashing	Kernel-Based Supervised Hashing (KSH)
		Two-Step Hashing (TSH)
		FastHash

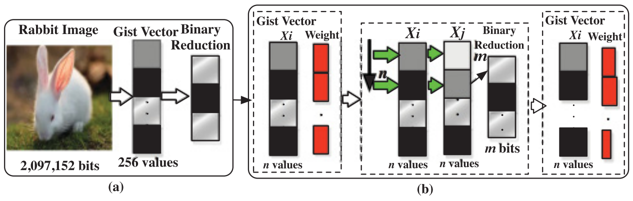


Fig. 10. (a) Supervised hashing with binary reduction: the three colors "black," "white," and "gray" represent different values in the gist vector and binary reduction. (b) The BoostSSC process: The three colors (black, white, and gray) represent different values in the gist vector; the width of the red denotes the corresponding weight value of the gist vector.

Representative supervised hashing methods include Boosting Similarity-Sensitive Coding [Shakhnarovich et al. 2003], Boltzmann machine-based hashing [Salakhutdinov and Hinton 2007], Binary Reconstructive Embedding [Kulis and Darrell 2009], Minimal Loss Hashing [Norouzi and Blei 2011], Kernel-based Supervised Hashing [Liu et al. 2012b], and Linear Discriminant Analysis-based Hashing [Strecha et al. 2012]. Most recently, some new supervised methods have been proposed, including Similarity-Preserving Hashing [Breitinger and Baier 2012], Two-Step Hashing [Lin et al. 2013], Multimodal Similarity-Preserving Hashing [Masci et al. 2014], Semantic Correlation Maximization [Zhang and Li 2014], Latent Factor Hashing [Zhang et al. 2014b], and FastHash [Lin et al. 2014].

According to the actual form of hashing functions used in each method, we also divide supervised hashing into linear hashing and nonlinear hashing. The category and corresponding important hashing algorithms are listed in Table IV.

Linear Supervised Hashing. In order to save data storage space, binary reduction technology is often used in hashing. Figure 10(a) demonstrates the binary reduction.

Figure 10(a) demonstrates that using binary reduction results in significant storage reduction. In the figure, the gist is defined as an abstract representation of the scene that spontaneously activates memory representation of scene categories [Oliva and Torralba 2001]. The original image has 2,097,152 bits. By using the gist vector, the image is converted to a gist vector with 256 values (i.e., represented using only 16 bits).

As a boosting method, labeled images (positive and negative image pairs) are used to train the discovery of the binary reduction. In order to learn a series of weighted hashing functions from labeled data, Shakhnarovich et al. [2003] proposed a Boosting Similarity Sensitive Coding (BoostSSC) technique, which uses a weighted Hamming distance to compute distances between images and learns an embedding of the original input space into a new space, as shown in Figure 10(b). During the beginning of BoostSSC, weights are uniform for each value (the red cells in Figure 10(b)) in gist vector of an image). For each bit in binary reduction (hashing function $H(X)$), BoostSSC chooses the index that causes a least weighted error across the training set and then updates the weights for the next calculation. At last, each bit in binary reduction has a corresponding index and a weight threshold (in the far right dotted box in Figure 10(b)).

Deep neural network has also been used to learn compact binary codes from high-dimensional inputs, by using stacked Restricted Boltzmann Machines (RBMs) [Salakhutdinov and Hinton 2007]. In order to explicitly preserve input distances after mapping to the Hamming space, Kulis and Darrell [2009] developed an efficient coordinate-descent algorithm, Binary Reconstructive Embedding (BRE), by minimizing a squared loss over the error between the input distances and the reconstructed Hamming distances. BRE is a supervised algorithm for learning hash functions for fast and accurate nearest neighbor search.

In BRE, data-dependent bit-correlated hashing is defined as follows:

$$\hbar_p(x) = \text{sign}\left(\sum_{q=1}^{s} \mathbf{W}_{pq} k(x_{pq},\ x)\right). \tag{14}$$

In Equation (14), $\{x_{pq},\ q = 1, \ldots, s\}$ is the training data for learning $\hbar_p$, $k(\cdot)$ is a kernel function, and $\mathbf{W}$ is a weight matrix. Hash functions aim to explicitly preserve the input distances when mapping to the Hamming space by minimizing the reconstruction error between original distances and reconstructed Hamming distances. The original Euclidean distance $d_{\mathcal{M}}$ and the reconstructed Hamming distance $d_{\mathcal{R}}$ are defined as

$$d_{\mathcal{M}}(x_i,\ x_j) = \frac{1}{2}\|x_i - x_j\|^2; \quad d_{\mathcal{R}}(x_i,\ x_j) = \frac{1}{K}\sum_{k=1}^{K}(\hbar_k(x_i) - \hbar_k(x_j))^2. \tag{15}$$

The goal is to derive optimal $\mathbf{W}$ by minimizing the following reconstruction error:

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \sum_{(i,\ j)\in\mathcal{N}} [d_{\mathcal{M}}(x_i,\ x_j) - d_{\mathcal{R}}(x_i,\ x_j)]^2 \ , \tag{16}$$

where the set of sample pairs $\mathcal{N}$ represents the training data that can be chosen based on the application. Because of the nondifferentiability of sign(.), it is difficult to optimize the previous objective function. Accordingly, BRE iteratively updates the hash functions to a local optimum by applying a coordinate-descent algorithm.

By setting a zero distance for each same-label pair, and a sufficiently large distance for each different-label pair, it is much easier to extend BRE to a supervised scenario.

For large-scale datasets, the training of BRE is inefficient, and it is almost impractical for BRE to train on a large-scale dataset because of expensive storage cost. In order to address this limitation, Norouzi and Blei [2011] proposed Minimal Loss Hashing (MLH) based on the latent structural SVM framework under a general class of loss functions that is suitable for training using Euclidean distance or using sets of labeled data points. In order to further improve the training efficiency, Latent Factor Hashing (LFH) was proposed in Zhang et al. [2014b] to learn similarity-preserving binary codes. In order to train LFH on large-scale datasets, the authors employed a stochastic linear-time variant learning approach. Meanwhile, for large-scale datasets, there exists retrieval and matching problems. In order to store and retrieve descriptor data, Strecha et al. [2012] proposed Linear Discriminant Analysis-based Hashing (LDAHash) to map descriptor vectors to Hamming space and reduce the size of the descriptors by representing them as short binary strings.

Nonlinear Supervised Hashing. Representative nonlinear supervised hashing methods include Kernel-based Supervised Hashing (KSH) [Liu et al. 2012b], Two-Step

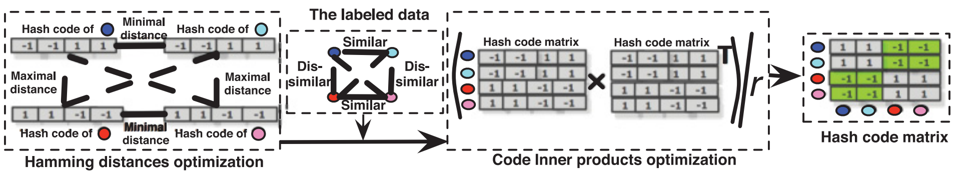


Fig. 11. The core process of Kernel-based Supervised Hashing (KSH): the different colors of circles represent different data. The green grids in the “Hash code matrix” represent adjusted parts in the hash code.

Hashing (TSH) [Lin et al. 2013], and FastHash [Lin et al. 2014]. Among these methods, KSH and TSH both use a kernel function as the nonlinear hash function, and FastHash uses boosted decision trees to achieve nonlinearity in hashing.

Currently, hashing methods are mainly used to solve large-scale data problems. By leveraging supervised information into hash function learning, the hashing quality could be improved. In order to further improve the hashing performance and solve the lengthy model training problem, Liu et al. [2012b] proposed a KSH model. When mapping data to compact binary codes, this model minimizes the Hamming distances between similar data pairs and maximizes the Hamming distances between dissimilar data pairs. While optimizing the code inner products and adjusting the Hamming distances, in order to obtain short and discriminative codes, the hash functions are sequentially trained 1 bit at a time by using equivalence. Figure 11 demonstrates the main process of KSH.

In Figure 11, labeled data have similar pairs and dissimilar pairs, respectively. Each data point has a corresponding hash code. KSH first optimizes the Hamming distances by trying to minimize the Hamming distance between similar pairs and maximize the Hamming distance between dissimilar pairs (see the dotted box named “Hamming distance optimization”), and then combines these hash codes as a matrix. By using matrix multiplication (see the dotted box named “Code Inner products optimization”), KSH further adjusts the Hamming distances by the final hash code matrix in the dotted box named “Hash code matrix.”

In practice, the optimization process of KSH is coupled to a specified hash function, which restricts the application range of the optimization. Taking into consideration accommodating different loss functions and hash functions, a new Two-Step Hashing (TSH) [Lin et al. 2013] proposed a flexible and simple framework, which is built on the fact that hash function learning processes and the code generation may be seen as separated steps, and that the former can be achieved by training standard binary classifiers. As a result, hashing learning is divided into two stages: hash bit learning and hash function learning with the help of the learned bits.

The two-step hashing approach has also been applied to large-scale datasets with high-dimensional features, where training and testing costs for kernel hash functions are extremely expensive. In order to solve this issue, FastHash [Lin et al. 2014] first uses decision trees as nonlinear hash functions to deal with large-scale training and testing data with high dimensionality. A two-step learning strategy is further applied where the binary code inference and the simple binary classification training are combined to form the learning process.

Due to the powerful generalization capability, nonlinear hash functions normally outperform linear hash functions in terms of the overall performance.

In summary, compared to unsupervised hashing methods, the main advantages of supervised hashing methods are the flexibility and adaptability for different real-world applications. Nevertheless, the training efficiency still remains a big challenge.

Table V. Summary of Cryptographic Hashing Methods

Cryptographic Hashing	**Keyed Cryptographic Hashing**	VMAC; UMAC; PMAC; OMAC; HMAC
		Poly1305-AES; MD6; BLAKE2
	Unkeyed Cryptographic Hashing	MD2/4/5/6
		SHA-1/3/224/256/384/512
		HAVAL; GOST; FSB; JH; ECOH
		RIPEMD/-128/-160/-320

4.3. Cryptographically Secure Hashing

For cryptographically secure hashing (or cryptographic hashing), the hash function not only compresses an arbitrary length input into a fixed length output but also is designed to be one-way and has three major properties: preimage resistance, second preimage resistance, and collision resistance. These attributes can guarantee that (1) the input data is difficult to be generated by a given hash value, and (2) it is difficult to find the two different inputs with the same hash value.

With the increasing popularity of authentication and digital verification in applications, such as digital signature and public key cryptography, cryptographic hashing has gained significant attention recently. For example, in digital signature, cryptographic hashing can be used to generate a digest for a message, and the encrypted digest, using a secret key, can be regarded as the digital signature of the message.

Depending on whether a secret key is used by the hashing function, the cryptographic hashing can be divided into two categories: unkeyed cryptographic hashing and keyed cryptographic hashing.

4.3.1. Unkeyed Cryptographic Hashing. Unkeyed cryptographic hashing refers to methods whose hash process does not require a secure key to be used to provably guarantee the hash security, and the hash function is particularly designed for hashing. Although no secret key is involved in the unkeyed cryptographic hashing, it still meets the basic security policies required by general cryptographic hashing.

Many unkeyed cryptographic hash methods exist. Among them, MD5 and SHA are relatively more popular in applications. In the following, we briefly introduce the history of MD-family and then review other relevant methods.

"MD" (the abbreviation for "Message Digest") represents a series of hash functions designed by R. Rivest of RSA Data Security Inc. Among the MD-family, MD1 is a proprietary method, and MD2 [Kaliski 1992] was designed to generate a simple 16-byte message digest for an arbitrary length of message, with two claimed properties: (1) given a message digest, the difficulty of finding the message is in the order of 2^{128} operations and (2) given the same message digest, the difficulty of finding two different messages is in the order of 2^{64} operations. However, several works have revealed some weaknesses of MD2, such as the collisions from MD2's hash function [Rogier and Chauvaud 1997]. As a result, in 2004, MD2 was defined as an insecure one-way hash function. MD4 [Rivest 1992] was proposed by Ronald Rivest in 1992, with a 128-byte digest message in length. The claims for the difficulty of finding collisions are the same as MD2's, but an attack on the last two rounds of MD4 was found in Den Boer and Bosselaers [1991]. In order to strengthen MD4, MD5 [Rivest 1992] was proposed to add one extra round. Even though MD5 is one of the most promising hashing methods and there is not yet any successful attack on the full MD5, the current big data environment requires a much faster hashing method. MD6 [Rivest et al. 2008] was proposed to better meet hash demand where the length of a message digest can be flexibly set from 1 to 512 bits. MD6 has good efficiency and can support multiple processors.

Another important unkeyed cryptographic hashing is SHA [FIPS 1995], which is an updated version of MD4 with two major differences: (1) the length of the message digest

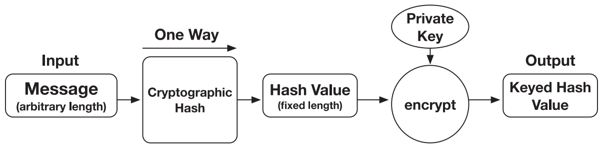


Fig. 12. Keyed cryptographic hashing scheme: One-way cryptographic hash function processes arbitrary-length input message and produces a fixed-length hash value to be combined with a private key to generate keyed hash value.

is 160 bytes, and (2) the step number in each round is 20 steps, which is longer than the 16 steps in MD4. HAVAL [Zheng et al. 1992] is also a popular unkeyed cryptographic hashing similar to MD5, but it is much more flexible because the hash length can include multiple byte options (128 or 256) and the number of rounds can be specified by users.

Many other unkeyed cryptographic hash functions, such as BLAKE-family, RIPEMD-family, ECOH [Brown et al. 2008], FSB-family [Augot et al. 2005], and GOST [Mendel et al. 2008], also exist for security-oriented applications.

4.3.2. Keyed Cryptographic Hashing. Although unkeyed cryptographic hashing, like MD5 or SHA, is very popular, it is, nevertheless, not provably secure. In keyed cryptographic hashing, a secret key is used to enhance the security as shown in Figure 12. In order to protect the information authenticity, most commonly, a form of Message Authentication Code (MAC) is employed by the hashing mechanism to compute a value called authentication tag, which can be regarded as the secret key and appended to the message. Without knowledge of this secret key, it is difficult to retrieve the message even though the hash function is known. The key is often shared between parties, so this kind of cryptography scheme can only avoid outsider attacks. Many applications, such as message authentication, password checking, and encryption, need the technical guarantee from keyed cryptographic hashing.

There are different types of MACs, such as message authentication code based on universal hashing (UMAC) [Black et al. 1999], block-cipher-based message authentication code algorithm using a universal hash (VMAC), One-key MAC (OMAC) [Iwata and Kurosawa 2003], Parallelizable MAC (PMAC), and keyed-hash message authentication code (HMAC) [Krawczyk et al. 1997]. All of these MACs enjoy provable cryptographic strength. Among of these, UMAC and VMAC are both based on the universal hashing, with UMAC being designed for 32-bit architectures and VMAC supporting both 32-bit and 64-bit architectures. Compared with other MACs, UMAC is much more computationally efficient. VMAC is specially designed for 64-bit CPU architectures, which can achieve an exceptional performance. PMAC makes use of a block cipher to create an efficient message authentication code, which is provably reducible in security to the underlying block cipher. OMAC is similar to PMAC in its functionality, with its message authentication code constructed from a block cipher. HMAC is a specific type of MAC, which considers not only a secret cryptographic key but also a cryptographic hash function. Furthermore, as with any MAC, HMAC can simultaneously verify both the data integrity and the authentication of a message. And any cryptographic hash function may be involved in the calculation of an HMAC. Many elements can be used to determine the cryptographic strength of HMAC, such as the output size, the underlying hash function, and the key's size and quality.

Our previous descriptions mainly focused on MAC-family hashing methods. Some other keyed cryptographic hash functions, such as Poly1305-AES [Bernstein 2005], SipHash [Aumasson and Bernstein 2012], and BLAKE2 [Aumasson et al. 2013], also exist but cannot be addressed in detail due to page limitations.

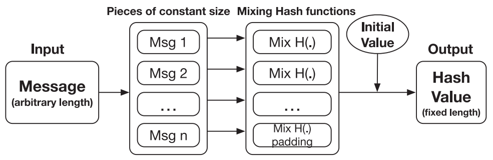


Fig. 13. General noncryptographic hashing scheme: An input message is divided into multiple pieces with a fixed length, with each piece being processed by a mixing hash function. Combining the initial value and the hash value of each piece generates a fixed size of hash value.

4.4. Cryptographically Insecure Hashing

Cryptographically insecure hashing, also referred to as noncryptographic hashing, shares a similar objective as cryptographic hashing, that is, taking a variable size of messages as input and returning a constant-size hash value smaller than the input message. However, the hashing process does not need to consider cryptography, and the basic goal is to reduce collision and improve the hashing speed as much as possible. For this kind of hash function, the secret information for the hashing operation is not required and the description of hash functions is known publicly.

For example, in security and forensic analysis, it is common to discover and find similar or homologous objects. While cryptographic hashing normally provides a yes/no or 1/0 answer to compare and find identical objects, it cannot be directly and efficiently applied to such applications that require a range of outcomes [0, 1], with the result being interpreted as a measure of similarity. Accordingly, cryptographically insecure hashing techniques, such as approximate matching [Breitinger et al. 2014], can be used to provide an approximate matching value (a measure of similarity) for finding similar or identical objects. Before the approximate matching method was proposed, there were several relevant approaches for different applications. The earlier one is Context-Triggered Piecewise Hashing (CTPH) [Kornblum 2006], which is used to identify homologous files (not identical files) by creating associations between files to help investigators quickly find relevant files or pieces of materials. CTPH advanced the traditional use of cryptographic hashing to computer forensics. For data fingerprinting, a similar idea with similarity digests was proposed in Roussev [2010] to generate data fingerprints. All these approximate matching approaches are more adaptable to general security and forensic analysis. For image identification applications, because of the fragility of the current hash methods to image processing operations, an efficient robust image hashing, leveraging the advantages of cryptographic hashing and image identification methods, was proposed [Steinebach et al. 2012] to support fast and stable image processing.

Compared to cryptographic hashing, noncryptographic hashing does not have secret information for the hashing operation. Methods in this category share four important features: (1) fast hashing speed, (2) low collision probability, (3) high probability of error detection, and (4) easy collision detection. As a result, they are popularly used in applications having big-size input data and requiring fast searches or processing, such as using MurmurHash3 hashing (a noncryptographic hash function) for feature hashing in Twitter sentiment analysis [Silva et al. 2014].

For almost all noncryptographic hash functions, the basic scheme is the same, which is called Merkle-Damgård construction as shown in the Figure 13, where the input message is divided into multiple pieces with each piece being processed by a corresponding mixing function. At last, all processed pieces are combined together to generate hash output.

Many noncryptographic hash functions have been proposed, and in this survey we focus on the most common ones. The Fowler-Noll-Vo (FNV) hash function [Fowler

Table VI. Noncryptographic Hashing Categorization

Noncryptographic Hashing	FNV Hash; xxHash; SuperFastHash
	MurmurHash2; lookup3; Pearson Hashing; BuzHash
	Approximate Matching
	DEK; BKDR; APartow; DJBX33A

1991] is an early noncryptographic hashing method proposed for a fast hash table and checksum. The most obvious advantage of the FNV is the simplicity of implementation. Another one worthy of mentioning is lookup3,[1] which mainly produces 32-bit hashes for hash table lookup. Lookup3 is one of the most important noncryptographic hashing methods and has been used in many products, such as Oracle and Google. Inspired by the ideas in the FNV and lookup3, SuperFastHash [Hsieh 2004] was proposed to achieve extremely high speed and provide an avalanche effect for the software industry. For the Open Source projects, another hashing method, MurmurHash2 [Appleby 2008], was very popular for a short period of time because of its extraordinary avalanche property. For hashing strings, DJBX33A was designed and popularly used in many programming languages and application servers, such as Python, PHP 5, and Tomcat. For general uses, BuzHash was designed by Robert Uzgalis in 1992. In BuzHash, a substitution table is used to replace each input piece by a randomized alias, and this design can fit almost all input distributions. In addition, in the "General Hash Function Library" [Partow 2013], there are three popular noncryptographic hash functions, the DEK [Knuth 1998], BKDR [Ritchie et al. 1988], and APartow [Partow 2013]. Among these three, the multiplicative hashing DEK is regarded as one of the earliest and simplest hashing methods and is still popular now. Most recently, a new hash function called xxHash[2] is very popular because of its extremely fast processing speed, running at RAM limits. xxHash can support both of 32 and 64 bits, and has been commonly used in databases and the gaming industry.

For noncryptographic hashing, its quality criteria include collision resistance, distribution of outputs, avalanche effect, and speed. For collision resistance, noncryptographic hashing tries to reduce the collisions, but because they do not follow the three security properties used by cryptographic hashing, collisions are still observable. For the distribution of outputs, it is very important for noncryptographic hashing to keep hash outputs in a uniform distribution because the uneven distribution may result in clustering issues and affect the performance of hash functions. For the avalanche effect, it will be very helpful to avoid clustering issues if a big change in the output can be observed for a very tiny change in the input. For the speed, noncryptographic hashing should perform as fast as possible to improve the hashing efficiency. According to the experimental analysis in Estébanez et al. [2014], MurmurHash2, SuperFastHash, and lookup3 have the best avalanche effect, and the lookup3, MurmurHash2, and SuperFastHash are more suitable for general uses.

4.5. Computational Complexity

We briefly review the computational complexity of hashing methods in terms of their computational efficiency and memory consumption. In our analysis, we define a Hamming space as the set of all 2^N binary strings of length N.

In summary, data-dependent hashing often uses training data to learn the hashing function with best compact codes for all data records. As a result, it can achieve much faster query time and less memory consumption than the data-independent hashing.

[1]http://burtleburtle.net/bob/c/lookup3.c.
[2]https://code.google.com/archive/p/xxhash/.

Table VII. Runtime Complexity *w.r.t.* the Number of Input Items

Linear Time $\mathcal{O}(n)$	Data-Independent Hashing: (1) Random Projection Hashing; (2) Universal Hashing; (3) LSH; (4) MinHash Data-Dependent Hashing: (1) Anchor Graph Hashing (AGH); (2) Locally Linear hashing (LLH); (3) Kernelized LSH (KLSH); (4) Semisupervised Hashing (SSH) Cryptographically Insecure Hashing: Most Common Methods
Linearithmic Time $\mathcal{O}(n \log n)$	Data-Independent Hashing: Binary Reconstructive Embedding (BRE)
Polynomial Time $\mathcal{O}(n^x)$	Data-Dependent Hashing: Spectral Hashing
Exponential Time $\mathcal{O}(x^n)$	Cryptographically Secure Hashing: Almost All Methods Are at Least in This Level

Table VIII. The Summary of Space Complexity

Constant Space	Data-Independent Hashing: (1) Random Projection Hashing
Linear Space	Data-Independent Hashing: (1) Universal Hashing; (2) LSH; (3) MinHash Cryptographically Secure Hashing: Almost All Cryptographically Insecure Hashing: Almost All (Generally Smaller Than the Secure Hashing)
Hamming Space	Data-Dependent Hashing: Almost All Methods Are in This Level

However, because the data-dependent hashing needs extra training time, the overall time performance difference cannot be arbitrarily determined. Security-oriented hashing is mainly more concerned about the security properties and has longer hashing codes than the data-oriented hashing codes. Therefore, security-oriented hashing methods are often more computationally expensive and are less efficient than data-oriented hashing methods. Compared with cryptographically secure hashing, cryptographically insecure hashing does not have to consider cryptography and only needs to reduce collision, which improves the hashing speed as much as possible. So cryptographically insecure hashing is often much more efficient than cryptographically secure hashing.

In Tables VII and VIII, we summarize the runtime complexity and space consumption of methods in different categories.

(1) For data-independent hashing, random projection hashing is a computationally efficient method, which randomly projects high-dimensional (d) data into a lower-dimensional (b) subspace. For n input items, the space consumption is $d \times \lg b$ bits, and the time complexity is linear $O(dbn)$. An improved hashing method, universal hashing, randomly chooses some hashing functions from a particular function set (not from all functions), so its space complexity is $n \times \lg d$ and the time complexity is $O(n)$. For another commonly known hashing LSH, assume the hash table size is k, the width parameter of a function is w, the space is $O(nk)$, and the processing time complexity is $O(nkw)$. The next hashing method worthy to mention is MinHash, which requires $O(nk)$ time and $O(n)$ bits for each single permutation. In summary, most data-independent hashing methods require linear time complexity.

(2) For data-dependent hashing, its time complexity can be divided into two parts: (1) training complexity and (2) codeword calculation complexity. For spectral hashing,

its training time is $O(n^3)$ and codeword calculation time is $O(rn)$, where r is the dimensionality of the Hamming space. Comparably, AGH, a popular linear unsupervised hashing, can build a resulting graph in $O(n)$ time. Another similar kind of hashing, LLH, uses a subset of size m as sample training data. The training time complexity is $O(m^2)$ and the coding time complexity is $O(m)$, with space complexity $O(m)$. For a nonlinear unsupervised hashing, KLSH, the training time complexity for building the matrix is $O(p^3)$, and coding computational complexity is $O(p^2)$ for each hash function, where the p is the sample size of database objects. When $p = O(\sqrt{n})$, KLSH approaches to sublinear search time. SSH relaxes the orthogonality constraints of PCA and has been experimentally verified that it has the same running time as LSH. For supervised hashing, each iteration needs to update the hash function, and the time complexity of the BRE is $O(nb(k + logn))$ (where k represents the number of nearest neighbors of each input data and b represents the bit size of a hash table). In summary, while some data-dependent hashing has linear time complexity, a handful of methods do require polynomial-time complexity.

(3) Cryptographically secure hashing must meet preimage resistance, second preimage resistance, and collision resistance. For a message with size n, the computational complexity of the hash value is $O(n)$. In order to be preimage resistant, 2^{n-1} messages have to be created and the complexity is $O(2^n)$. Meanwhile, for being second preimage resistant, the time complexity is also $O(2^n)$. For being collision resistant, because of the randomness, if an attacker wants to find two messages with the same hash value, almost $2^{n/2}$ hashing operations need to be performed. Therefore, in summary, the overall complexity of cryptographically secure hashing is $O(2^n)$.

(4) For cryptographically insecure hashing, the hash functions mainly aim for a fast hash table and checksum use. Ideally, regardless of data size, it is possible for cryptographically insecure hashing to search data within a constant time O(1) [Estébanez et al. 2014]. Especially for some extremely fast hashing, such as xxHash, the speed is almost close to the RAM limits. For other common noncryptographic hash functions, as shown in the experimental studies in Estébanez et al. [2014], the speed greatly depends on the size of the hashed keys. SuperFastHash and MurmurHash2 perform the best for long keys. Similarly with the cryptographically insecure hashing, the space complexity also depends on the output size of each message. But differently, the output size in cryptographically insecure hashing is generally shorter than that in cryptographically secure hashing, which emphasizes the security properties.

5. APPLICATIONS OF HASHING METHODS

5.1. Data-Oriented Applications

From the data domain perspective, hashing has been widely applied to a variety of applications, such as images, network, graphs and texts, and digital signature.

Because of the popularity of image data and the large volumes and high dimensionality involved in these applications, majority hashing methods are designed for the image domain. The main target of image application is the image retrieval [Kulis et al. 2009; Poullot et al. 2007; Jegou et al. 2008; Kulis and Grauman 2009; Xu et al. 2011b; Kong et al. 2012; Wang et al. 2010a; Fu et al. 2013; Korman and Avidan 2011; Yu et al. 2013]. When searching images related to a query image, it is computationally expensive and time-consuming to directly compare the similarity between images in the original feature space. In order to achieve fast similarity calculation, many hashing methods are proposed. Among these methods, Kulis et al. [2009] constructed random hash functions for fast approximate similarity search with learned metric, and Kulis and Grauman [2009] generalized locality-sensitive hashing to accommodate the arbitrary kernel function, especially for image retrieval tasks. Wang et al. [2010a] designed a semisupervised hashing method to learn efficient hash codes to handle metric and semantic similarity among images. Another method [Fu et al. 2013] proposed to use boosting iterative quantization hashing with query-adaptive reranking for large-scale image retrieval. Yu et al. [2013] used unsupervised PCA hashing for large-scale medical image search, which is a unique way to find similar clinical cases for doctors. In other image applications, Wang et al. [2006] designed an AnnoSearch to annotate images, and Chum et al. [2008] and Li et al. [2013] used MinHash and Spectral hashing, respectively, for fast image indexing.

On the Internet, hashing technologies have been used to solve many challenges. For example, IP address lookup is time sensitive and its speed largely affects the overall network performance. Motivated by the hashing idea, Martinez et al. [2005], Martinez and Lin [2006], and Martinez et al. [2009] proposed different hashing methods for IP address lookup in computer networks. More specifically, Martinez et al. [2005] proposed an optimal XOR hashing to facilitate a linearly distributed address lookup. In order to further adapt to different distributions, Martinez and Lin [2006] proposed an adaptive hashing for all practical databases, especially for the nonuniformly distributed IP address lookup. Martinez et al. [2009] proposed a preprocessing method to extract certain regularity based on XOR operations.

For other network-related applications, Lu et al. [2006] designed a hardware-friendly scheme for minimal perfect hashing, which can be used for route lookups, packet classification, and monitoring. Choi et al. [2009] and Yoshioka et al. [2008], in order to improve the packet classification in network intrusion detection, respectively used maximum entropy hashing and rule hashing.

Graph-structured data are common in biology, chemistry, health informatics, and communication networks. For all these applications, supporting efficient access to graph-structured data is crucial. Ou et al. [2013] applied a heterogeneous hashing for heterogeneous networks such as Facebook, Flickr, and Twitter. Weiss et al. [2009] and Liu et al. [2011], studied theoretical hashing algorithm design for graph databases and proposed spectral hashing and anchor graph hashing, respectively, for graph partitioning. Meanwhile, hashing techniques can also be used to support fast graph classification. In Li et al. [2012], we introduced a NSH for structured data and also proposed a fast graph stream classification method using DIscriminative Clique Hashing (DICH) [Chi et al. 2013].

For text applications, Chi et al. [2014] and Xu et al. [2011a] used context-preserving hashing, which is based on Recursive Min-wise Hashing (RMH) and multidimensional progressive perfect hashing, for fast text classification and string matching. Jin and Yoo [2009], Mu et al. [2012], Zhu et al. [2013a, 2013b], and Song [2015] designed different hashing methods for efficient large-scale multimedia search, and Yue et al. [2011, 2013] used hashing-based fast palm-print identification for large-scale palm-print databases.

5.2. Security-Oriented Applications

Security-oriented hashing is commonly used in information security-related applications, such as the verification of the integrity and source of data, digital signatures [Kaur and Kaur 2012], data authentication, message authentication codes [Black Jr 2000], and encryption.

(1) Digital signature: Keyed cryptographic hash functions are commonly used in digital signature [Halevi and Krawczyk 2006], where the signature is a cryptographic value generated from the message and a secret key, which is only known to the signer. For the message recipients, they only need to make sure that the message is truly from the sender by using a public key for verification, so the private key is used for message signing and the public key is only for message verification. During the signing process, the private key works with the hash value (from cryptographic hashing) of the

signature algorithm to generate digital signature, which is appended to the message and sent to the recipients. During the verification process, the recipients verify the signature using the public key to obtain an output. Meanwhile, the recipients can hash the received data using the same hash function used by the sender to obtain a hash value. By comparing this hash value and the output from the verification algorithm, the recipient can determine whether the digital signature originates from the sender. Such a digital signature framework can be extended to data authentication and verification of data integrity, such as image authentication [Schneider and Chang 1996].

(2) Message authentication codes: In security applications, a forgery user may masquerade as another user to send a message, and message authentication codes are used to prevent this kind of attack, where the message authentication codes can only be created by the original sender. When multiple parties need to communicate, keyed cryptographic hash functions can produce a shared secret key for all involved parties for message authentication, which would verify the originality of data and avoid outsider attacks. The UMAC [Black et al. 1999], VMAC, OMAC [Iwata and Kurosawa 2003], PMAC, and HMAC [Krawczyk et al. 1997] are all specially designed for fast and secure message authentication.

6. CONCLUSION

In this survey, we categorized existing hashing techniques as a hierarchical taxonomy with two major groups, data-oriented hashing versus security-oriented hashing. The former aims to speed up the data access by developing efficient hashing mechanisms, and the latter focuses on using hashing to generate message digests (or signatures) for verification. Data-oriented hashing includes two subgroups: data-independent hashing and data-dependent hashing. For data-independent hashing, hashing functions are defined independently of the data to be processed without involving a training process from the data. Data-dependent hashing, on the other hand, defines and learns the hashing function family with respect to a given training dataset. For security-oriented hashing, we categorized its methods into cryptographically secure hashing and cryptographically insecure hashing. We further summarized computational complexity and applications of hashing methods in each group. Our survey reviewed the uniqueness and methodologies of mainstream hashing techniques in each category and summarized major domain applications involving hashing. The combined review, from technique and application perspectives, provided practical reference for real-world implementations, as well as in-depth research guidance for future development.

ACKNOWLEDGMENTS

This research is partially sponsored by the Program for Professor of Special Appointment (Eastern Scholar) at Shanghai Institutions of Higher Learning.

REFERENCES

Austin Appleby. 2008. Murmurhash 2.0.

Vassilis Athitsos and Stan Sclaroff. 2003. Estimating 3D hand pose from a cluttered image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Vol. 2. II–432–9.

Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. 2005. A family of fast syndrome based cryptographic hash functions. In *Proceedings of the International Conference on Cryptology in Malaysia*. Springer, 64–83.

Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. SipHash: A fast short-input PRF. In *Proceedings of the International Conference on Cryptology in India*. Springer, 489–508.

Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-OHearn, and Christian Winnerlein. 2013. BLAKE2: Simpler, smaller, fast as MD5. In *Proceedings of the International Conference on Applied Cryptography & Net Security*. Springer, 119–135.

Daniel J. Bernstein. 2005. The poly1305-AES message-authentication code. In *Proceedings of the International Workshop on Fast Software Encryption*. Springer, 32–49.

Krishna Bharat and Andrei Broder. 1998. A technique for measuring the relative size and overlap of public web search engines. *Computer Networks and ISDN Systems* 30, 1 (1998), 379–388.

John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. 1999. UMAC: Fast and secure message authentication. In *Annual Intl. Cryptology Conf*. Springer, 216–233.

John R. Black Jr. 2000. *Message Authentication Codes*. Ph.D. Dissertation. University of California Davis.

Zalán Bodó and Lehel Csató. 2014. Linear spectral hashing. *Neurocomputing* 141 (2014), 117–123.

Jean Bourgain. 1985. On Lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics* 52, 1–2 (1985), 46–52.

Jonathan Brandt. 2010. Transform coding for fast approximate nearest neighbor search in high dimensions. In *Proceedings of the 2010 IEEE Conf. Computer Vision and Pattern Recognition (CVPR'10)*. IEEE, 1815–1822.

Frank Breitinger and Harald Baier. 2012. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *Proceedings of the International Conference on Digital Forensics and Cyber Crime*. Springer, 167–182.

Frank Breitinger, Barbara Guttman, Michael McCarrin, Vassil Roussev, and Douglas White. 2014. Approximate matching: Definition and terminology. *NIST Special Publication* 800 (2014), 168.

Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences 1997*. 21–29.

Daniel R. L. Brown, Adrian Antipa, Matt Campagna, and Rene Struik. 2008. ECOH: The elliptic curve only hash. *Submission to NIST* (2008).

J. Lawrence Carter and Mark N. Wegman. 1977. Universal classes of hash functions. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*. ACM, 106–112.

Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*. ACM, 380–388.

Lianhua Chi, Bin Li, and Xingquan Zhu. 2013. *Fast Graph Stream Classification Using Discriminative Clique Hashing*. Springer, 225–236.

Lianhua Chi, Bin Li, and Xingquan Zhu. 2014. Context-preserving hashing for fast text classification. In *Proceedings of the 2014 SIAM International Conference on Data Mining (SDM'14)*. 100–108.

Lynn Choi, Hyogon Kim, Sunil Kim, and Moon Hae Kim. 2009. Scalable packet classification through rulebase partitioning using the maximum entropy hashing. *IEEE/ACM Transactions on Networking (TON)* 17, 6 (2009), 1926–1935.

Ondrej Chum, James Philbin, and Andrew Zisserman. 2008. Near duplicate image detection: Min-hash and tf-idf weighting. In *BMVC*, Vol. 810. 812–815.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th Annual Symposium on Computational Geometry*. 253–262.

Bert Den Boer and Antoon Bosselaers. 1991. An attack on the last two rounds of MD4. In *Annual Intl. Cryptology Conf*. Springer, 194–203.

E. Knuth Donald. 1999. The art of computer programming. *Sorting and Searching* 3 (1999), 426–458.

César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. 2014. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience* 44, 6 (2014), 681–698.

Christos Faloutsos and King-Ip Lin. 1995. *FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets*. Vol. 24. ACM.

Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica* 4, 1 (1974), 1–9.

PUB FIPS. 1995. 180-1. secure hash standard. *National Institute of Standards and Tech* 17 (1995), 45.

G. Fowler. 1991. Fowler/Noll/Vo (FNV) hash. Retrieved from http://isthe. com/chongo/tech/comp/fnv.

Haiyan Fu, Xiangwei Kong, and Jiayin Lu. 2013. Large-scale image retrieval based on boosting iterative quantization hashing with query-adaptive reranking. *Neurocomputing* 122 (2013), 480–489.

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity search in high dimensions via hashing. In *VLDB*, Vol. 99. 518–529.

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

Yunchao Gong and Svetlana Lazebnik. 2011. Iterative quantization: A procrustean approach to learning binary codes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'11)*. 817–824.

Shai Halevi and Hugo Krawczyk. 2006. *Strengthening Digital Signatures via Randomized Hashing*. Springer, 41–59.

Junfeng He, Wei Liu, and Shih-Fu Chang. 2010. Scalable similarity search with optimized kernel hashing. In *Proceedings of the 16th SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1129–1138.

Kaiming He, Fang Wen, and Jian Sun. 2013. K-means hashing: An affinity-preserving quantization method for learning binary compact codes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2938–2945.

Jae-Pil Heo, Youngwoon Lee, Junfeng He, Shih-Fu Chang, and Sung-Eui Yoon. 2012. Spherical hashing. In *Proceedings of the 2012 IEEE Conference Computer Vision and Pattern Recognition (CVPR'12)*. IEEE, 2957–2964.

Gisli R. Hjaltason and Hanan Samet. 2003. Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25, 5 (2003), 530–549.

Paul Hsieh. 2004. Hash functions.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*. 604–613.

Sergey Ioffe. 2010. Improved consistent sampling, weighted minhash and l1 sketching. In *Proceedings of the 2010 IEEE International Conference on Data Mining*. IEEE, 246–255.

Go Irie, Zhenguo Li, Xiao-Ming Wu, and Shih-Fu Chang. 2014. Locally linear hashing for extracting non-linear manifolds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2115–2122.

Tetsu Iwata and Kaoru Kurosawa. 2003. Omac: One-key cbc mac. In *Intl. Workshop on Fast Software Encryption*. Springer, 129–153.

H. V. Jagadish. 1997. Analysis of the hilbert curve for representing two-dimensional space. *Information Processing Letters* 62, 1 (1997), 17–22.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2008. *Hamming Embedding and Weak Geometric Consistency for Large Scale Image Search*. Springer, 304–317.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.

Jianqiu Ji, Jianmin Li, Shuicheng Yan, Bo Zhang, and Qi Tian. 2012. Super-bit locality-sensitive hashing. In *Advances in Neural Information Processing Systems*. 108–116.

Minho Jin and Chang Dong Yoo. 2009. Quantum hashing for multimedia. *IEEE Transactions on Information Forensics and Security* 4, 4 (2009), 982–994.

Alexis Joly and Olivier Buisson. 2011. Random maximum margin hashing. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'11)*. IEEE, 873–880.

Alexis Joly, Carl Frélicot, and Olivier Buisson. 2004. Feature statistical retrieval applied to content based copy identification. In *Proceedings of the International Conference on Image Processing*, Vol. 1. IEEE, 681–684.

Burton Kaliski. 1992. *The MD2 Message-Digest Algorithm*. Technical Report.

Yoonseop Kang, Saehoon Kim, and Seungjin Choi. 2012. Deep learning to hash with multiple representations. In *ICDM*. 930–935.

Ravneet Kaur and Amandeep Kaur. 2012. Digital signature. In *Proceedings of the 2012 International Conference on Computing Sciences (ICCS'12)*. IEEE, 295–301.

Saehoon Kim and Seungjin Choi. 2011. Semi-supervised discriminant hashing. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining (ICDM'11)*. IEEE, 1122–1127.

Donald Ervin Knuth. 1998. *The Art of Computer Programming: Sorting and Searching*. Vol. 3. Pearson Education.

Weihao Kong and Wu-Jun Li. 2012. Isotropic hashing. In *Advances in Neural Information Processing Systems*. 1646–1654.

Weihao Kong, Wu-Jun Li, and Minyi Guo. 2012. Manhattan hashing for large-scale image retrieval. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 45–54.

Vanja Kontak, Sinisa Srbljic, and Dejan Skvorc. 2012. Hashing scheme for space-efficient detection and localization of changes in large data sets. In *Proceedings of the 35th International Convention*. 1496–1501.

Simon Korman and Shai Avidan. 2011. Coherency sensitive hashing. In *Proceedings of the 2011 IEEE International Conference on Computer Vision (ICCV'11)*. IEEE, 1607–1614.

Jesse Kornblum. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation* 3 (2006), 91–97.

Hugo Krawczyk, Ran Canetti, and Mihir Bellare. 1997. HMAC: Keyed-hashing for message authentication. *Informational* (1997).

Brian Kulis and Trevor Darrell. 2009. Learning to hash with binary reconstructive embeddings. In *Advances in Neural Information Processing Systems*. 1042–1050.

Brian Kulis and Kristen Grauman. 2009. Kernelized locality-sensitive hashing for scalable image search. In *Proceedings of the 2009 IEEE 12th International Conference on Computer Vision*. IEEE, 2130–2137.

Brian Kulis, Prateek Jain, and Kristen Grauman. 2009. Fast similarity search for learned metrics. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 12 (2009), 2143–2157.

Bin Li, Xingquan Zhu, Lianhua Chi, and Chengqi Zhang. 2012. Nested subtree hash kernels for large-scale graph classification over streams. In *Proceedings of the IEEE International Conference on Data Mining*. 399–408.

Ping Li, Arnd Konig, and Wenhao Gui. 2010. B-bit minwise hashing for estimating three-way similarities. In *Advances in Neural Information Processing Systems*. 1387–1395.

Ping Li and Arnd Christian Konig. 2011. Theory and applications of b-bit minwise hashing. *Communications of the ACM* 54, 8 (2011), 101–109.

Ping Li and Christian Konig. 2010. B-bit minwise hashing. In *Proceedings of the 19th International Conference on World Wide Web*. ACM, 671–680.

Ping Li, Anshumali Shrivastava, Joshua L. Moore, and Arnd C. Konig. 2011. Hashing algorithms for large-scale learning. In *Advances in Neural Information Processing Systems*. 2672–2680.

Peng Li, Meng Wang, Jian Cheng, Changsheng Xu, and Hanqing Lu. 2013. Spectral hashing with semantically consistent graph for image indexing. *IEEE Transactions on Multimedia* 15, 1 (2013), 141–152.

Guosheng Lin, Chunhua Shen, Qinfeng Shi, Anton van den Hengel, and David Suter. 2014. Fast supervised hashing with decision trees for high-dimensional data. In *Proceedings of the IEEE Conference on CVPR*. 1963–1970.

Guosheng Lin, Chunhua Shen, David Suter, and Anton van den Hengel. 2013. A general two-step approach to learning-based hashing. In *Proceedings of the 2013 IEEE International Conference on Computer Vision (ICCV'13)*. IEEE, 2552–2559.

Yue Lin, Rong Jin, Deng Cai, Shuicheng Yan, and Xuelong Li. 2013. Compressed hashing. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'13)*. IEEE, 446–451.

Wei Liu, Jun Wang, Rongrong Ji, Yu-Gang Jiang, and Shih-Fu Chang. 2012b. Supervised hashing with kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12)*. 2074–2081.

Wei Liu, Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2011. Hashing with graphs. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*. 1–8.

Xianglong Liu, Junfeng He, Di Liu, and Bo Lang. 2012a. Compact kernel hashing with multiple features. In *Proc. of the 20th ACM Intl. Conf. on Multimedia*. ACM, 881–884.

Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 2006. Perfect hashing for network applications. In *2006 IEEE Intl. Symp. on Information Theory*. IEEE, 2774–2778.

Hans Peter Luhn. 1953. A new method of recording and searching information. *American Documentation* 4, 1 (1953), 14–16.

Mark Manasse, Frank McSherry, and Kunal Talwar. 2010. Consistent weighted sampling. Unpublished Technical Report. Retrieved from http://research.microsoft.com/en-us/people/manasse.

Christopher Martinez and Wei-Ming Lin. 2006. Adaptive hashing for IP address lookup in computer networks. In *Proceedings of the 14th IEEE International Conference on Networks, 2006 (ICON'06)*, Vol. 1. IEEE, 1–6.

Christopher J. Martinez, Wei-Ming Lin, and Parimal Patel. 2005. Optimal XOR hashing for a linearly distributed address lookup in computer networks. In *Proceedings of the ACM/IEEE ANCS Symposium*. 203–210.

Christopher J. Martinez, Devang K. Pandya, and Wei-Ming Lin. 2009. On designing fast nonuniformly distributed ip address lookup hashing algorithms. *IEEE/ACM Transactions on Networking* 17, 6 (2009).

Jonathan Masci, Michael M. Bronstein, Alexander M. Bronstein, and Jürgen Schmidhuber. 2014. Multimodal similarity-preserving hashing. *IEEE Transactions on Pattern Analalysis and Machine Intelligence* 36, 4 (2014), 824–830.

Florian Mendel, Norbert Pramstaller, Christian Rechberger, Marcin Kontak, and Janusz Szmidt. 2008. Cryptanalysis of the GOST hash function. In *Proceedings of the Annual International Cryptology Conference*. Springer, 162–178.

Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. 1996. *Handbook of Applied Cryptography*. CRC Press.

Sean Moran, Victor Lavrenko, and Miles Osborne. 2013a. Neighbourhood preserving quantisation for lsh. In *Proceedings of the 36th ACM SIGIR Conference on Research and Development in Information Retrieval*. 1009–1012.

Sean Moran, Victor Lavrenko, and Miles Osborne. 2013b. Variable bit quantisation for LSH. In *ACL (2)*. 753–758.

Robert Morris. 1968. Scatter storage techniques. *Communications of the ACM* 11, 1 (1968), 38–44.

Yadong Mu, Xiangyu Chen, Xianglong Liu, and et al. 2012. Multimedia semantics-aware query-adaptive hashing with bits reconfigurability. *International Journal of Multimedia Information Retrieval* 1, 1 (2012), 59–70.

Yadong Mu, Jialie Shen, and Shuicheng Yan. 2010. Weakly-supervised hashing in kernel space. In *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'10)*. IEEE, 3344–3351.

Mohammad Norouzi and David M. Blei. 2011. Minimal loss hashing for compact binary codes. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*. 353–360.

Aude Oliva and Antonio Torralba. 2001. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision* 42, 3 (2001), 145–175.

Mingdong Ou, Peng Cui, Fei Wang, Jun Wang, Wenwu Zhu, and Shiqiang Yang. 2013. Comparing apples to oranges: A scalable solution with heterogeneous hashing. In *Proceedings of the ACM SIGKDD Conference*. 230–238.

Arash Partow. 2013. General purpose hash function algorithms. Retrieved from http://www.partow.net/programming/hashfunctions.

W. Wesley Peterson. 1957. Addressing for random-access storage. *IBM Journal of Research and Development* 1, 2 (1957), 130–146.

Sébastien Poullot, Olivier Buisson, and Michel Crucianu. 2007. Z-grid-based probabilistic retrieval for scaling up content-based copy detection. In *Proceedings of the 6th ACM Conference on Image and Video Retrieval*. 348–355.

Maxim Raginsky and Svetlana Lazebnik. 2009. Locality-sensitive binary codes from shift-invariant kernels. In *Advances in Neural Information Processing Systems*. 1509–1517.

Mohammad Rastegari, Jonghyun Choi, Shobeir Fakhraei, Daume Hal, and Larry Davis. 2013. Predictable dual-view hashing. In *Proceedings of the 30th International Conference on Machine Learning*. 1328–1336.

Dennis M. Ritchie, Brian W. Kernighan, and Michael E. Lesk. 1988. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ.

Ronald Rivest. 1992. *The MD4 Message-Digest Algorithm, RFC 1320*. MIT and RSA Data Security, Inc (1992).

Ronald L. Rivest, Benjamin Agre, Daniel V. Bailey, Christopher Crutchfield, Yevgeniy Dodis, Kermin Elliottet Fleming, Asif Khan, Jayant Krishnamurthy, Yuncheng Lin, and Leo Reyzin. 2008. The MD6 hash function–a proposal to NIST for SHA-3. *Submission to NIST* 2 (2008), 3.

N. Rogier and Pascal Chauvaud. 1997. MD2 is not secure without the checksum byte. *Designs, Codes and Cryptography* 12, 3 (1997), 245–251.

Vassil Roussev. 2010. Data fingerprinting with similarity digests. In *Proceedings of the IFIP International Conference on Digital Forensics*. Springer, 207–226.

Caitlin Sadowski and Greg Levin. 2007. Simhash: Hash-based similarity detection. www.googlecode.com/sun/trunk/paper/SimHashwithBib.pdf (2007).

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

Ruslan Salakhutdinov and Geoffrey E Hinton. 2007. Learning a nonlinear embedding by preserving class neighbourhood structure. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*. 412–419.

Robert E. Schapire and Yoram Singer. 1999. Improved boosting algorithms using confidence-rated predictions. *Machine Learning* 37, 3 (1999), 297–336.

Marc Schneider and Shih-Fu Chang. 1996. A robust content based digital signature for image authentication. In *Proceedings of the International Conference on Image Processing, 1996*, Vol. 3. IEEE, 227–230.

Gregory Shakhnarovich. 2005. *Learning Task-Specific Similarity*. Thesis.

Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. 2006. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice.* The MIT Press (2006).

Gregory Shakhnarovich, Paul Viola, and Trevor Darrell. 2003. Fast pose estimation with parameter-sensitive hashing. In *Proceedings of the 9th International Conference on Computer Vision*. 750–757.

Fumin Shen, Chunhua Shen, Qinfeng Shi, Anton Van Den Hengel, and Zhenmin Tang. 2013. Inductive hashing on manifolds. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition*. 1562–1569.

Anshumali Shrivastava. 2016. Exact weighted minwise hashing in constant time. *arXiv Preprint arXiv:1602.08393* (2016).

Anshumali Shrivastava and Ping Li. 2014a. Densifying one permutation hashing via rotation for fast near neighbor search. In *ICML*. 557–565.

Anshumali Shrivastava and Ping Li. 2014b. Improved densification of one permutation hashing. *arXiv Preprint arXiv:1406.4784* (2014).

Anshumali Shrivastava and Ping Li. 2014c. In defense of minhash over simhash. In *AISTATS*. 886–894.

Alan Siegel. 2004. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing* 33, 3 (2004), 505–543.

Nádia F. F. Silva, Eduardo R. Hruschka, and Estevam Rafael Hruschka Jr. 2014. Biocom usp: Tweet sentiment analysis with adaptive boosting ensemble. *SemEval 2014* (2014), 123.

Jingkuan Song. 2015. *Effective Hashing for Searching Large-scale Multimedia Databases*. Thesis.

Martin Steinebach, Huajian Liu, and York Yannikos. 2012. Forbild: Efficient robust image hashing. In *Proceedings of the SPIE Conference on Media Watermarking, Security and Forensics*, Vol. 8303.

Christoph Strecha, Alexander M. Bronstein, Michael M. Bronstein, and Pascal Fua. 2012. LDAHash: Improved matching with smaller descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34, 1 (2012), 66–78.

Antonio Torralba, Robert Fergus, and Yair Weiss. 2008. Small codes and large image databases for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2008 (CVPR'08)*. IEEE, 1–8.

Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2010a. Semi-supervised hashing for scalable image retrieval. In *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'10)*. IEEE, 3424–3431.

Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2010b. Sequential projection learning for hashing with compact codes. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 1127–1134.

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

Xiong Wang, Jason T. L. Wang, King-Ip Lin, Dennis Shasha, Bruce A. Shapiro, and Kaizhong Zhang. 2000. An index structure for data mining and clustering. *Knowledge and Information Systems* 2, 2 (2000), 161–184.

X.-J. Wang, Lei Zhang, Feng Jing, and Wei-Ying Ma. 2006. Annosearch: Image auto-annotation by search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Vol. 2. 1483–1490.

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

Mark N. Wegman and J. Lawrence Carter. 1981. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences* 22, 3 (1981), 265–279.

Yair Weiss, Rob Fergus, and Antonio Torralba. 2012. *Multidimensional Spectral Hashing*. Springer, 340–353.

Yair Weiss, Antonio Torralba, and Rob Fergus. 2009. Spectral hashing. In *Advances in Neural Information Processing Systems*. 1753–1760.

Chenxia Wu, Jianke Zhu, Deng Cai, Chun Chen, and Jiajun Bu. 2013. Semi-supervised nonlinear hashing using bootstrap sequential projection learning. *IEEE Transactions on Knowledge and Data Engineering* 25, 6 (2013), 1380–1393.

Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. 2011b. Complementary hashing for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision (ICCV'11)*. 1631–1638.

Yang Xu, Lei Ma, Zhaobo Liu, and H Jonathan Chao. 2011a. A multi-dimensional progressive perfect hashing for high-speed string matching. In *Prof. of ACM/IEEE ANCS Symp*. 167–177.

Zhao Xu, Kristian Kersting, and Christian Bauckhage. 2012. Efficient learning for hashing proportional data. In *2012 IEEE 12th International Conference on Data Mining (ICDM'12)*. 735–744.

Atsushi Yoshioka, Shariful Hasan Shaikot, and Min Sik Kim. 2008. Rule hashing for efficient packet classification in network intrusion detection. In *Proc. of Computer Communications and Networks*. 1–6.

Xiang Yu, Shaoting Zhang, Bo Liu, Lin Zhong, and Dimitris N Metaxas. 2013. Large scale medical image search via unsupervised PCA hashing. In *Proceedings of IEEEE CVPR Workshops*. 393–398.

Feng Yue, Bin Li, Ming Yu, and JiaQiang Wang. 2011. Fast palmprint identification using orientation pattern hashing. In *Proceedings of the 2011 International Conference on Hand-Based Biometrics (ICHB'11)*. IEEE, 1–6.

Feng Yue, Bin Li, Ming Yu, and Jiaqiang Wang. 2013. Hashing based fast palmprint identification for large-scale databases. *IEEE Transactions on Information Forensics and Security* 8, 5 (2013), 769–778.

Dongqing Zhang and Wu-Jun Li. 2014. Large-scale supervised multimodal hashing with semantic correlation maximization. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Dell Zhang, Jun Wang, Deng Cai, and Jinsong Lu. 2010a. *Laplacian Co-Hashing of Terms and Documents*. Springer, 577–580.

Dell Zhang, Jun Wang, Deng Cai, and Jinsong Lu. 2010b. Self-taught hashing for fast similarity search. In *Proceedings of the 33rd ACM SIGIR Conference.* 18–25.

Lei Zhang, Yongdong Zhang, Xiaoguang Gu, Jinhui Tang, and Qi Tian. 2014a. Scalable similarity search with topology preserving hashing. *IEEE Transactions on Image Processing* 23, 7 (2014), 3025–3039.

Lei Zhang, Yongdong Zhang, Dongming Zhang, and Qi Tian. 2013. *Distribution-Aware Locality Sensitive Hashing*. Springer, 395–406.

Peichao Zhang, Wei Zhang, Wu-Jun Li, and Minyi Guo. 2014b. Supervised hashing with latent factor models. In *Proceedings of the 37th ACM SIGIR Conf*. ACM, 173–182.

Yi Zhen and Dit-Yan Yeung. 2012. A probabilistic model for multimodal hash function learning. In *Proceedings of the 18th SIGKDD Conf. on Knowledge Discovery and Data Mining*. ACM, 940–948.

Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry. 1992. HAVAL: A one-way hashing algorithm with variable length of output. In *Proceedings of the International Workshop on the Theory and Application of Cryptographic Technology*. Springer, 81–104.

Xiaofeng Zhu, Zi Huang, Hong Cheng, Jiangtao Cui, and Heng Tao Shen. 2013a. Sparse hashing for fast multimedia search. *ACM Transactions on Information Systems (TOIS)* 31, 2 (2013), 9.

Xiaofeng Zhu, Zi Huang, Heng Tao Shen, and Xin Zhao. 2013b. Linear cross-modal hashing for efficient multimedia search. In *Proceedings of the 21st ACM Intl. Conf. on Multimedia*. ACM, 143–152.

Received March 2016; revised December 2016; accepted January 2017