Multimed Tools Appl
DOI 10.1007/s11042-015-3217-x

CrossMark

Accelerated Manhattan hashing via bit-remapping with location information

Wenshuo Chen[1] · Guiguang Ding[1] · Zijia Lin[2] · Jisheng Pei[2]

Received: 20 May 2015 / Revised: 2 December 2015 / Accepted: 29 December 2015
© Springer Science+Business Media New York 2016

Abstract Hashing is a binary-code encoding method which tries to preserve the neighborhood structures in the original feature space, in order to realize efficient approximate nearest neighbor search in large-scale databases. Existing hashing methods usually adopt a two-stage strategy (projection stage and quantization stage) to encode data points, and threshold-based single-bit quantization (SBQ) is used to binarize each projected dimension into 0 or 1. Data similarity between hash codes is measured by their Hamming distance. However, SBQ may destroy the original neighborhood structures by quantizing neighboring points near threshold into different binary values. Double-bit quantization (DBQ) and its derivative, Manhattan hashing, have been proposed to fix this problem. Experimental results showed that Manhattan hashing outperformed state-of-the-art methods in terms of effectiveness, but lost the advantage of efficiency because it used decimal arithmetic instead of fast bitwise operations for similarity measurement between hash codes. In this paper, we propose an accelerated strategy of Manhattan hashing by making full use of bitwise operations. Our main contributions are: 1) a new encoding method which assigns location information to each binary digit is proposed to avoid the time-consuming decimal arithmetic; 2) a novel hash code distance measurement that accelerates the calculation of Manhattan distance is proposed to improve query efficiency. Extensive experiments on three benchmark datasets

✉ Guiguang Ding
dinggg@tsinghua.edu.cn

Wenshuo Chen
cws13@mails.tsinghua.edu.cn

Zijia Lin
linzijia07@tsinghua.org.cn

Jisheng Pei
pjs07@mails.tsinghua.edu.cn

[1] School of Software, Tsinghua University, Beijing, 100084, China

[2] Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China

Published online: 16 January 2016

show that our approach improves the speed of data querying on 2-bit, 3-bit and 4-bit quantized hash codes by at least one order of magnitude on average, without any precision loss.

Keywords Accelerated Manhattan hashing · Bit-remapping · Multiple-bit quantization · Manhattan distance

1 Introduction

In recent years, there has been an increasing growth of a large variety of data on the Internet, such as image [17], video [25], and text [28], which has brought great challenges for machine learning and related fields such as information retrieval and data mining. This has led much attention to the research on similarity search, particularly nearest neighbor (NN) search, in massive databases [3, 5, 8, 26]. However, the traditional brute-force NN search strategy is too time-consuming and space-consuming to be used in real-world applications. Thus researchers proposed to use hashing techniques for efficient approximate nearest neighbor (ANN) search, which sacrifice query accuracy moderately in exchange for dramatic boost of query speed. Hashing methods are to learn binary-code representations of data points with the principle of preserving neighborhood structures in the original feature space. To be more specific, the original feature points will be encoded into compactly-stored binary strings, and the mapped binary strings of similar points in the original feature space should be close to each other in the Hamming space. Furthermore, hashing methods store binary-code data compactly with hardware bits, requiring much less RAM when performing ANN, and calculate Hamming distances between data points using fast bitwise XOR and bit-count[1] operations which drastically boost the query speed.

Analysis on existing hashing methods indicates that it is an NP hard problem to directly compute the best binary codes for a given dataset [29]. Hence, nearly all previous hashing methods adopt a two-stage strategy to encode data points, i.e. a projection stage and a quantization stage. In the former stage, data points are projected from the original n-dimensional feature space to d-dimensional space. In the latter stage, the projected vectors are quantized into binary strings with length of c. Currently, the majority of the existing hashing methods use threshold-based single-bit quantization (SBQ) to binarize each projected dimension into 0 or 1, as illustrated in Fig. 1a. Although the SBQ strategy is widely adopted, the inevitable quantization loss introduced by the selection of threshold will violate the principle of hashing which is data similarity-preserving. In practice, threshold of a certain projected dimension is usually set as the mean value or the median value of the projected values on this dimension. However, statistics show that both kinds of threshold values generally lie in the region of the highest point density, as revealed in [14], and thus a large number of neighboring points near a threshold may be mapped into different bits (such as point p_2 in region B and point p_3 in region C in Fig. 1), which is surely a breach of the intention to preserve the neighborhood structures of the original feature space.

As far as we know, hierarchical quantization [19] is the first proposed non-SBQ quantization method. The authors found that, there is always a possibility that neighboring points close to the threshold are hashed to different bits (e.g. p_2 and p_3 in Fig. 1). To correct this

[1]The method bit-count(n) counts the number of '1' bits in the binary representation of n, which is also known as the calculation of n's Hamming weight.

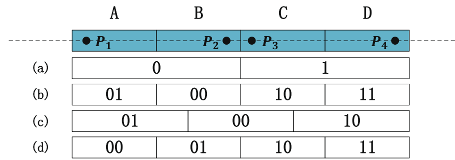

Fig. 1 Different encoding methods of four quantization strategies: **a** single-bit quantization (SBQ); **b** hierarchical quantization; **c** double-bit quantization (DBQ); **d** 2-bit Manhattan hashing

error, hierarchical quantization divides each dimension into four regions and uses two bits to encode each region as illustrated in Fig. 1b (which by coincidence is the same as our proposed 2-bit quantization encoding method). It is true that hierarchical quantization narrows the distances between point pairs like p_2 and p_3 (namely $d_h(p_2, p_3)$) to 1, but it also results in $d_h(p_1, p_4) = 1$ due to the use of Hamming distance. Furthermore, we can get $d_h(p_1, p_3) = d_h(p_2, p_4) = 2$ which is unreasonable because $d_h(p_1, p_4) < d_h(p_1, p_3)$ in this case, so the destruction of neighborhood structure of data is still not well handled.

Recent researches suggest that the adoption of double-bit quantization (DBQ) [14] strategy is a feasible solution to tackle the weak points of SBQ and hierarchical quantization. The basic idea of DBQ is to quantize each projected dimension into double bits with adaptively learned thresholds that divide the real-valued axis into three regions. It is proposed to preserve the neighboring structures by omitting the ‘11’ code for encoding, as shown in Fig. 1c, and Hamming distance is used as distance measurement. Experiments in [14] showed that DBQ outperforms SBQ significantly.

Manhattan hashing [15] is an updated version of DBQ. It uses Manhattan distance to replace Hamming distance for the calculation of data similarities in the Hamming space, which leads to further improvement in query precision. As shown in Fig. 1d, 2-bit Manhattan quantization divides each dimension into four sections, encoding each section with the binary representation of its index. As a consequence, it must turn the 2-bit quantized binary codes into decimal values first to calculate the Manhattan distances. The value scope of 2-bit Manhattan distance is {0, 1, 2, 3}, while the range of Hamming distance is {0, 1, 2} in which value 3 cannot be covered. Furthermore, 2-bit quantization can be easily expanded to 3-bit and 4-bit [15], which leads to an even wider scope of distances and further improvement in query precision. Hence, Manhattan distance is more powerful than Hamming distance when it comes to the preservation of the original neighborhood structures. However, Manhattan distance measurement loses the advantage of efficiency which is the highlight of using hashing methods to do ANN search, for decimal operations are much slower than bitwise operations.

Motivated by the demand to speed up Manhattan hashing, we propose a new encoding method for quantization and an accelerated Manhattan distance measurement. The whole solution is referred as accelerated Manhattan hashing (AMH). First, essential changes in terms of code mapping are made to encode the projected values in a way that is different from the original Manhattan hashing. To be more specific, some location information is assigned to each digit of the q-bit quantized binary codes and a hierarchical mapping method is used rather than directly encoding each cluster with its index. We refer to this process as “bit-remapping”. The word “remapping” is used here to indicate that the accelerated Manhattan hashing uses the same set of codes with the original Manhattan hashing while adopting a different bijection. Next, with the help of our bit-remapping strategy, we

propose a divide-and-conquer type of distance measure function which reduces the query time drastically by making full use of bitwise operations instead of decimal arithmetic. Our "bit-remapping" method and the distance measure function guarantee that we get exactly the same query result as Manhattan hashing at much faster speed, because we calculate the Manhattan distance in a different and improved way that will accelerate the calculation. Thus, accelerated Manhattan hashing tackles the original Manhattan distance efficiency problem without any loss of query precision.

The rest of the paper is organized as follows. In Section 2, we introduce the previous related work. Section 3 describes the new bit-remapping encoding method. Then the accelerated distance measure algorithm is introduced in Section 4. Section 5 reports the experimental results. Finally, we conclude the whole paper in Section 6.

2 Previous work

Thanks to the unremitting effort put by researchers in this field in the past few decades, existing hashing methods have already shown extraordinary query speed, powerful capability of storage space compression and promising performance in terms of query precision. As a result, various hashing methods [2, 4, 22, 25, 31–35] have been widely used in a variety of ANN applications, such as audio retrieval, content-based image retrieval (CBIR), and video retrieval.

As stated before, the general process of nearly all hashing methods is a two-stage strategy: projection stage and quantization stage. Figure 2 illustrates this encoding process.

2.1 Projection stage

The goal of hashing is to encode an n-dimensional data point with a binary string of c bits. To do that we first need to project the n-dimensional data point into d-dimensional. If SBQ

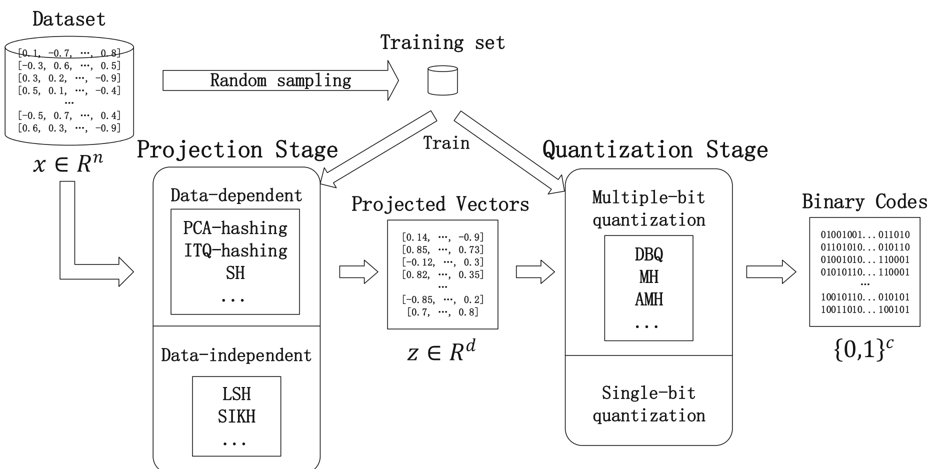

Fig. 2 Illustration of the two-stage general process of hashing methods. For data-dependent projection methods and multiple-bit quantization methods, randomly sampled training set is provided to compute relevant parameters and thresholds

is used, we'll have $d = c$. This process is denoted as the "projection stage" in hashing, as illustrated in Fig. 2.

Since most of the existing hashing methods focus on the projection stage, here we categorize them as "projection methods". Generally, these methods can be divided into two categories: data-independent and data-dependent. Locality-sensitive hashing (LSH) [1, 6, 9] and its extensions are typical data-independent methods. They simply adopt random projections that are independent of the training data to encode original data points. Shift-invariant kernel hashing [24] extends the original LSH by applying a shifted cosine function to generate hash values. Although the theory of data-independent methods is simple, the resulting hash codes need to be sufficiently long to preserve neighborhood structures in the original feature space. As a consequence, data-independent methods usually need longer hash codes to encode the data than data-dependent methods to achieve comparable performance, costing more storage space and query time.

Data-dependent methods, on the other hand, learn better hashing functions from training data through machine learning techniques, striving to enhance their ability of neighborhood structures preservation with as short hash codes as possible. PCA-Hash [7, 13] performs principal component analysis [30] on raw data and then quantizes the projected dimensions into binary hash codes. Spectral hashing (SH) [29] learns hash functions using spectral graph partitioning strategy, in which the graph is constructed with data points and their similarities. Iterative Quantization (ITQ) [7] works in an alternating minimization scheme for finding a rotation of zero-centered data so as to minimize the quantization loss of mapping the data to a binary hypercube. Both unsupervised data embeddings such as PCA and supervised embeddings such as canonical correlation analysis (CCA) can be used as its preliminary step to get the projected vectors. Experiments showed that ITQ achieves state-of-the-art ANN performance among hashing methods. There are still a lot of other researches on hashing like Semi-supervised hashing (SSH) [27], Minimal loss hashing (MLH) [23], Cross-view hashing [18], etc.

2.2 Quantization stage

Quantization stage is to binarize the projected d-dimensional value vector into binary string of c bits. SBQ quantizes each projected value into 0 or 1 by thresholding, so $c = d$ in this case. But if multi-bit quantization is used, each projected value will be quantized into a q-bit (q may be variable) binary code, and thus we'll have $c \geqslant d$.

Quantization stage is relatively less researched compared to the projection stage, but the authors of [15] found that these two stages are of equal importance. Besides the previously introduced Hierarchical hashing [19], DBQ [14] and Manhattan hashing [15], there has been some other quantization method proposed in recent years. Variable bit quantisation (VBQ) [21] provides a data-driven non-uniform bit allocation across hyperplanes. It optimally allocates a variable number of bits per LSH hyperplane because a subset of hyperplanes may be more informative than others. Neighbourhood preserving quantisation (NPQ) [20] is similar to VBQ except that it allocates equal number of bits to each LSH hyperplane. Quadra-Embedding hashing [16] adopts the same encoding method of Hierarchical hashing, i.e. assigning two bits for each projection to define four quantization regions. And to fix the problem of directly using Hamming distance in [19], the authors of [16] defined a novel binary code distance function tailored to their method which combines the use of bitwise operations and decimal operations. However, this method only works in 2-bit quantization.

As a scalable and flexible method, Manhattan hashing [15] achieves an outstanding performance in query precision among all hash-based quantization methods, but it loses the advantage of query efficiency. In this paper, we propose a whole quantization solution including bit-remapping and accelerated Manhattan distance measurement, in order to fix the efficiency problem of Manhattan hashing.

3 Encoding method for accelerated Manhattan hashing

Given a data point $x \in \mathcal{R}^n$, the goal of hashing is to learn a hash function to encode it into a binary string $\{0, 1\}^c$, namely a mapping that is from the original $n-$dimensional real-valued space to the $c-$dimensional Hamming space.

In the case of traditional single-bit quantization, hash functions are in need to generate an intermediate $c-$dimensional real-valued vector $\mathbf{z}$ for each point in the projection stage. Then in the quantization stage, the projected vector $\mathbf{z}$ is encoded into a binary string $\mathbf{y}$ by thresholding. For example, if the threshold of the kth projected dimension is θ, an encoding function $sgn(z_k)$ is adopted where $sgn(z_k) = 1$ if $z_k \geqslant \theta$ and 0 otherwise.

In the case of multi-bit quantization, new encoding methods will be needed to binarize each projected dimension. For example, q-bit Manhattan hashing [15] uses k-means clustering algorithm to cluster the real values of each projected dimension into 2^q clusters, and the midpoints of the line joining neighboring cluster centers will be used as thresholds. Then it uses a q-bit binary code to encode the index of each cluster. Figure 3a, c and e illustrate respectively the detailed encoding results of 2-bit, 3-bit and 4-bit Manhattan hashing.

In this paper, we aim to bridge the efficiency gap between the original Hamming distance and the Manhattan distance calculation. As mentioned above, q-bit Manhattan hashing encodes each projected real-valued dimension with q binary digits, so we need a d-dimensional real-valued projected vector to form a binary string of length c, where $c = d \times q$. The problem *w.r.t.* Manhattan distance calculation is that it needs to convert each q-bit binary code into a decimal number first, and then performs "+" operation $d - 1$ times and "−" operation d times to get the final result [15]. Evidently, d increases linearly with hash code length c, and so do the frequencies of binary-to-decimal conversion and "+/−" operations. For example, for 2-bit Manhattan hashing ($q = 2$) with hash code length of 256

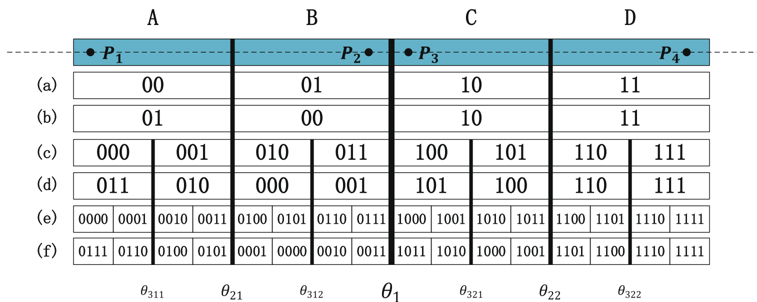

Fig. 3 Different encoding methods of Manhattan hashing and accelerated Manhattan hashing: **a** 2-bit Manhattan hashing; **b** 2-bit accelerated Manhattan hashing; **c** 3-bit Manhattan hashing; **d** 3-bit accelerated Manhattan hashing; **e** 4-bit Manhattan hashing; **f** 4-bit accelerated Manhattan hashing

($c = 256$, $d = 128$), Manhattan distance measurement ought to perform binary-to-decimal conversion 128 times first, then "+" operation 127 times and "−" operation 128 times. Compared with Hamming distance measurement in which only bitwise XOR is performed, Manhattan hashing loses the advantage of efficiency which is the core value of hashing methods, because it uses decimal operations too many times.

In order to tackle this problem, we consider replacing the time-consuming decimal arithmetic with bitwise operations when calculating Manhattan distance. First, we need to avoid the binary-to-decimal conversion which involves breaking the compactly stored hash codes into q-bit-length pieces. This problem is mainly caused by simply encoding each projected value using the binary representation of its k-means cluster's index as in Fig. 3a, c and e. Hence, we propose a new encoding method which encodes the projected values in a different way by bit-remapping. To be specific, we assign some location information (or hierarchical sign) to each single bit of the q-bit codes, so that all the ith bits of the projected values can be considered on the same encoding level as a whole. Second, Manhattan distance between the newly encoded hash codes must be calculated correctly using as few "+/−" operations as possible. With the help of our new encoding method, we propose an accelerated Manhattan distance measure function through a divide-and-conquer approach in which the number of decimal arithmetic is linear to the quantization factor q rather than the code length c. Generally, c is much greater than q, which makes our approach more efficient than the original Manhattan hashing. Take the example of 2-bit quantized Manhattan hashing with hash code length of 256 again, our proposed accelerated Manhattan hashing performs "+" one time and "−" one time (will be explained later in Section 4.1), which will solve the efficiency problem brought by Manhattan hashing while keeping the same distance measure results.

3.1 2-bit quantization

To simplify the explanation of our encoding process for general q-bit quantization, 2-bit quantization will be introduced first as an example, and then the q-bit encoding can be easily deduced from the simple 2-bit encoding. The encoding process of 2-bit quantization goes as follows:

1. Given the projected values (denoted as z_k) of a certain dimension, we divide their value scope into four sections as Fig. 3b by k-means like the original Manhattan hashing does, which is explained before.
2. K-means algorithm with 4 clusters ($k = 4$) will generate 3 thresholds (thresholds θ_1, θ_{21} and θ_{22} in Fig. 3) to divide the value scope. The reason we denote the threshold in the middle as θ_1 is that, it can be seen as the one which does the first division on the first Hierarchical level, similar as in Hierarchical hashing [19]. For $z_k < \theta_1$, namely the value lies in left of θ_1, the first bit of z_k is set as 0, and 1 otherwise.
3. The other two thresholds are denoted as θ_{21} and θ_{22}, because they both do the second division after θ_1 and they lie respectively on the left and right of θ_1. They can be seen as the second-layer thresholds. For $\theta_{21} \leqslant z_k < \theta_{22}$, namely the value lies between the two second-layer thresholds, the second bit of z_k is set as 0, and 1 otherwise.

Notice that it is the bit-remapping on the second layer that distinguishes our 2-bit encoding method from the original 2-bit Manhattan hashing, for we use location information to decide the second bit, while Manhattan hashing only uses the binary form of some decimal

value. Using our 2-bit encoding method, the second bit of each cluster in Fig. 3b is in turn encoded into 1, 0, 0 and 1, where clusters near θ_1 is set to be 0 and clusters far from θ_1 is set to be 1. The motivation of proposing this bit-remapping rule is to remove the decimal information from the q-bit code and replace it with location information (close to or far from θ_1), in order to realize the Manhattan distance calculation using fast bitwise operations on compactly stored binary codes without breaking them into q-bit length of pieces. By this encoding method, the region pairs A/D and B/C in Fig. 3 can be easily separated, which is the basis of our accelerated Manhattan distance measurement. Detailed explanations will be given later in Section 4.1.

3.2 q-bit quantization

As can be seen in Fig. 3d, the encoding rule of 3-bit quantization is an extension to the 2-bit quantization. The first bit and second bit of a 3-bit quantized code is encoded using the 2-bit encoding rule. The third-layer thresholds (θ_{311}, θ_{312}, θ_{321} and θ_{322} in Fig. 3) further divide each second-layer cluster into two third-layer clusters. By ignoring the first bit, encoding either side of θ_1 is reduced to the 2-bit quantization problem, so that the "1,0,0,1" mapping can be used to encode the third bit.

Hence, the encoding rule of q-bit quantization can be easily deduced from $(q-1)$-bit quantization. The ith bit ($i > 1$) of a q-bit quantized code represents whether the projected value z_k lies between two ith-layer thresholds (namely set as 0) or otherwise (namely set as 1). Following this encoding rule, the details about the bit-mapping result of 3-bit and 4-bit quantizations is shown in Fig. 3d and f. Formal description of the q-bit encoding process can be found in Algorithm 1.

As explained before, the original Manhattan hashing involves breaking the compactly stored binary strings into q-bit length of pieces and performing "$+/-$" operations on them. Mostly, the quantization factor q is set to be a small value ($2 \leq q \leq 4$), but code length c is usually large (128, 256 or more). Decimal arithmetic is unavoidable when calculating Manhattan distance, but the measure method [15] adopts causes the query time to climb with the growing code length c. However, in our encoding method each ith bit of all projected values tells some hierarchical information on the same level, so they can be considered as a whole rather than single binary digits which only make sense in the context of arithmetic representation. In this way, when calculating Manhattan distance based our bit-mapping rule, a binary string is broken into q pieces, each with the length of the projected dimension d ($c = d \times q$), which makes the query complexity linear to q rather than d in Manhattan hashing. This feature brings convenience and efficiency when calculating Manhattan distance, which will be illustrated later in Section 4. To use this advantage, our method stores all ith bits of the values of a projected vector together, unlike Manhattan hashing which stores each q-bit code consecutively. Specifically, the ith part of a c-dimensional binary string are all the ith bits of the vector. Figure 4 gives an example on how Manhattan hashing and accelerated Manhattan hashing store their codes respectively. In Fig. 4, Let's assume that z_1 and z_2 both lie in the first clusters of their own dimensions, and z_3 and z_4 lie in the third clusters. With the bit-mapping rule of Manhattan hashing in (a), both z_1 and z_2 are mapped into the code "00", and z_3 and z_4 mapped into "10". Thus we get the binary string "00001010" for Manhattan hashing. By using the accelerated Manhattan hashing method of (b), z_1 and z_2 are mapped into "01" and z_3 and z_4 mapped into "10", as can be seen in Fig. 4. Since accelerated Manhattan hashing puts all the ith bit of each q-bit code together, we get the binary string "00111100" for accelerated Manhattan hashing. Please refer to Fig. 4 for details.

Algorithm 1 describes formally the encoding method of q-bit quantization in our accelerated Manhattan hashing. Note that for input requirement, we use cluster center vector instead of threshold vector, because they are equivalent and the former can simplify the calculation. To help readers get a grasp of the encoding process, we define a recursive method here to simplify the illustration, while in reality this algorithm is implemented differently in a iterative form. In the algorithm, the statement $bit = [1, 0, 0, 1]$ in line 1 defines the newly proposed bit-remapping rule. Then it finds out which cluster the value z_i is in (line 2). When $l = 1$, Line 3–8 decide the binary value of its 1st bit. For the cases of $l = 2$ to q, the first $l - 1$ bits is encoded first by calling this method recursively on $l - 1$ layer (line 10), then encode the lth layer bit using our proposed bit-remapping rule (line 11).

Algorithm 1 Encode(z_i, q, τ, l): encoding method of accelerated Manhattan hashing

Input:

1. The ith projected value z_i from the projected vector $\mathbf{z} = [z_1, z_2, \ldots, z_d]$ of the original data point after projection stage;
2. The quantization factor q to quantize z_i into a q-bit binary code;
3. The cluster center vector τ trained using k-means. Each element τ_i in τ represents the ith cluster center of current dimension through all training data.
4. The hierarchical level l indicating which encoding layer this method is currently on. This method is always called as Encode(z_i, q, τ, q) from outside, for the encoding process starts from the bottom layer.

Output:

The output binary string $\left[x_i^1, x_i^2, \ldots, x_i^l\right]$. It only outputs the complete q-bit hash code of z_i when $l = q$, otherwise the output binary code will be an intermediate result of the recursive process.

```
1:  bit = [1, 0, 0, 1];
2:  index = min(abs(z_i − τ)); // min(a) returns the index of the minimum value in a,
    which indicates the k-means cluster the value z_i is in.
3:  if l is 1 then
4:      if index is less than 2^q/2 then
5:          x_i^1 = 0; // index < 2^q/2 means that z_i lies left to the threshold θ_1 in Fig. 3.
6:      else
7:          x_i^1 = 1; // index ≥ 2^q/2 means that z_i lies right to the threshold θ_1 in Fig. 3.
8:      end if
9:  else
10:     [x_i^1, x_i^2, ..., x_i^{l−1}] = Encode(z_i, q, τ, l − 1); // Encode the first l − 1 bits first.
11:     x_i^l = bit[index/2^{q−l} %4]; // index/2^{q−l} gets the cluster index on lth quantization layer, and by
    index/2^{q−l} %4 we get the location of z_i in our bit-remapping strategy [1, 0, 0, 1] as explained
    before in Section 3.2.
12: end if
```

4 Accelerated Manhattan distance measurement

In the original Manhattan hashing proposed in [15], a projected d-dimensional real-valued vector is binarized into a c-dimensional binary string by quantizing each projected value

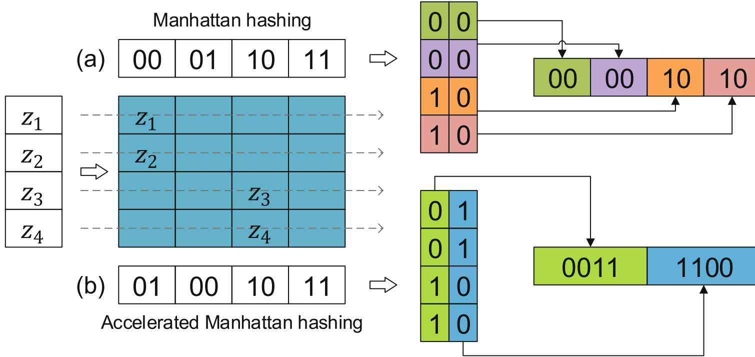

Fig. 4 For 2-bit quantization ($q = 2$) with code length 8 ($c = 8$), there are four projected dimensions ($d = 4$), z_1, z_2, z_3 and z_4. Manhattan hashing stores each q-bit code consecutively, while accelerated Manhattan hashing puts all the ith bit of each q-bit code together. Binary string with the same background color means that these bits are stored compactly together and considered as a whole when measuring Manhattan distance. For ease of understanding, this picture is better to be seen in color mode

into a q-bit binary code, where $c = d \times q$. To calculate the Manhattan distance between two binary strings, we must break them into q-bit length of codes first. Formally, let $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ and $\mathbf{y} = [y_1, y_2, \ldots, y_d]$ be two q-bit quantized binary strings. Here x_i denotes the ith q-bit code of $\mathbf{x}$ and d is the number of dimensions of the projected vectors. The Manhattan distance between $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$d_m(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} |x_i - y_i| \quad (1)$$

Although the decimal distance measurement in Formula (1) is powerful if query precision is the only consideration, it is much more time-consuming compared with Hamming distance measurement. Detailed analyses of the drawbacks of Formula (1) can be found in Section 3.

With the previously introduced encoding method, we can now measure the Manhattan distances between a query hash code and hash codes in the database using as few decimal arithmetic as possible. With the help of our bit-remapping encoding method, we propose a new Manhattan distance measurement which makes full use of bit operations rather than decimal arithmetic. Accelerated Manhattan hashing retains the advantage in query precision of the original Manhattan hashing because it gets exactly the same result with Manhattan hashing, while it runs much faster.

Let $\mathbf{x} = [\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^q}]$ and $\mathbf{y} = [\mathbf{y^1}, \mathbf{y^2}, \ldots, \mathbf{y^q}]$ be two q-bit quantized binary strings to be compared, where $\mathbf{x^i} = [x_1^i, x_2^i, \ldots, x_d^i]$ stands for the set of all the ith bits of the values in a projected vector. For 2-bit quantized accelerated Manhattan hashing, $\mathbf{x} = [\mathbf{x^1}, \mathbf{x^2}]$ and $\mathbf{y} = [\mathbf{y^1}, \mathbf{y^2}]$. If the total code length c of $\mathbf{x}$ is 256, then the length of $\mathbf{x^1}$ and $\mathbf{x^2}$ is 128 respectively, equal to the number of projected dimensions. Here we denote the Manhattan distance between two q-bit quantized binary strings as $M(q, \mathbf{x}, \mathbf{y})$. If 1-bit quantization (SBQ) is adopted, we can get $M(1, \mathbf{x}, \mathbf{y}) = p(\mathbf{x} \oplus \mathbf{y})$ in which $\oplus$ is the bitwise XOR operation and $p(\cdot)$ is the bit-count operation.

Before the detailed description of 2-bit quantization Manhattan distance measurement, three notions concerning relative positions of projected values must be explained first, i.e. “near-end”, “far-end” and “same-end”. For two q-bit quantized codes of the same dimension

from different projected vectors, e.g. x_i and y_i in Formula (1), the above notions denotes which ends of the $(q-1)$-bit regions these two codes are in and their relations. Figuratively, the qth-layer thresholds divide each $(q-1)$-bit cluster into two smaller clusters. For example, Fig. 5a shows the 2-bit encoding result, and with four third-layer thresholds (the four thinner lines in Fig. 5b), each 2-bit cluster is divided into two 3-bit clusters. The notions “near-end”, “far-end” and “same-end” denote the relative location relationship between two 3-bit clusters from two different 2-bit clusters that generate them. For example, the 2-bit region “01” is divided into “011” and “010”, and “10” is divided into “101” and “100”, as shown in Fig. 5c. Among the four pairs of 3-bit clusters from different 2-bit regions, distance between “010” and “101” is 3, which is the smallest, so they are on the “near-end” to each other. Likewise, distance between “011” and “100” is 5 which is the largest, so their relationship is called “far-end”. As for the other two pairs, 011, 101 and 010, 100, the distances are both 4, which make them “same-end”. Every pair of 3-bit clusters generated from different 2-bit regions has similar location relationship, and for clarity we only show some of them in Fig. 5. These notions are very important in the deduction process of the following 2-bit and general q-bit quantization distance measurement.

4.1 2-bit quantization measurement

As shown in Fig. 3b, 2-bit quantization extends the Manhattan distance value scope from {0, 1} of 1-bit quantization to {0, 1, 2, 3} by making a further bisection on both sides of θ_1 in Fig. 3. The basic distance between regions ‘0x’ and ‘1y’ (x and y stand for the newly added bit) is extended from 1 to 2, which can be denoted as $2M(1, \mathbf{x^1}, \mathbf{y^1}) = 2p(\mathbf{x^1} \oplus \mathbf{y^1})$, because after the 1-bit to 2-bit splitting both sides of θ_1 are split into 2 regions, generating twice clusters to each projected dimension. It is obvious in Fig. 3b that distance between ‘01’ and ‘11’ is 3 and distance between ‘00’ and ‘10’ is 1 (readers can simply count the

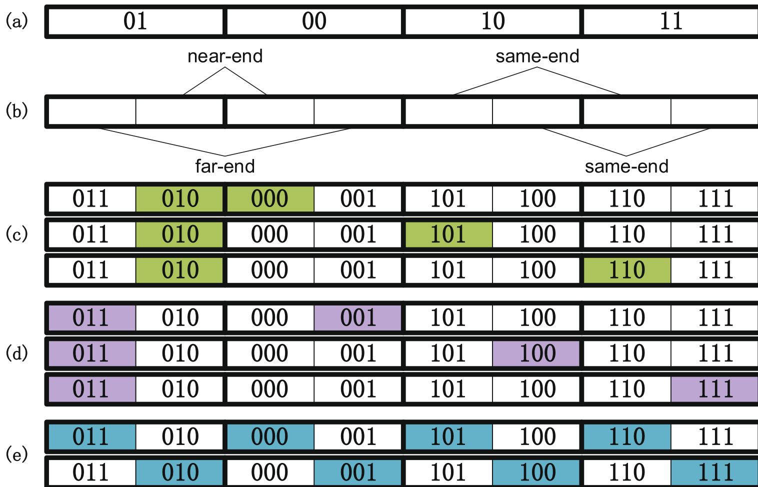

Fig. 5 **a** 2-bit encoding result; **b** illustration of the notions “near-end”, “far-end” and “same-end” on 3-bit quantization; **c** examples of “near-end”: clusters with *green background* is on “near-end” to “010”; **d** examples of “far-end”: clusters with *purple background* is on “far-end” to “011’; **e** examples of “same-end”: clusters with *blue background* is on “same-end” to each other. For ease of understanding, this picture is better to be seen in color mode

distance between clusters in Fig. 3b). Our algorithm, however, calculates the Manhattan distance by making some adjustment to the basic extended distance based on the value of the newly added bits 'x' and 'y'. But no matter what the values of 'x' and 'y' are there's one thing for sure, that is distance between '0x' and '1y' is $2 + \alpha$, where α can be -1, 0 and 1. The actual values of 'x' and 'y' decide the value of α: if 'x' = 1, 'y' = 0 or 'x' = 0, 'y' = 1 ('0x' and '1y' are on same-end), α will be 0, and thus the distance is 2; if 'x' = 1 and 'y' = 1 ('0x' and '1y' are on far-end), α will be 1, and thus the distance is 3; if 'x' = 0 and 'y' = 0 ('0x' and '1y' are on near-end), α will be -1, and thus the distance is 1. For example, distance between '01' and '11' is $3 = 2+1$, where 2 is the basic extended distance between regions '0x' and '1y' after the secondary division and the appended distance 1 is for distance compensation to the "far-end" location relationship; '00' and '10' lie on the near-end, so their distance is $1 = 2 - 1$.

It is evident that only distance between clusters on the same-end of the original 1-bit regions (such as '01' and '10') needs no adjustment to the basic extended distance $2M(1, \mathbf{x^1}, \mathbf{y^1})$. In other situations, i.e. clusters lying on different ends (far-end or near-end) and clusters generated from the same 1-bit region, distance factor 1 must be added to or subtracted from $2M(1, \mathbf{x^1}, \mathbf{y^1})$ to get the correct Manhattan distance.

Remember that our goal is to calculate the correct Manhattan distance between two binary strings with code length c using only $O(q)$ decimal arithmetic, no matter how large c is and how small q is. Our bit storage strategy in Fig. 4 has guaranteed the $O(q)$ complexity for distance calculation by considering all the ith bits of different projected dimensions as a whole. For 2-bit quantization, if we ignore the second bit 'x' and 'y' of each 2-bit binary code '0x' and '1y', we can easily get the total Manhattan distance by simply doubling the 1-bit Manhattan distance of the first bits, that is $M(2, \mathbf{x}, \mathbf{y}) = 2M(1, \mathbf{x^1}, \mathbf{y^1})$. However, 'x' and 'y' cannot be ignored, so adjustment must be made to each value pair of $\mathbf{x}$ and $\mathbf{y}$ based on their location relationship. Since we've already known that the "far-end" situation needs to be appended by 1 and the "near-end" situation needs to subtract 1 from the basic extended distance, we can count the number a of value pairs that need to do addition adjustment as well as the number s of subtraction adjustment, and thus $M(2, \mathbf{x}, \mathbf{y}) = 2M(1, \mathbf{x^1}, \mathbf{y^1}) + a - s$, because all adjustments is of distance factor 1. Two decimal arithmetic have been used here, so to achieve our goal of $O(q)$ complexity, a and s must be calculated using only bitwise operations.

For two input vectors $\mathbf{x}$ and $\mathbf{y}$, if the kth value pair is on the same-end, their first bits and second bits must be different respectively. This is because only two values lie in different sides of θ_1 and same sides of θ_{21} and θ_{22} (i.e. one lies between θ_{21} and θ_{22}, the other is not) can they be considered to be on "same-end", which is evident in Fig. 3. Thus we can easily get $(x_k^1 \oplus y_k^1) \oplus (x_k^2 \oplus y_k^2) = (x_k^1 \oplus x_k^2) \oplus (y_k^1 \oplus y_k^2) = 0$, where x_k^i denotes the kth value of $\mathbf{x^i}$. Therefore, the result of the following formula denotes which codes need to do adjustment by 1, and thus we can count the number of '1's and add it to $2M(1, \mathbf{x^1}, \mathbf{y^1})$:

$$A(2, \mathbf{x}, \mathbf{y}) = (\mathbf{x^1} \oplus \mathbf{x^2}) \oplus (\mathbf{y^1} \oplus \mathbf{y^2}) \tag{2}$$

The result of $A(2, \mathbf{x}, \mathbf{y})$ is a d-bit-length (d is the number of projected dimensions) binary string in which the digits with value "1" denotes that its corresponding projected value pair fall in different ends and need to do adjustment by 1, that is "far-end", "near-end" or clusters generated from the same 1-bit region.

Notice that $A(2, \mathbf{x}, \mathbf{y})$ denotes the value pairs that need to do adjustment, in which the "far-end" situation and clusters of the same 1-bit region need to do addition adjustment,

while the "near-end" situation requires a subtraction adjustment. It is obvious from Fig. 3b that only distance between '00' and '10' needs subtraction adjustment. Thus we can get the following formula, of which the result denotes which codes need to do subtraction by 1:

$$S(2, \mathbf{x}, \mathbf{y}) = (\mathbf{x^1} \oplus \mathbf{y^1}) \wedge [(\mathbf{x^2} \oplus \mathbf{1}) \wedge (\mathbf{y^2} \oplus \mathbf{1})] \tag{3}$$

where "$\mathbf{1}$" in $S(2, \mathbf{x}, \mathbf{y})$ denotes a binary string containing only 1, i.e. 0xFFFF if code length d of $\mathbf{x^i}$ is 16, and $\wedge$ is the bitwise AND operation.

Therefore, number a of value pairs that need to do addition adjustment is $a = p(A(2, \mathbf{x}, \mathbf{y})) - p(S(2, \mathbf{x}, \mathbf{y}))$, and the number s of subtraction adjustment is $s = p(S(2, \mathbf{x}, \mathbf{y}))$, where $p(\mathbf{x})$ is the bit-count operation which counts the numbers of 1s in the binary string $\mathbf{x}$. Summarizing the formulas given above, we get the following 2-bit accelerated Manhattan distance measurement formula:

$$M(2, \mathbf{x}, \mathbf{y}) = 2M(1, \mathbf{x^1}, \mathbf{y^1}) + p(A(2, \mathbf{x}, \mathbf{y})) - 2p(S(2, \mathbf{x}, \mathbf{y})) \tag{4}$$

in which $M(1, \mathbf{x^1}, \mathbf{y^1}) = \mathbf{x^1} \oplus \mathbf{y^1}$.

In Formula (4), the $2 \times a$ operation can be realized by the efficient bit-shifting operation, i.e. performing a left shift by 1 bit on a. Thus, there are only 2 decimal operations in the whole distance measuring process for 2-bit quantization, no matter how long the input codes are.

For ease of understanding of the 2-bit new Manhattan distance algorithm, we list all pairs of 2-bit hash codes in Table 1, along with their adjustment factors, subtraction factors and the 2-bit Manhattan distances. Readers can refer to Table 1 to grasp a deeper understanding of Formula (4).

4.2 q-bit measurement

Given Manhattan distance $M(q-1, \mathbf{x}, \mathbf{y})$ and its corresponding subtraction adjustment factor $S(q-1, \mathbf{x}, \mathbf{y})$ for $(q-1)$-bit quantization, we can easily deduce the measurement $M(q, \mathbf{x}, \mathbf{y})$ for q-bit quantization. Firstly, as can be seen from the way we calculate $M(2, \mathbf{x}, \mathbf{y})$, it is intuitive that $2M(q-1, \mathbf{x^1}, \mathbf{y^1})$ is the basic extended distance for two q-

Table 1 The 2-bit Manhattan distance based on our newly proposed encoding rule

	01		00		10		11	
01	$A=0$	$M_1=0$	$A=1$	$M_1=0$	$A=0$	$M_1=1$	$A=1$	$M_1=1$
	$S=0$	$\mathbf{M_2}=0$	$S=0$	$\mathbf{M_2}=1$	$S=0$	$\mathbf{M_2}=2$	$S=0$	$\mathbf{M_2}=3$
00	$A=1$	$M_1=0$	$A=0$	$M_1=0$	$A=1$	$M_1=1$	$A=0$	$M_1=1$
	$S=0$	$\mathbf{M_2}=1$	$S=0$	$\mathbf{M_2}=0$	$S=1$	$\mathbf{M_2}=1$	$S=0$	$\mathbf{M_2}=2$
10	$A=0$	$M_1=1$	$A=1$	$M_1=1$	$A=0$	$M_1=0$	$A=1$	$M_1=0$
	$S=0$	$\mathbf{M_2}=2$	$S=1$	$\mathbf{M_2}=1$	$S=0$	$\mathbf{M_2}=0$	$S=0$	$\mathbf{M_2}=1$
11	$A=1$	$M_1=1$	$A=0$	$M_1=1$	$A=1$	$M_1=0$	$A=0$	$M_1=0$
	$S=0$	$\mathbf{M_2}=3$	$S=0$	$\mathbf{M_2}=2$	$S=0$	$\mathbf{M_2}=1$	$S=0$	$\mathbf{M_2}=0$

All pairs of 2-bit hash codes are listed, in which 'A' denotes the adjustment factor of Formula (2), 'S' denotes the subtraction factor in Formula (3), 'M_1' denotes the 1-bit Manhattan distance between $\mathbf{x^1}$ and $\mathbf{y^1}$ (the first bits of 2-bit hash codes), and 'M_2' denotes the 2-bit Manhattan distance in Formula (4)

bit quantized binary strings. Specifically, the actual distance between $\mathbf{x}$ and $\mathbf{y}$ is $2M(q-1, \mathbf{x^1}, \mathbf{y^1}) + a - s$ where a and s denotes respectively the number of value pairs that need to do addition adjustment and subtraction adjustment, and $M(q-1, \mathbf{x^1}, \mathbf{y^1})$ calculates the Manhattan distance between $[\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^{q-1}}]$ and $[\mathbf{y^1}, \mathbf{y^2}, \ldots, \mathbf{y^{q-1}}]$.

Next, we need to deduce the two adjustment factors $A(q, \mathbf{x}, \mathbf{y})$ and $S(q, \mathbf{x}, \mathbf{y})$ from their $(q-1)$-bit counterparts. It is easy to induct from the 2-bit encoding method that in q-bit quantization if the kth pair of values are on same-end, the combined number of '1's in this two codes shall be an even number, which is provable using mathematical induction. This statement is already proved to be true when $q = 2$. When it comes to q-bit quantization, if the original two $(q-1)$-bit regions are in the same-end, the qth bits of the two codes will be the same. This is natural due to our bit-mapping rule. To put it simply, each $(q-2)$-bit cluster is divided into four sections and the last bit of each section is encoded into $[1, 0, 0, 1]$ (see the Algorithm 1). Two $(q-1)$-bit clusters share the same last-bit-mapping ($[1, 0]$ or $[0, 1]$) if they are on the same-end, which makes the combined '1's of the two q-bit codes to be even. On the other hand, if the original two $(q-1)$-bit regions are in different ends, the qth bits of the two codes will be different which also get an even number of '1's. Hence, we can get $xor(q, x_k^1) \oplus xor(q, y_k^1) = 0$ for the kth pair of values fall in same-end, where $xor(q, \mathbf{x^1}) = \mathbf{x^1} \oplus \mathbf{x^2} \oplus \ldots \oplus \mathbf{x^q}$. Thus $A(q, \mathbf{x}, \mathbf{y})$ is as follows, of which the result denotes the pairs of codes whose distance need adjustment:

$$A(q, \mathbf{x}, \mathbf{y}) = xor(q, \mathbf{x^1}) \oplus xor(q, \mathbf{y^1}) \tag{5}$$

Subtraction factor, however, is not as easy to deduce as the addition factor. With the increase of quantization number q, it's getting more complex to identify whether the two input values fall in the near-end or not. Therefore, we propose a recursive-like (or can be seen as divide-and-conquer) formula to simplify this problem. To be specific, if $x_k^1 \oplus y_k^1 = 1$ denotes that the kth pair of codes are not in the same side of threshold θ_1, we can get that $xor(q-1, x_k^2) \oplus xor(q-1, 1)$ and $xor(q-1, y_k^2) \oplus xor(q-1, 1)$ are necessary and sufficient conditions for near-end situation, where $xor(q-1, \mathbf{x^2}) = \mathbf{x^2} \oplus \mathbf{x^3} \oplus \ldots \oplus \mathbf{x^q}$; far-end, on the other hand, changes the condition to $xor(q, 1)$ rather than $xor(q-1, 1)$. This statement is also provable using mathematical induction.

For start, let's denote the to-be-compared q-bit value pair as x_k and y_k. Suppose that x_k lies in left side of threshold θ_1 and y_k in right side, and thus the near-end situation is equivalent to the situation that x_k is the right-hand q-bit cluster of its corresponding $(q-1)$-bit cluster and y_k be the left-hand. As we've mentioned, each $(q-2)$-bit cluster is divided into four sections and the last bit of each section is encoded into $[1, 0, 0, 1]$. If the $(q-1)$-bit cluster is the left-hand of its corresponding $(q-2)$-bit cluster, there will be $xor(q-2, x_k^2) \oplus xor(q-1, 1) = 1$ according to our assumption. In this case, the bit-mapping of this $(q-1)$-bit cluster is $[1, 0]$, which makes the last bit of x_k be 0 and $xor(q-1, x_k^2) \oplus xor(q-1, 1) = 1$ hold up. If the $(q-1)$-bit cluster is the right-hand of its corresponding $(q-2)$-bit cluster, there will be $xor(q-2, x_k^2) \oplus xor(q-2, 1) = 1$. With the $[0, 1]$ bit-mapping, the last bit 1 also makes $xor(q-1, x_k^2) \oplus xor(q-1, 1) = 1$ hold up. The similar statement of y_k can be proved accordingly. Therefore, if x_k and y_k is "near" to each other, we'll have $[xor(q-1, x_k^2) \oplus xor(q-1, 1)] \wedge [xor(q-1, y_k^2) \oplus xor(q-1, 1)]$.

On the other hand, if $x_k^1 \oplus y_k^1 = 0$, namely they fall in the same side of threshold θ_1, by abandoning the first bits x_k^1 and y_k^1 we reduce this problem to $(q-1)$-bit quantization. Summarizing all the explanations given above , we get the following formula which recursively calculates the pairs of codes whose distance need subjection adjustment:

$$\begin{aligned} S(q, \mathbf{x}, \mathbf{y}) = & \Big[(\mathbf{x^1} \oplus \mathbf{y^1}) \wedge (xor(q-1, \mathbf{x^2}) \oplus xor(q-1, \mathbf{1})) \\ & \wedge (xor(q-1, \mathbf{y^2}) \oplus xor(q-1, \mathbf{1}))\Big] \\ & \vee \Big[\neg(\mathbf{x^1} \oplus \mathbf{y^1}) \wedge S(q-1, \mathbf{x^2}, \mathbf{y^2})\Big] \end{aligned} \tag{6}$$

where $\wedge$ is the bitwise AND operation, $\vee$ denotes the bitwise OR operation and $\neg$ denotes the bitwise NOT operation.

Finally, the measurement of Manhattan distance is as follows:

$$M(q, \mathbf{x}, \mathbf{y}) = 2M(q-1, \mathbf{x^1}, \mathbf{y^1}) + p(A(q, \mathbf{x}, \mathbf{y})) - 2p(S(q, \mathbf{x}, \mathbf{y})) \tag{7}$$

The advantages of our encoding method is that there's no need to break the compactly stored hash strings into q-bit quantized codes to get its corresponding decimal values; the ith bit of all dimensions can be seen and calculated as a whole. It is evident from Formula (5) and Formula (6) that $A(q, \mathbf{x}, \mathbf{y})$ and $S(q, \mathbf{x}, \mathbf{y})$ consist only bitwise operations. Therefore, with $2\times$ realized by left bit-shifting operation, the number of decimal operations is linear to quantization number q rather than code length c. Generally we have $c \gg q$, so our proposed method runs much faster than the original Manhattan hashing.

Algorithm 2, 3 and 4 are respectively the formal measurement algorithms for the calculating of accelerated Manhattan distance, adjustment factor and subtraction adjustment factor.

Algorithm 2 $M(q, \mathbf{x}, \mathbf{y})$: Manhattan distance measurement of accelerated Manhattan hashing

---

Input:

The quantization factor q;

Two binary strings $\mathbf{x} = [\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^q}]$ and $\mathbf{y} = [\mathbf{y^1}, \mathbf{y^2}, \ldots, \mathbf{y^q}]$, in which $\mathbf{x^j} = \left[x_1^j, x_2^j, \ldots, x_d^j\right]$.

Output:

The Manhattan distance M between $\mathbf{x}$ and $\mathbf{y}$.

1: $M_{q-1} = M(q-1, \mathbf{x^1}, \mathbf{y^1})$;
// $M(q-1, \mathbf{x^1}, \mathbf{y^1})$ calculates the Manhattan distance between $[\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^{q-1}}]$ and $[\mathbf{y^1}, \mathbf{y^2}, \ldots, \mathbf{y^{q-1}}]$

2: $A = A(q, \mathbf{x}, \mathbf{y})$; // The adjustment factor in Algorithm 3.

3: $S = S(q, \mathbf{x}, \mathbf{y})$; // The subtraction adjustment factor in Algorithm 4.

4: $M = M_{q-1} << 1 + popcnt(A) - popcnt(S) << 1$;
// $popcnt(A)$ counts the number of 1s in binary string A, and $<<$ denotes the left bit-shifting operation in programming languages.

Algorithm 3 $A(q, \mathbf{x}, \mathbf{y})$: the adjustment factor of accelerated Manhattan distance measurement

Input:
The quantization factor q;
Two binary strings $\mathbf{x} = [\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^q}]$ and $\mathbf{y} = [\mathbf{y^1}, \mathbf{y^2}, \ldots, \mathbf{y^q}]$, in which $\mathbf{x^j} = [x_1^j, x_2^j, \ldots, x_d^j]$.

Output:
The adjustment factor A between $\mathbf{x}$ and $\mathbf{y}$.

1: $A = xor(q, \mathbf{x^1}) \oplus xor(q, \mathbf{y^1})$;
$// \; xor(q, \mathbf{x^1}) = \mathbf{x^1} \oplus \mathbf{x^2} \oplus \ldots \oplus \mathbf{x^q}$

Algorithm 4 $S(q, \mathbf{x}, \mathbf{y})$: the subtraction adjustment factor of accelerated Manhattan distance measurement

Input:
The quantization factor q;
Two binary strings $\mathbf{x} = [\mathbf{x^1}, \mathbf{x^2}, \ldots, \mathbf{x^q}]$ and $\mathbf{y} = [\mathbf{y^1}, \mathbf{y^2}, \ldots, \mathbf{y^q}]$, in which $\mathbf{x^j} = \left[x_1^j, x_2^j, \ldots, x_d^j\right]$.

Output:
The subtraction adjustment factor S between $\mathbf{x}$ and $\mathbf{y}$.

1: $bit1_{xor} = \mathbf{x^1} \oplus \mathbf{y^1}$; // Decide whether each pair are on the same side of threshold θ_1 in Fig. 3.
2: $S_q = bit1_{xor} \wedge (xor(q-1, \mathbf{x^2}) \oplus xor(q-1, \mathbf{1})) \wedge (xor(q-1, \mathbf{y^2}) \oplus xor(q-1, \mathbf{1}))$; // The near-end determination if each comparing pair are not on the same side of θ_1.
$// \; xor(q-1, \mathbf{x^2}) = \mathbf{x^2} \oplus \mathbf{x^3} \oplus \ldots \oplus \mathbf{x^q}$
3: $S_{q-1} = \neg bit1_{xor} \wedge S(q-1, \mathbf{x^2}, \mathbf{y^2})$; // If the pair is on the same side then this problem is reduced to $(q-1)$-bit.
4: $S = S_q \vee S_{q-1}$;

5 Experiment

5.1 Data sets

To evaluate the efficiency of our accelerated Manhattan distance measure method, we use three publicly available datasets, i.e. ANN_SIFT1m [10–12], ANN_GIST1m [12] and Tiny580K [7].

All datasets we use are widely popular image datasets, each has different characteristics, image sizes and resolutions, along with variable capacities. ANN_SIFT1m and ANN_GIST1m [12] both contain 1M feature vectors, but the dimension of the former is 128 while that of the latter is 960. Tiny580K [7] consists of 580K 320-dimensional GIST descriptors of Tiny Images (whose resolution is 32×32 pixels).

5.2 Baselines

Accelerated Manhattan hashing can be used with any projection functions to compose a complete hashing method. In this paper, we choose representative projection functions, namely LSH [1, 6], PCA-Hash [7, 13], SH [29] and ITQ [7]. Detailed introduction of these

Table 2 Computational Cost of ANN retrieval on ANN_SIFT1m dataset with time measured in seconds

	$q = 2$		$q = 3$		$q = 2$		$q = 4$	
	AMH	MH	AMH	MH	AMH	MH	AMH	MH
	$c = 16$				$c = 32$			
LSH	7.7	37.2	–	–	10.1	76.5	27.9	43.1
PCA	8.3	39.1	–	–	9.5	76.2	29.7	44.7
SH	8.8	37.6	–	–	9.3	75.7	29.3	44.0
ITQ	8.7	37.3	–	–	9.7	71.3	27.8	39.7
	$c = 48$				$c = 64$			
LSH	19.5	114.9	13.1	79.7	8.4	160.5	27.8	91.0
PCA	20.0	121.9	12.3	81.0	8.5	167.8	28.1	95.3
SH	18.6	116.91	12.7	81.3	7.7	164.4	28.5	92.0
ITQ	19.7	113.6	12.6	72.7	8.4	155.6	26.0	84.6
	$c = 96$				$c = 128$			
LSH	20.4	253.7	14.5	161.2	11.1	348.3	20.5	187.4
PCA	20.1	256.9	14.0	165.1	10.6	349.2	22.0	196.6
SH	20.3	257.3	14.4	168.0	10.9	344.3	21.0	195.5
ITQ	20.6	257.3	13.4	153.1	11.6	348.1	19.1	178.2

four projection functions can be found in Section 2.1. For quantization stage, the original Manhattan hashing [15] is the baseline for efficiency comparisons. Therefore, we can get different variants of a specific hashing method by combining one of the four projection functions (LSH, PCA-Hash, SH and ITQ) with one quantization method (the original Manhattan hashing or the proposed accelerated Manhattan hashing). For example, "PCA-MH" denotes the combination of PCA projection with Manhattan hashing quantization, and "ITQ-AMH" denotes one variant of ITQ projection combined with accelerated Manhattan hashing quantization.

For all the baseline functions we refer to, we use the source codes as well as parameters provided by the authors, and all experiments are conducted on our workstation with Intel(R) Xeon(R) CPU of 2.40 GHz and 48G memory. We use Matlab2014a to do the encoding work. The experiments to evaluate retrieval efficiency are conducted using C++ applications in which bit operations can be realized by directly invoking the built-in functions, such as $__popcnt$, $__popcnt16$ and $__popcnt64$ for bit-count operation.[2]

5.3 Results

For all experiments, 1000 points are randomly selected to be queries, and the rest are left to be the retrieval set on which the queries are performed. We also select 10,000 points randomly from the database to form the training set on which the projection functions and thresholds are learned. As mentioned, we train the four projection functions using their source codes and default parameter settings, and we learn all the thresholds in Fig. 3 to

[2]Codes are provided on http://ise.thss.tsinghua.edu.cn/MIG/resources.jsp

Table 3 Computational Cost of ANN retrieval on ANN_GIST1m dataset with time measured in seconds

	$q = 2$		$q = 3$		$q = 2$		$q = 4$	
	AMH	MH	AMH	MH	AMH	MH	AMH	MH
	$c = 16$				$c = 32$			
LSH	6.9	36.4	–	–	8.6	68.4	29.2	44.5
PCA	6.5	35.8	–	–	8.0	71.0	27.6	45.2
SH	7.2	35.0	–	–	8.9	70.4	31.7	45.0
ITQ	6.4	35.5	–	–	8.7	70.6	28.1	42.3
	$c = 48$				$c = 64$			
LSH	19.3	112.4	12.5	77.9	7.0	154.3	26.8	89.8
PCA	19.3	112.8	13.4	81.7	7.6	156.2	27.6	94.4
SH	19.2	115.1	13.2	82.0	7.1	158.7	30.1	95.2
ITQ	18.6	105.9	13.2	76.4	7.5	149.7	27.8	83.8
	$c = 96$				$c = 128$			
LSH	20.2	239.8	14.4	164.7	8.6	331.4	20.0	186.1
PCA	20.5	236.8	15.0	172.5	8.6	322.9	20.7	186.7
SH	18.8	237.9	14.2	166.1	8.8	324.9	21.8	189.7
ITQ	19.7	246.5	13.5	166.0	9.5	330.9	19.3	177.9
	$c = 192$				$c = 256$			
LSH	18.9	492.4	13.6	350.8	11.3	677.6	20.4	397.0
PCA	19.5	512.5	14.7	348.5	11.4	665.6	20.4	395.7
SH	19.6	499.9	14.4	346.7	10.7	677.1	20.6	393.2
ITQ	20.2	497.8	13.4	345.3	11.2	657.0	23.6	395.8
	$c = 384$				$c = 512$			
LSH	20.3	1013.9	30.7	733.8	24.7	1341.6	53.7	794.5
PCA	20.0	1010.3	30.6	770.5	25.7	1346.3	51.1	780.6
SH	19.9	1004.2	33.2	747.4	23.3	1357.5	53.5	790.9
ITQ	21.4	1010.0	28.1	718.9	24.9	1364.4	51.7	830.7

cluster projected values using k-means algorithm as proposed in [15]. All the experiments reported in this paper are averaged over 10 random training/test partitions.

For the validation of effectiveness, we do queries using both Manhattan hashing and accelerated Manhattan hashing and compare their result to make sure that our method calculates Manhattan distance correctly. Meanwhile, we record their query timecost respectively to evaluate the efficiency of our method.

Tables 2, 3 and 4 respectively show the Computational Cost of ANN retrieval experiments on the three datasets using Manhattan distance measurement and accelerated Manhattan distance combined with four basic hashing methods we choose. Each result shows the query costs of one of the four projection functions combined with one quantization method, i.e. Manhattan hashing (MH) or accelerated Manhattan hashing (AMH), on one setting of c and q. Evidently, AMH outperforms MH in all conditions, demonstrating that our proposed encoding method and distance measurement achieves great improvement

Table 4 Computational Cost of ANN retrieval on Tiny580K dataset with time measured in seconds

	$q = 2$		$q = 3$		$q = 2$		$q = 4$	
	AMH	MH	AMH	MH	AMH	MH	AMH	MH
	$c = 16$				$c = 32$			
LSH	4.4	20.9	–	–	5.3	41.2	17.3	26.0
PCA	4.0	20.8	–	–	5.3	41.5	16.9	26.8
SH	4.2	21.7	–	–	5.3	41.5	17.0	26.1
ITQ	4.2	20.3	–	–	5.2	40.2	16.3	24.8
	$c = 48$				$c = 64$			
LSH	10.8	64.7	7.6	45.1	4.6	92.7	16.9	55.4
PCA	11.7	67.0	7.4	47.3	4.7	90.2	16.7	53.7
SH	11.0	63.9	7.3	45.3	4.8	92.0	16.5	54.8
ITQ	11.2	63.1	7.2	44.3	4.3	89.9	16.5	52.1
	$c = 96$				$c = 128$			
LSH	11.0	142.9	8.1	95.2	5.5	193.5	12.3	111.3
PCA	10.6	140.3	8.2	93.4	5.0	188.7	12.4	109.4
SH	11.0	141.6	7.9	94.2	5.6	190.2	12.7	114.6
ITQ	11.0	142.7	8.0	91.0	5.0	193.6	12.6	110.3
	$c = 192$				$c = 256$			
LSH	11.2	295.4	8.5	202.5	6.7	386.8	12.8	230.8
PCA	11.7	294.6	8.0	203.0	6.8	390.8	12.5	226.6
SH	11.3	287.6	8.0	201.2	7.0	389.3	13.3	233.5
ITQ	12.3	294.9	8.1	195.5	6.7	388.2	12.8	229.2

on query efficiency by making full use of bitwise operations. For example, when querying on ANN_GIST1m dataset with code length $c = 256$ and quantization number $q = 2$, AMH is 50 times faster than MH; querying on Tiny580K with $c = 192$ and $q = 3$ using AMH is at least 20 time faster than using MH. Furthermore, as can be seen from Formula (1) and Formula (7), for calculating distance between two q-bit quantized hash codes with code length c, Manhattan hashing does "+" operation at least $(2d - 1)$ times, with d being the number of projected dimensions, while our method performs only $2q$ times. That is to say, the query complexity of Manhattan hashing is $O(d)$, while for accelerated Manhattan hashing it is $O(q)$. Generally, q is much smaller than d, for q is usually set to be 2, 3 or 4, while d's value grows with code length c (c is usually set to be 64, 128, 256 or more). Hence, with code length getting longer, the running time of Manhattan hashing is climbing linearly, while there is no additional cost for our method, which illustrates the advantage of using bitwise operations over decimal arithmetic.

To test the necessity and capability of further expanding the quantization number q into larger values, i.e. q-bit quantization with $q > 4$, we make some extra experiments on ANN_GIST1m dataset and Tiny580K dataset. For the evaluation of effectiveness, we directly use the mean average precision (mAP) metric as defined in [15]. The hash code length is set to be a certain fixed value as we gradually enlarge the quantization number, and mAP and Computational Cost are recorded accordingly. As can be seen in Figs. 6 and 7, the query time of Manhattan hashing is decreasing with q getting larger, because the number of

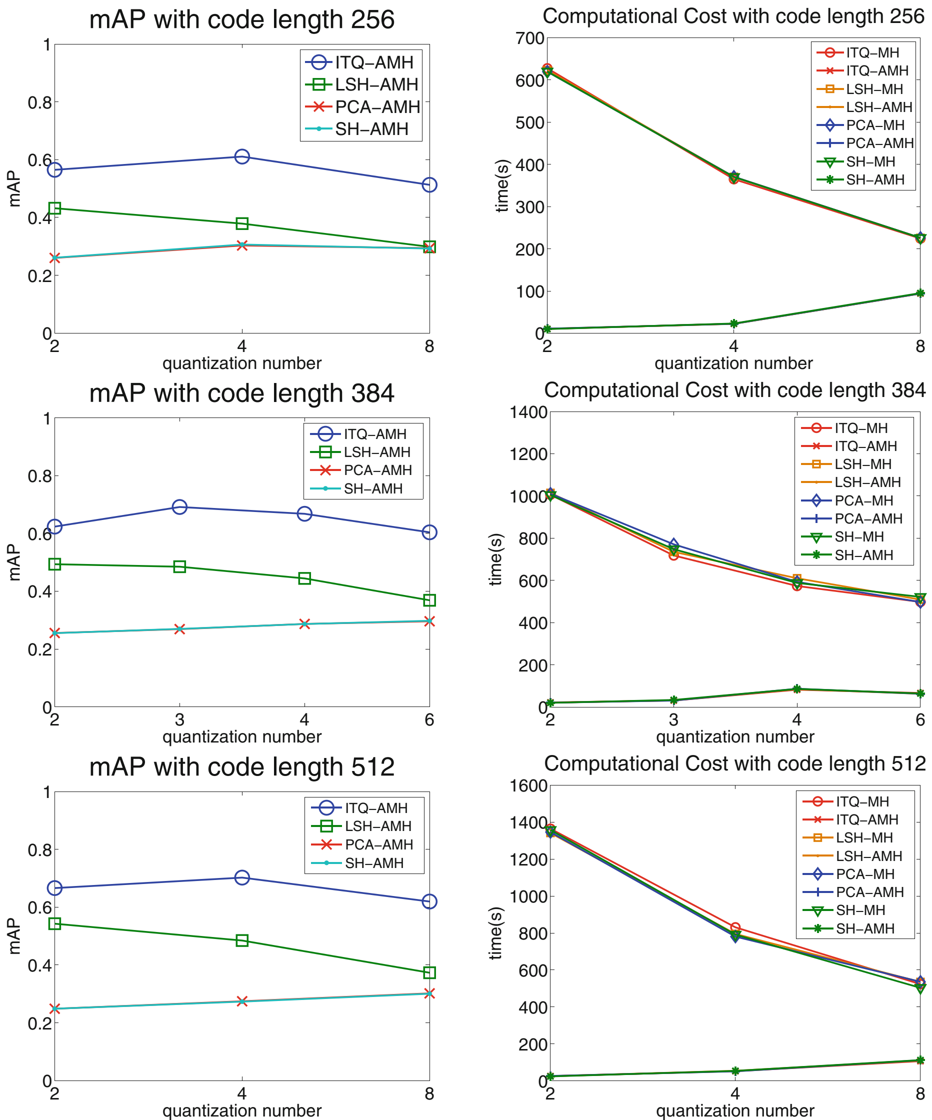

Fig. 6 MAP and Computational Cost curves on ANN_GIST1m with fixed code length and varying quantization number

decimal arithmetic is linear to d, which is c/q where c is the fixed code length. However, as we've mentioned before, the query time of accelerated Manhattan Hashing is linear to q (the number of bits for coding each component of a vector), so with q increasing the computational cost of accelerated Manhattan hashing climbs slightly. But as can be seen in Figs. 6, 7 and [15], it is not necessarily good to have a large quantization number. For most cases, the best mAPs of ITQ-AMH and LSH-AMH are achieved when quantization number is 3 or 4, and their curves show a downward trend with q getting larger. Although PCA-AMH and SH-AMH share a climbing trend in both datasets, it is too subtle to make a difference, and

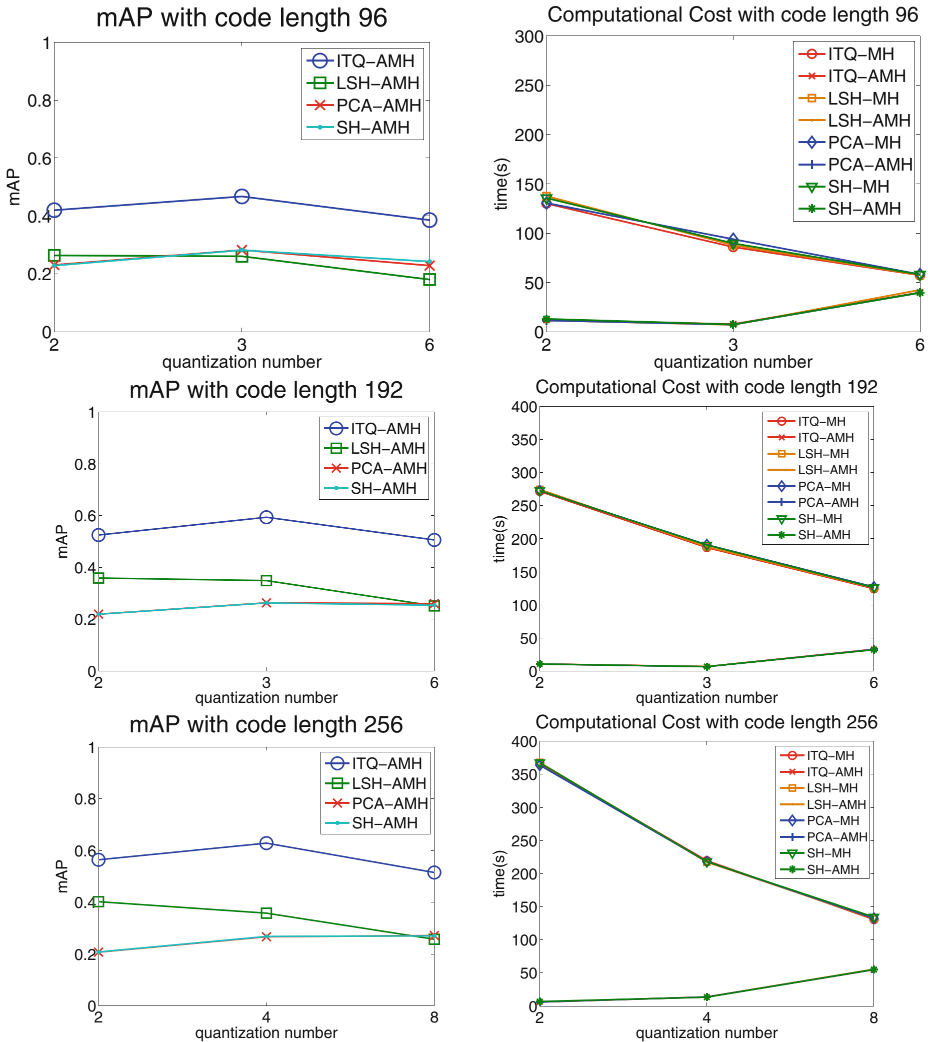

Fig. 7 mAP and Computational Cost curves on Tiny580K with fixed code length and varying quantization number

their performances are far worse than ITQ-AMH. What's more, with fixed code length c and enlarging quantization number q, the number of projected dimensions c/q is lowering, causing more information loss after the projection stage. According to [15], $q = 2$ (2-bit quantization) achieves the best performance for most cases, which makes our method very competitive.

For further reference, the evaluation of effectiveness of our method is also posted here in Fig. 8, in which Manhattan hashing is denoted by hollow signs and accelerated Manhattan hashing marked by solid ones. It is evident from Fig. 8 that MH and AMH get the exact same results when it comes to query precision. AMH adopts the idea of multi-bit quantization in MH, but we improve it with our newly proposed encoding method in Section 3 and distance

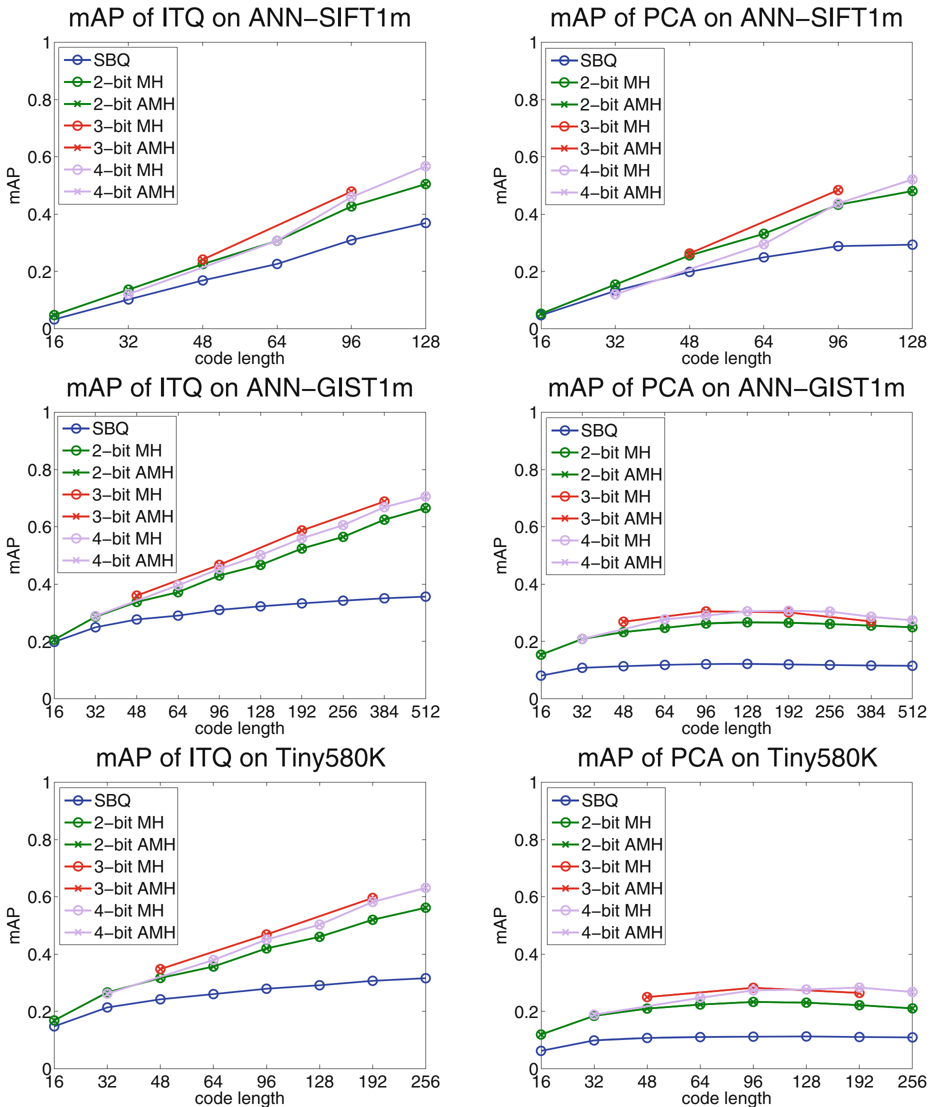

Fig. 8 mAP curves on ANN_SIFT1m, ANN_GIST1m and Tiny580K datasets

measure algorithm in Section 4, in order to get the correct Manhattan distance measurement proposed in [15] using faster bitwise operations to improve query efficiency. To demonstrate the efficiency of our method, we directly adopt the *k*-means cluster algorithm to learn the thresholds when encoding. Therefore, query results of these two quantization methods are supposed to be the same. Since accelerated Manhattan hashing precisely reproduces the experimental results of Manhattan hashing in terms of query precision, the multi-bit quantization methods outperforms SBQ significantly just as what was reported in [15].

When it comes to real applications, the relevant input parameters c and q can be adjusted according to the real-world cases and user demands. Take ANN_GIST1m dataset for example, it is ITQ-AMH with 4-bit quantization and code length $c = 512$ that achieves the best mAP, and in this case accelerated Manhattan hashing runs at least 15 times faster than Manhattan hashing. The mAP of ITQ-AMH with 3-bit quantization and the code length being 384 is slightly worse, but this 1 % query precision sacrifice helps us save at least a quarter of the query time comparing to accelerated Manhattan hashing with $q = 4$ and $c = 512$, further widening the computational gap between Manhattan hashing and accelerated Manhattan hashing. Hence we can choose the quantization number q and code length c flexibly.

6 Conclusion

Manhattan hashing is a hashing quantization solution which fixes the quantization loss caused by traditional SBQ, but it uses time-consuming decimal operations to calculate distances instead of bitwise operations so that it loses the advantage of efficiency. In this paper, we propose an accelerated version of Manhattan hashing by making full use of bitwise operations via bit-remapping. It does the encoding work in a way that is different from the original Manhattan hashing. Based on this, a novel hash code distance measurement that optimizes the calculation of Manhattan distance is proposed to improve query efficiency significantly without any precision loss. Experiments on three benchmark datasets showed that our approach improves the speed of data querying on 2-bit, 3-bit as well as 4-bit quantization by at least one order of magnitude on average without any precision loss.

Acknowledgments This research was supported by the National Natural Science Foundation of China (Grant No.61271394 and 61571269). The authors would like to thank the anonymous reviewers for their valuable comments.

References

1. Andoni A, Indyk P (2008) Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: Communications of the ACM - 50th anniversary issue: 1958–2008, vol 51
2. Baluja S, Covell M (2008) Learning to hash: forgiving hash functions and applications. Data Min Knowl Disc 17(3)
3. Cheng W, Jin X, Sun J-T, Lin X, Zhang X, Wang W (2014) Searching dimension incomplete databases. Knowl Data Eng 26(3)
4. Ding G, Guo Y, Zhou J (2014) Collective matrix factorization hashing for multimodal data. In: Computer vision and pattern recognition
5. Friedman JH, Bentley JL, Finkel RA (1977) An algorithm for finding best matches in logarithmic expected time. ACM Trans Math Softw 3(3)
6. Gionis A, Indyk P, Motwani R et al. (1999) Similarity search in high dimensions via hashing. In: Very large data bases, vol 99
7. Gong Y, Lazebnik S (2011) Iterative quantization: a procrustean approach to learning binary codes. In: Computer vision and pattern recognition
8. Guttman A (1984) R-trees: a dynamic index structure for spatial searching 14(2)
9. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the 13th annual ACM symposium on theory of computing
10. Jegou H, Douze M, Schmid C (2008) Hamming embedding and weak geometric consistency for large scale image search. In: European conference on computer vision
11. Jégou H, Douze M, Schmid C (2010) Improving bag-of-features for large scale image search. Int J Comput Vis 87(3)

12. Jegou H, Douze M, Schmid C (2011) Product quantization for nearest neighbor search. Pattern Analysis and Machine Intelligence 33(1)
13. Jolliffe I (2002) Principal component analysis
14. Kong W, Li W-J (2012) Double-bit quantization for hashing. In: Association for the advancement of artificial intelligence
15. Kong W, Li W-J, Guo M (2012) Manhattan hashing for large-scale image retrieval. In: ACM special interest group on information retrieval
16. Lee Y, Heo J-P, Yoon S-E (2014) Quadra-embedding: binary code embedding with low quantization error. Comput Vis Image Underst 125
17. Lin Z, Ding G, Hu M (2014) Image auto-annotation via tag-dependent random search over range-constrained visual neighbours. Multimedia tools and applications
18. Lin Z, Ding G, Hu M, Wang J (2015) Semantics-preserving hashing for cross-view retrieval. In: Computer vision and pattern recognition
19. Liu W, Wang J, Kumar S, Chang S-F (2011) Hashing with graphs. In: Proceedings of the 28th international conference on machine learning
20. Moran S, Lavrenko V, Osborne M (2013) Neighbourhood preserving quantisation for lsh. In: Proceedings of the 36th international ACM SIGIR conference on research and development in information retrieval
21. Moran S, Lavrenko V, Osborne M (2013) Variable bit quantisation for lsh. In: Association for computational linguistics
22. Mu Y, Shen J, Yan S (2010) Weakly-supervised hashing in kernel space. In: Computer vision and pattern recognition
23. Norouzi M, Blei DM (2011) Minimal loss hashing for compact binary codes. In: International conference on machine learning
24. Raginsky M, Lazebnik S (2009) Locality-sensitive binary codes from shift-invariant kernels. In: Advances in neural information processing systems
25. Song J, Yang Y, Huang Z, Shen HT, Hong R (2011) Multiple feature hashing for real-time large scale near-duplicate video retrieval. In: Proceedings of the 19th ACM international conference on multimedia
26. Uhlmann JK (1991) Satisfying general proximity/similarity queries with metric trees. Inf Process Lett 40(4)
27. Wang J, Kumar S, Chang SF (2010) Semi-supervised hashing for scalable image retrieval. In: Computer vision and pattern recognition
28. Wang X, Jin X, Chen M-E, Zhang K, Shen D (2012) Topic mining over asynchronous text sequences. Knowl Data Eng 24(1)
29. Weiss Y, Torralba A, Fergus R (2009) Spectral hashing. In: Advances in neural information processing systems
30. Wold S, Esbensen K, Geladi P (1987) Principal component analysis. Chemom Intell Lab Syst 2(1)
31. Yu Z, Wu F, Yang Y, Tian Q, Luo J, Zhuang Y (2014) Discriminative coupled dictionary hashing for fast cross-media retrieval. In: Proceedings of the 37th international ACM SIGIR conference on research and development in information retrieval
32. Zhou J, Ding G, Guo Y (2014) Latent semantic sparse hashing for cross-modal similarity search. In: Proceedings of the 37th international ACM SIGIR conference on research and development in information retrieval
33. Zhu X, Huang Z, Cheng H, Cui J, Shen HT (2013) Sparse hashing for fast multimedia search. ACM Trans Inf Syst 31(2)
34. Zhu X, Huang Z, Shen HT, Zhao X (2013) Linear cross-modal hashing for efficient multimedia search. In: Proceedings of the 21st ACM international conference on multimedia
35. Zhu X, Zhang L, Huang Z (2014) A sparse embedding and least variance encoding approach to hashing. Image Processing 23(9)

Wenshuo Chen received her B.Sc. degree from School of Software, Nanjing University, Jiangsu, China in 2013, and currently is a M.D. candidate in School of Software in Tsinghua University, Beijing, China. Her research interests include multimedia information retrieval, human action recognition and video event detection.

Guiguang Ding received his Ph.D degree in electronic engineering from the University of Xidian. He is currently an associate professor of School of Software, Tsinghua University. Before joining School of Software in 2006, he worked as a postdoctoral researcher in Automation Department of Tsinghua University. His current research centers on the area of multimedia information retrieval and mining, in particular, visual object classification, automatic semantic annotation, content-based multimedia indexing, and personal recommendation. He has published about 40 research papers in international conferences and journals and applied for 18 Patent Rights in China.

Zijia Lin received his B.Sc. degree from School of Software, Tsinghua University, Beijing, China in 2011, and currently is a Ph.D. candidate in Department of Computer Science and Technology in the same campus. His research interests include multimedia information retrieval and machine learning.

Jisheng Pei received the B.E. degree in computer software from Tsinghua University in 2011. He is currently working toward the Ph.D. degree in the Department of Computer Science and Technology at Tsinghua university. His research interests include data provenance and business process management.