A General and Efficient Querying Method for Learning to Hash

Jinfeng Li, Xiao Yan, Jian Zhang, An Xu,
James Cheng, Jie Liu, Kelvin K. W. Ng, Ti-chung Cheng
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{jfli,xyan,jzhang,axu,jcheng,jliu6,kwng6,tccheng4}@cse.cuhk.edu.hk

ABSTRACT

As an effective solution to the approximate nearest neighbors (ANN) search problem, learning to hash (L2H) is able to learn similarity-preserving hash functions tailored for a given dataset. However, existing L2H research mainly focuses on improving query performance by learning good hash functions, while Hamming ranking (HR) is used as the default querying method. We show by analysis and experiments that Hamming distance, the similarity indicator used in HR, is too coarse-grained and thus limits the performance of query processing. We propose a new fine-grained similarity indicator, quantization distance (QD), which provides more information about the similarity between a query and the items in a bucket. We then develop two efficient querying methods based on QD, which achieve significantly better query performance than HR. Our methods are general and can work with various L2H algorithms. Our experiments demonstrate that a simple and elegant querying method can produce performance gain equivalent to advanced and complicated learning algorithms.

ACM Reference Format:
Jinfeng Li, Xiao Yan, Jian Zhang, An Xu, James Cheng, Jie Liu, Kelvin K. W. Ng, Ti-chung Cheng. 2018. A General and Efficient Querying Method for Learning to Hash. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3183750

1 INTRODUCTION

The problem of finding the nearest neighbors for a query in a dataset emerges in a variety of applications such as recommendation [5], sequence matching [3], entity resolution [13], de-duplication [31], and similar item retrieval [18, 47]. However, the cost of finding the exact nearest neighbors can be very high for large scale datasets with high dimensionality [21]. Fortunately, it usually suffices to find approximate nearest neighbors (ANN) [14, 19, 21, 27, 39, 42, 51] and ANN search has attracted widespread attention. There are many performance metrics for ANN search, e.g., efficiency, memory consumption, and preprocessing overhead. Among which, efficiency, i.e., retrieving a large portion of the true nearest neighbors in a short time, is an important metric.

Extensive hashing algorithms [2, 6, 14, 19, 27, 39, 41, 42, 51] have been developed for ANN search. Hashing-based methods map items in the dataset to buckets in hash tables and the hash functions are designed to be similarity-preserving so that similar items are hashed to the same bucket with high probability. A query is answered by evaluating the items that are hashed to the same or similar buckets as the query. There are two kinds of hashing-based methods, *locality sensitive hashing* (*LSH*) [16, 21] and *learning to hash* (*L2H*)[1] [19, 41, 42]. LSH uses predefined hash functions without considering the underlying dataset, while L2H learns tailored hash functions for the dataset. Although an additional training stage is needed, recent studies [9, 11, 41, 44] showed by experiments that L2H significantly outperforms LSH in terms of querying efficiency.

The performance of L2H depends on both the hash function and the querying method. Hash function determines the buckets that the items are hashed to, and a good hash function should preserve the similarity between items as much as possible. It explains the superior performance of L2H over LSH in practice because L2H learns dataset-dependent hash functions. As a result, existing L2H researches mainly focus on learning good hash functions and many learning algorithms have been proposed, including spectral hashing (SH) [44], principal component analysis hashing (PCAH) [8, 43], iterative quantization (ITQ) [8, 9], semi-supervised hashing (SSH) [40], and so on.

However, the querying method is also important to the efficiency as it determines the order in which the buckets are probed. A good querying method should fully utilize the information provided by the hash function and probe the buckets containing the true nearest neighbors first. The majority of existing L2H methods adopt *Hamming ranking* (*HR*) as their querying method [4, 19], which probes the buckets in ascending order of their *Hamming distance* to the query.

Although HR is simple and intuitive, it has a critical drawback. Specifically, Hamming distance is a coarse-grained indicator of the similarity between the query and items in a bucket as it is discrete and takes only a limited number of values. As a result, Hamming ranking cannot define a good order for buckets with the same Hamming distance from the query. Therefore, HR usually probes a large number of unfavorable buckets, resulting in poor efficiency. One solution is to use long code so that Hamming distance can classify the buckets into more categories. However, long code has

[1]This paper refers to L2H as algorithms that learn binary embeddings.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183750

problems such as time-consuming sorting, high storage demand and poor scalability, especially for large-scale datasets [21].

While advanced hash function learning algorithms have become increasingly complicated but lead to only marginal gain in query performance, in this paper we show that we can achieve significant gain in query performance with a novel design of the querying method. Observing that L2H projects a query to a real number vector as an intermediate step, we find that the distance between the real number vector and a bucket serves as a better similarity indicator than Hamming distance. We formally define *quantization distance* (*QD*) as the error of quantizing the projected real number vector of a query to the signature of a bucket, and prove that QD provides a (scaled) lower bound for the distance between the items in the bucket and the query. Comparing with Hamming distance, QD is fine-grained as it is continuous and can distinguish buckets with the same hamming distance.

Using QD, we develop a querying algorithm named *QD ranking* which probes the buckets in ascending order of their QD. By analyzing the property of QD, we also design an efficient algorithm called *GQR* to generate the bucket to probe on demand, thus avoiding sorting all buckets at the start. Experimental results show that GQR consistently achieves significantly better performance than Hamming ranking. Our method is also proven to be a general querying method and shown to achieve good performance for a variety of L2H algorithms (e.g., ITQ [8, 9], PCAH [8, 43], and SH [44]) in our experiments.

Our contributions are three folds. Firstly, we show by theory and analysis that QD is a better similarity indicator than Hamming distance, which forms the basis of our QD ranking algorithm. Secondly, we propose the GQR algorithm, which is efficient and solves the slow start problem of QD ranking. Thirdly, extensive experiments on a variety of datasets and hash function learning algorithms were conducted to verify the performance of our methods. We demonstrate that a simple and elegant querying method can achieve performance gain equivalent to complicated learning algorithms, which we hope can stimulate more research on new querying methods for L2H.

Section 2 gives the background of our work. Section 3 discusses the limitations of Hamming ranking. Section 4 defines QD and introduces the QD ranking algorithm. Section 5 proposes the more efficient GQR algorithm. Section 6 reports the performance results. Section 7 discusses related work. Section 8 concludes the paper.

2 LEARNING TO HASH

L2H is conducted in two stages, *learning* and *querying*. Learning trains good hash functions for the dataset of interest. The hash functions are used to construct the hash tables. In the querying stage, a querying method chooses the buckets in each hash table to probe, and items in the chosen buckets are ranked by their distance to the query to obtain the final results. As hash function learning is not the focus of this paper, we refer the interested readers to existing work such as ITQ [8, 9], SH [44], and SSH [40] for details. In this section, we briefly introduce the hashing and querying processes of L2H to provide some background for our work.

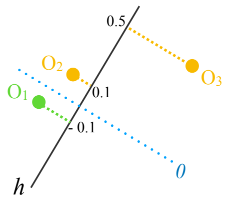


Figure 1: Quantization

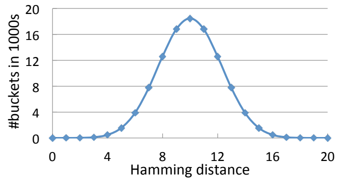


Figure 2: Number of buckets versus hamming distance

2.1 Hashing

Given a d-dimensional item (or query), hashing maps it to a binary code with length m. Similarity is preserved in the mapping, that is, if two items are close to each other in the original d-dimensional space, then there is a high probability that they have identical or similar m-dimensional binary codes. The mapping is achieved by two operations, *projection* and *quantization*.

Projection. Projection applies the learned hash functions to an item and projects it to an m-dimensional real number vector. Denote the hash functions as h_1, h_2, ..., h_m, the projected vector of item o is $p(o)$ =($h_1(o)$, $h_2(o)$, ..., $h_m(o)$). A hash function is usually a d-dimensional vector and the projection is calculated as $h_i(o) = h_i^T o$.

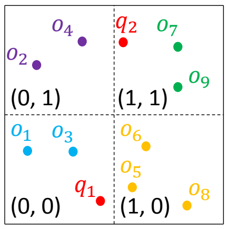
Quantization. Quantization discretizes the projected vector into a binary code $c(o)$ =($c_1(o)$, $c_2(o)$, ..., $c_m(o)$) with $c_i(o) \in \{0, 1\}$ for $1 \leq i \leq m$. Thresholding is usually applied, that is, $c_i(o) = 1$ if $h_i(o)$ is non-negative and $c_i(o) = 0$ otherwise. For two items that are close in the original d dimensional space, an L2H algorithm ensures that their binary codes have a high probability to be similar. A bucket in a hash table contains items with the same binary code. Given a bucket b, we also use b to denote its binary code (also called *signature*) when there is no ambiguity, and we use b_j to denote the j-th bit of the signature.

Quantization inevitably loses some similarity information. Consider an example of mapping items to a single bit in Figure 1. There are three items o_1, o_2, o_3 and their projected values are $-0.1, 0.1, 0.5$, respectively. Thus, o_1 and o_2 are quantized to different buckets, while o_2 and o_3 are quantized to the same bucket. However, o_1 is more similar to o_2 than o_3.

2.2 Querying

Given a query q, the querying process probes a number of buckets and evaluates their belonging items to obtain the query results. Querying usually consists of two steps, *retrieval* and *evaluation*. Retrieval determines which buckets to probe according to the querying method and fetches items in the chosen buckets. Evaluation calculates the distances between q and the retrieved items, and updates the nearest neighbors by ranking the distances.

The querying method has a significant influence on the performance of similarity search as it decides which buckets to probe. Datasets in today's applications are usually large, and to achieve a short query delay, usually only a small fraction of buckets are probed. Thus, a good querying method should effectively identify the buckets containing items similar to the query and probe them as early as possible. Otherwise, computation will be wasted on evaluating unfavorable buckets, resulting in poor efficiency. In addition, the querying method should also be simple as spending too much

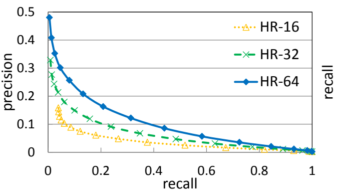


(a) Hamming ranking

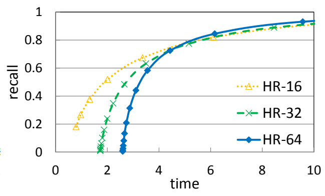
Bucket ID	HD	QD
(0, 0)	0	0
(1, 0)	1	0.2
(0, 1)	1	0.8
(1, 1)	2	1

(b) Comparison between Hamming distance and quantization distance

Figure 3: An example of Hamming ranking



(a) recall-precision curve



(b) recall-time curve

Figure 4: performance of Hamming ranking on cifar-10 dataset

time on deciding which bucket to probe will also harm efficiency. To the best of our knowledge, existing work on L2H [4, 42] extensively use Hamming ranking or its variant, hash lookup, as their querying method.

Hamming ranking. Given a query q, Hamming ranking calculates its binary code $c(q)$ and sorts all the buckets by their hamming distances to $c(q)$. The bucket with smaller hamming distance (where ties are broken arbitrarily) is probed first. If there are B buckets, both the time and space complexity of retrieval (sorting) is $O(B)$. The complexity of evaluation is linear to the number of items to be evaluated.

We further analyze in details the limitations of Hamming ranking in Section 3, which motivates the design of a more efficient and effective querying method.

2.3 Performance Metrics

The performance of a querying method is mainly measured by efficiency, which can be observed from the *recall-time* curve.

Recall-time curve. For a query q, if r of the true k-nearest neighbors are found by a querying method using t seconds, the recall achieved by this method is r/k at t. As time goes on and more buckets are probed, recall gradually increases . The recall-time curve is obtained by plotting the achieved recall against the querying time, which is the main performance indicator we will use to compare different querying methods.

3 LIMITATIONS OF HAMMING RANKING

Although Hamming ranking is simple and intuitive, it suffers from a fundamental problem that its similarity indicator, *Hamming distance*, is too coarse-grained. Firstly, the relationship between *Hamming distance* and *similarity (between a query and the items in a bucket)* is unclear. Secondly, Hamming distance cannot distinguish between buckets with the same Hamming distance. Let m be the code length, then the number of buckets having Hamming distance r from a query is C_m^r. We plot the number of buckets as a function of the Hamming distance from a query in Figure 2, which shows that the number can be quite large even for a moderate Hamming distance. Buckets with the same Hamming distance have the same priority for probing according to Hamming ranking but their items can have very different similarities to the query.

We further illustrate the problems of Hamming ranking using an example. Consider Figure 3a where the code length is 2. Query q_1 is mapped to bucket $(0, 0)$, and hence Hamming ranking will probe bucket $(0, 0)$ first. Both buckets $(0, 1)$ and $(1, 0)$ have a Hamming distance of 1 from q_1, and thus Hamming ranking will assign them equal priority for probing. However, it is obvious that bucket $(1, 0)$ should be probed before bucket $(0, 1)$, since q_1 is actually closer to items in bucket $(1, 0)$. Similarly arguments also apply for another query q_2.

The above example reveals the coarse-grain problem of Hamming ranking that leads to probing many unfavorable buckets, thus resulting in low efficiency. The problem becomes more severe when high recall is required, as buckets with larger Hamming distances may be probed and the number of buckets having the same Hamming distance grows rapidly as shown in Figure 2.

Long code length [8, 9, 11, 20, 44] has been used to address the coarse-grain problem of Hamming ranking. With a code length of m, Hamming ranking can classify the buckets into $(m + 1)$ categories according to their Hamming distance from a query. Thus, longer code provides better ability to distinguish the buckets, so that the evaluation is more likely to be conducted on favorable buckets. To show that, precision, which is the ratio between the number of returned true k-nearest neighbors and the number of retrieved items, is plotted against recall in Figure 4a. It can be observed that precision increases with code length for the same recall. However, long code brings other problems, as shown in Figure 4b, the efficiency decreases with code length because it is more time-consuming to process long codes and decide which bucket to probe (i.e., the retrieval complexity increases). Moreover, long code also means more memory consumption, which can be a problem when multiple hash tales are used.

To summarize, the problem of Hamming ranking is inherently in its similarity indicator, i.e., Hamming distance. For better query performance and lower memory consumption, a new similarity indicator is needed and two requirements should be satisfied. First, the new indicator needs to provide useful information about the similarity between the query and items in a bucket, and be able to distinguish buckets having the same Hamming distance. Second, the indicator should have nice properties that enables the design of an efficient retrieval algorithm.

4 QUANTIZATION DISTANCE RANKING

In this section, we begin by introducing **quantization distance** (**QD**), and show by analysis that it is a better similarity indicator than Hamming distance. Using QD, the **QD ranking** algorithm is then formulated. Without loss of generality, we assume that there is only one hash table, and extending QD ranking to multiple hash tables is straightforward. We conduct our analysis on Euclidean distance but other similarity metrics such as angular distance can also be adapted with some modifications.

4.1 Quantization Distance

To address the problems of Hamming distance and preserve more information about similarity, we measure the difference between the projected vector of a query and the binary code of a bucket, which motivates the definition of quantization distance as follows.

Definition 1. (Quantization Distance) The *quantization distance* between query q and bucket b is given as

$$dist(q,b) = \sum_{i=1}^{m} (c_i(q) \oplus b_i) * |p_i(q)|.$$

In the definition above, $c_i(q)$ and b_i are the i^{th} bit of the binary codes for query q and bucket b, respectively; $p_i(q)$ is the i^{th} entry of q's projected vector; and $\oplus$ is the exclusive-or operation. Intuitively, QD measures the minimum change that we need to make to the projected vector $p(q)$ such that it can be quantized to bucket b. When $c_i(q) = b_i$, no change is needed for $p_i(q)$. When $c_i(q) \neq b_i$, we need to add or deduct $|p_i(q)|$ to/from $p_i(q)$ to ensure that $p_i(q)$ can be quantized to b_i. As an illustration, consider q_1 in Figure 3, where $p(q_1)$ =$(-0.2, -0.8)$. The QD and Hamming distance of the buckets from q_1 are shown in Figure 3b. While Hamming distance cannot distinguish bucket $(0, 1)$ from $(1, 0)$, QD can identify that bucket $(1, 0)$ should be probed before bucket $(0, 1)$ for q_1.

The advantages of QD over Hamming distance are two folds. Firstly, using projected vector rather than binary code, QD preserves more information about similarity. When two bits in the binary codes (of the query and bucket) are different, Hamming distance only returns 1 but does not measure how large the difference is. In contrast, QD views each different position in the binary codes differently and measures the minimum change required to make the bits identical. Secondly, as a real number rather than an integer, QD has the ability to distinguish between buckets with the same Hamming distance from a query. With a code length of m, Hamming distance can only classify the buckets into $(m + 1)$ categories while QD could classify the buckets into up to 2^m categories, thus significantly reducing buckets with the same distance from the query.

In the following, we show by theoretical analysis that QD is a good similarity indicator.

Theorem 1. (The projection operation of L2H is bounded) Organizing the hash vectors learned by L2H into a hashing matrix H, the projection operation can be expressed by $p(q) = Hq$. There exists a constant M, such that for any $q \in R^d$, we have $\|Hq\|_2 \leqslant M\|q\|_2$.

Each hash vector is a d-dimensional column vector, that is, $h_i \in R^d$ for $1 \leq i \leq m$. With h_i^T as its rows, the hashing matrix, $H \in R^{m\times d}$ can be expressed as $H = [h_1, h_2, \cdots, h_m]^T$. Theorem 1 follows directly from the definition of the spectral norm of matrix [49] with $M = \sigma_{\max}(H)$, where $\sigma_{\max}(H)$ is the largest singular value of H. A direct corollary of Theorem 1 is that the projection operation in L2H is similarity preserving, that is, two items close to each other in the original d-dimensional space also have their projected vectors close to each other, as stated in Corollary 1 below.

Corollary 1. For two items q and o, we have $\|p(q) - p(o)\|_2 \leq M\|q - o\|_2$.

The proof of Corollary 1 is straightforward, as $\|p(q) - p(o)\|_2 = \|H(q - o)\|_2 \leq M\|q - o\|_2$. Corollary 1 shows that the distance between the projected vectors gives a (scaled) lower bound on the distances of two items in the original d-dimensional space. However, the projected vectors of the items are not stored in the hash tables to reduce memory consumption. Thus, we need to connect Corollary 1 to the quantization rule and rewrite the bound using the binary code, from which we obtain Theorem 2 as follows.

Theorem 2. (QD gives a (scaled) lower bound of the distance between a query and the items in a bucket) Denote the QD between query q and bucket b as $dist(q, b)$, there exists a constant μ, such that for any item o in bucket b, we have $\|o - q\|_2 \geq \mu \cdot \text{dist}(q, b)$.

Proof.

$$\begin{aligned}
\|o - q\|_2 &\geq \frac{1}{M}\|p(o) - p(q)\|_2 \\
&\geq \frac{1}{M\sqrt{m}}\|p(o) - p(q)\|_1 \\
&\geq \frac{1}{M\sqrt{m}}\sum_{i=1}^{m}(c_i(q) \oplus c_i(o)) * |p_i(q)| \\
&= \frac{1}{M\sqrt{m}}\sum_{i=1}^{m}(c_i(q) \oplus b_i) * |p_i(q)|)
\end{aligned}$$

□

The first inequality follows from Corollary 1. The second inequality holds as for any m dimensional vector x, we have $\sqrt{m}\|x\|_2 \geq \|x\|_1$ [49]. The last equality holds because item o has the same binary code as b, as it is in bucket b. The third inequality holds because of the quantization rule and is explained as follows. When $c_i(q) = c_i(o)$, the term $|p_i(o) - p_i(q)| \geq 0$ and it is included in the $l_1 - norm$ in Line 2 of the proof but not included in the summation of Line 3. When $c_i(q) \neq c_i(o)$, $p_i(o)$ and $p_i(q)$ have different signs, which means $|p_i(o) - p_i(q)| = |p_i(o)| + |p_i(q)| \geq |p_i(q)|$. Thus, we have $\mu = \frac{1}{M\sqrt{m}}$.

From Theorem 2, we know that $(\mu \cdot \text{dist}(q, b))$ provides a lower bound of the distance between a query and the items in a bucket. Since μ is just a scaling factor irrelevant to either the query or the bucket, we can drop it and use QD directly as the similarity indicator. It is reasonable to probe buckets with a smaller lower bound first, which is the idea behind the QD ranking algorithm in the next subsection. Moreover, QD can also be used as a criteria for early stop. If we are only interested in finding items within a certain distance to the query, retrieval and evaluation can stop when all buckets with a QD smaller than the corresponding threshold are probed. We can also keep a record of the distance between the query and the k-th nearest neighbor in the evaluated items as d_k, if all remaining buckets have $\mu \cdot \text{dist}(q, b) \geq d_k$, probing can be stopped.

4.2 The QD Ranking Algorithm

We have analyzed that QD is a good indicator of similarity. Using QD, our QD ranking algorithm is presented in Algorithm 1.

Algorithm 1 QD Ranking

```
Input: query $q$, and the number of candidates to collect $N$
Output: approximate $k$-nearest neighbors to $q$
 1: Set candidate set $C = \emptyset$
 2: Count the total number of buckets $M$
 3: Compute the projected vector $p(q)$ and binary code $c(q)$ of $q$
 4: Calculate QD for all buckets and sort the buckets with QD
 5: $i = 1$
 6: while $|C| < N$ and $i \leq M$ do
 7:     Find the $i$-th bucket in the sorted order and collect items in it to $C$
 8:     $i = i + 1$
 9: end while
10: re-rank the items in $C$ by their Euclidean distances to $q$ in ascending order
11: return the top-$k$ items
```

QD ranking uses QD to define a linear order for the buckets and probe the buckets with small QD first. In Algorithm 1, we set the number of items to evaluate as N and use it as the stopping criteria, but other stopping criteria can also be used, such as probing a certain number of buckets, after a period of time or early stop explained in the previous subsection. The re-ranking in Line 10 can be conducted only once after all items are collected or multiple times when probing each bucket. The latter supports early stop as we can keep a record of the current k-th nearest neighbor. The sorting process of QD ranking has a higher complexity than that of Hamming ranking, since Hamming ranking can use efficient bucket sort. However, we will show in the experiments that QD ranking benefits from the fine-grained similarity indicator it uses and hence provides superior performance.

For many approximate k-nearest neighbor search applications, the query delay is an important concern and there is usually a time limit within which results should be returned. In this case, only a small number of buckets may be probed for fast response. For a specific query, a large fraction of the buckets is irrelevant as they will not be probed, thus calculating QD and conducting sorting for them are wasteful. We call this the *slow start* problem of the QD ranking algorithm, in which sorting all the buckets takes a long time and evaluation can only start after sorting. Slow start may result in poor performance as the time spent on sorting can be dominant when only a small number of buckets are actually probed. In the next section we propose a solution to generate the buckets to probe on demand, so that sorting of unnecessary buckets is avoided.

5 GENERATE TO PROBE

To solve the slow start problem of the QD ranking algorithm, we propose an efficient **generate-to-probe QD ranking** (**GQR**) algorithm. Designing a generate-to-probe algorithm is challenging as the linear orders of the buckets w.r.t. different queries vary significantly. We first present a framework of the GQR algorithm, and then introduce the concept of *flipping vector*, which is key to our algorithm design. Lastly, we prove the correctness and improve the efficiency of the GQR algorithm by leveraging the properties of QD.

5.1 Algorithm Framework

We first give the framework of the GQR algorithm in Algorithm 2. GQR keeps generating buckets to probe until the required number of candidates (N) are collected or all the buckets are probed. Different from the QD ranking algorithm, GQR does not sort all the buckets according to their QD at the beginning. Instead, GQR invokes the *generate_bucket*() function whenever it needs a bucket to probe.

Algorithm 2 Generate-to-Probe QD Ranking

```
Input: query $q$, code length $m$, number of candidates to collect $N$
Output: approximate $k$-nearest neighbors of $q$
 1: Set candidate set $C = \emptyset$
 2: Compute the binary code $c(q)$ and projected vector $p(q)$ of $q$
 3: $i = 1$
 4: while $|C| < N$ and $i \leq 2^m$ do
 5:     $b = generate\_bucket(c(q),\ p(q),\ i,\ m)$
 6:     Collect items in bucket $b$ into $C$
 7:     $i++$
 8: end while
 9: re-rank the items in $C$ by their Euclidean distances to $q$ in ascending order
10: return the top-$k$ items
```

The *generate_bucket*() function is key to the correctness and efficiency of GQR. For correctness, the function needs to fulfill two requirements: **(R1)** each bucket is generated at most once given any N and each bucket is generated at least once when N is sufficiently large; and **(R2)** the bucket b produced by *generate_bucket*() in the i^{th} iteration is indeed the bucket with the i^{th} smallest QD. (R1) ensures that GQR can probe all the buckets and does not probe the same bucket multiple times. (R2) guarantees that GQR probes the buckets in ascending order of their QD. If both (R1) and (R2) are fulfilled, GQR will be equivalent to QD ranking in semantics. For the efficiency of GQR, *generate_bucket*() should produce buckets with low complexity.

5.2 Flipping Vector

The main idea of *generate_bucket*() is to manipulate the binary code $c(q)$ to generate the buckets to probe without sorting. We achieve this using *flipping vector*, which is defined as follows.

Definition 2. (Flipping Vector) Under a code length of m, for a given query q, the flipping vector of bucket b is an m dimensional binary vector $v = (v_1, v_2, ..., v_m)$ with $v_i = c_i(q) \oplus b_i$ for $1 \leq i \leq m$.

As the $\oplus$ operation is reflexive, given a flipping vector v, its corresponding bucket b can be calculated by $b_i = c_i(q) \oplus v_i$. We call v a **flipping vector** as the resultant bucket is calculated by flipping the entries in $c(q)$ that corresponds to the non-zero elements in v. For a given query, flipping vector v is uniquely determined by bucket b and vice versa. Recall the expression of QD in Definition 1, we can rewrite it as follows.

$$\begin{aligned} dist(q,b) &= \sum_{i=1}^{m} (c_i(q) \oplus b_i) * |p_i(q)|. \\ &= \sum_{i=1}^{m} v_i * |p_i(q)|. \end{aligned} \tag{1}$$

The above equation shows that the QD of a bucket can be calculated without actually knowing the binary code of the bucket, i.e., knowing $p(q)$ and flipping vector v is sufficient. We define $dist(q,v) = \sum_{i=1}^{m} v_i * |p_i(q)|$ as the QD of flipping vector v (w.r.t. q), and Theorem 3 shows the relationship between $dist(q,v)$ and $dist(q,b)$.

Theorem 3. (Quantization distance of flipping vector) Given a query q, for a flipping vector v and its corresponding bucket b, we have $dist(q,v) = dist(q,b)$.

Theorem 3 follows from the definition of QD and flipping vector. Since both QD and bucket can be derived from flipping vector, we focus on how to generate the flipping vectors in ascending order of their QDs. Intuitively, the all-zero flipping vector has the smallest QD. For the flipping vector with the second smallest QD, we need to set the bit corresponding to the entry with the smallest absolute value in projected vector $p_i(q)$ to 1 while keeping other bits 0. As the QD is calculated using the absolute values of the entires of the projected vector and the order of these absolute values matters, we introduce the definition of sorted projected vector as follows.

Definition 3. (Sorted Projected Vector) For a query q, its sorted projected vector $\tilde{p}(q)$ is obtained by sorting the absolute values of the elements in its projected vector $p(q)$. In other words, we sort $p(q)$=($|p_1(q)|$, $|p_2(q)|$, ..., $|p_m(q)|$) to obtain the sorted projected vector $\tilde{p}(q)$.

The sorting process defines a one-to-one mapping between the index of an entry in the projected vector and that in the sorted projected vector. We denote this mapping by $y = f(x)$, where y is the index of an entry in the projected vector and x is the index of the corresponding entry in the sorted projected vector. Consider a projected vector p=(0.6, 0.2, −0.4), its sorted projected vector is $\tilde{p} = (0.2, 0.4, 0.6)$, we have $f(1) = 2$, $f(2) = 3$ and $f(3) = 1$.

We use sorted projected vector because generating flipping vectors from it is more efficient. Instead of generating flipping vectors v directly, we generate **sorted flipping vector**, $\tilde{v}$, which is a binary vector indicating the entries to flip in the sorted projected vector. Given a sorted flipping vector, the corresponding bucket can be obtained by Algorithm 3.

Algorithm 3 Sorted Flipping Vector to Bucket

```
Input: sorted flipping vector $\tilde{v}$, binary code $c(q)$, code length $m$
Output: bucket to probe $b$
1: $b = c(q)$, $i = 1$
2: while $i <= m$ do
3:     if $\tilde{v}_i == 1$ then
4:         $l$=$f(i)$
5:         $b_l = 1 - c_l(q)$
6:     end if
7:     $i = i + 1$
8: end while
9: return $b$
```

In Algorithm 3, when $\tilde{v}_i = 1$, we first find the index of the entry in the original projected vector by l=$f(i)$, and then flip the l-th entry of $c(q)$ to obtain the bucket. We can see that the QD of the resultant bucket is $dist(q,\tilde{v}) = \sum_{i=1}^{m} \tilde{v}_i * |\tilde{p}_i(q)|$.

When query q is clear from the context, we simply denote the QD of flipping vector v from q as $dist(v)$, and that of sorted flipping vector $\tilde{v}$ from q as $dist(\tilde{v})$. In the next subsection, we show that sorted flipping vector has nice properties that lead to the design of an efficient *generate_bucket*() algorithm.

5.3 Bucket Generation

With the definitions of sorted projected vector and sorted flipping vector, we present the *generate_bucket*() algorithm in Algorithm 4. We use a min-heap, h_{min}, to generate sorted flipping vectors when needed. Each element in h_{min} is a tuple $(\tilde{v}, dist(\tilde{v}))$, where $dist(\tilde{v})$ is the QD of sorted projected vector $\tilde{v}$. The top of the min-heap is the tuple with the smallest $dist(\tilde{v})$. When the function is called for the first time, we initialize h_{min} with $\tilde{v}^r = (1, 0, ..., 0)$. When we need a bucket to probe, we dequeue the top element, $\tilde{v}$, from h_{min}, and use it to construct the bucket as described in Algorithm 3. We also apply the *Swap* and *Append* operations (which will be explained later) on $\tilde{v}$ to generate two new tuples and insert them into h_{min}. Note that we need to probe bucket $c(q)$ at the very beginning, and Line 3 in Algorithm 4 is used to handle this special case.

Algorithm 4 *generate_bucket*($c(q)$, $\tilde{p}(q)$, i, m)

```
Input: binary code $c(q)$, sorted projected vector $\tilde{p}(q)$, iteration count $i$, code length $m$
Output: the next bucket to probe $b$
1: if $i == 1$ then
2:     $h_{min}.insert(\tilde{v}^r = (1, 0, ..., 0), |\tilde{p}_1(q)|)$
3:     $\tilde{v} = (0, 0, ..., 0)$
4: else
5:     $\tilde{v} = h_{min}.del\_min()$
6:     Set $j$ as the index of the rightmost 1 in $\tilde{v}$
7:     if $j < m$ then
8:         $\tilde{v}^+ = Append(\tilde{v})$
9:         $h_{min}.insert(\tilde{v}^+, dist(\tilde{v}) + \tilde{p}_{j+1}(q))$
10:        $\tilde{v}^- = Swap(\tilde{v})$
11:        $h_{min}.insert(\tilde{v}^-, dist(\tilde{v}) + \tilde{p}_{j+1}(q) - \tilde{p}_j(q))$
12:    end if
13: end if
14: Use $\tilde{v}$ to calculate the bucket $b$ to probe by Algorithm 3
15: Return the bucket $b$
```

The *Swap* and *Append* operations on a sorted flipping vector $\tilde{v}$ are key to Algorithm 4 in fulfilling (R1) and (R2) stated in Section 5.1. We give their definitions as follows.

Append. If the index of the rightmost non-zero entry of $\tilde{v}$ is i and $i < m$, *Append* returns a new sorted flipping vector $\tilde{v}^+$, by assigning $\tilde{v}^+ = \tilde{v}$, and $\tilde{v}^+_{i+1} = 1$.

Swap. If the index of the rightmost non-zero entry of $\tilde{v}$ is i and $i < m$, *Swap* returns a new sorted flipping vector $\tilde{v}^-$, by assigning $\tilde{v}^- = \tilde{v}$, $\tilde{v}^-_i = 0$, and $\tilde{v}^-_{i+1} = 1$.

Intuitively, *Append* adds a new non-zero entry to the right-hand side of the rightmost 1 in $\tilde{v}$, while *Swap* exchanges the positions of the rightmost 1 and the 0 on its right-hand side. Note that *Append* and *Swap* are only *valid* when the index of the rightmost non-zero entry of $\tilde{v}$ is smaller than m. Starting with $\tilde{v}^r = (1, 0, ..., 0)$, an

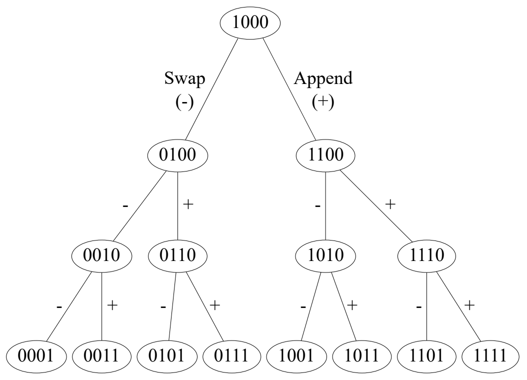


Figure 5: An example of generation tree

example for *Append* and *Swap* with a code length of 4 is given in Figure 5.

Let $\tilde{V}$ be the set of all flipping vectors with length m, $\tilde{v}^r = (1, 0, ..., 0)$ and $\tilde{v}^0 = (0, 0, ..., 0)$. To better understand the properties of *Append* and *Swap*, we define the *generation tree* as follows.

Definition 4. (Generation Tree) The *generation tree* is a rooted binary tree with $\tilde{v}^r$ as the root, and the two edges of each internal node correspond to the *Append* and *Swap* operations, such that the right child of an internal node is generated by applying *Append* on the node, while the left child is generated by applying *Swap* on the node.

An example of generation tree is given in Figure 5, and it has the following properties which are crucial to the efficiency and correctness of Algorithm 2.

Property 1. (Exactly once) Let T be the generation tree with root $\tilde{v}^r$. Each flipping vector $\tilde{v} \in \tilde{V} \setminus \{\tilde{v}^0\}$ appears exactly once in T.

Proof. For an arbitrary vector $\tilde{v} \in \tilde{V} \setminus \{\tilde{v}^0, \tilde{v}^r\}$, by the definitions of *Append* and *Swap*, we can derive a unique sequence of operations on $\tilde{v}^r$ to generate it as follows. If $\tilde{v}_i = 0$ and there is a non-zero entry to the right of $\tilde{v}_i$, then *Swap* is applied in the i-th step of generating $\tilde{v}$, i.e., *Swap* is in the i-th position of the sequence. If $\tilde{v}_i = 1$ and i is not the rightmost non-zero entry, then *Append* is applied in the i-th step of generating $\tilde{v}$. For example, $\tilde{v} = (1, 1, 0, 1)$ in Figure 5 is generated by (*Append*,*Append*,*Swap*) and $\tilde{v} = (1, 0, 1, 0)$ is generated by (*Append*,*Swap*). Thus, for each $\tilde{v} \in \tilde{V} \setminus \{\tilde{v}^0, \tilde{v}^r\}$, we have a unique sequence of *Append* and *Swap* that generates $\tilde{v}$ from $\tilde{v}^r$, which corresponds to a unique path in T from $\tilde{v}^r$ to a non-root node, and hence each flipping vector in $\tilde{V} \setminus \{\tilde{v}^0\}$ appears exactly once in T. □

Property 1 shows that all possible sorted flipping vectors can be obtained from the generation tree exactly once. The relationship among the QDs of an internal node in the generation tree and its two children is stated in Property 2.

Property 2. (Quantization distance order) Let $\tilde{v}$ be an internal node in the generation tree, and $\tilde{v}^+$ and $\tilde{v}^-$ be the right and left child of $\tilde{v}$, i.e., $\tilde{v}^+ = Append(\tilde{v})$ and $\tilde{v}^- = Swap(\tilde{v})$. We have $dist(\tilde{v}) \leq dist(\tilde{v}^+)$ and $dist(\tilde{v}) \leq dist(\tilde{v}^-)$.

Proof. According to the definition of QD and assuming that the index of the rightmost non-zero entry in $\tilde{v}$ is i (note that $i < m$ since $\tilde{v}$ is an internal node), we have
$dist(\tilde{v}^+) - dist(\tilde{v}) = \tilde{p}_{i+1}(q) \geq 0$ and
$dist(\tilde{v}^-) - dist(\tilde{v}) = \tilde{p}_{i+1}(q) - \tilde{p}_i(q) \geq 0.$ □

Property 2 shows that in the generation tree, the QD of a child is no smaller than its parent. The proof also explains why we can calculate the QD of newly generated sorted flipping vector by $dist(\tilde{v}^+) = dist(\tilde{v}) + \tilde{p}_{i+1}(q)$ and $dist(\tilde{v}^-) = dist(\tilde{v}) + \tilde{p}_{i+1}(q) - \tilde{p}_i(q)$ in Line 9 and Line 11 of Algorithm 4.

Now we prove the correctness of Algorithm 2. To ensure the correctness of Algorithm 2, Algorithm 4 should produce the buckets in the order of their QD and generate each bucket exactly once. First, we prove that Algorithm 4 generates each bucket exactly once for sufficiently large N. While this follows directly from Property 1, another proof can also be provided. One observation is that each $\tilde{v} \in \tilde{V} \setminus \{\tilde{v}^0\}$ has only one parent in the generation tree and is inserted into h_{min} only when its parent is dequeued from h_{min}. As $\tilde{v}$ is dequeued from h_{min} after all its ancestors according to Property 2, thus $\tilde{v}$ will not be inserted into h_{min} again and each flipping vector $\tilde{v}$ is generated at most once by Algorithm 4. Moreover, Property 1 also ensures that each flipping vector $\tilde{v}$ in $\tilde{V} \setminus \{\tilde{v}^0\}$ is generated at least once for sufficiently large N. Since the all-zero flipping vector is handled in Line 3 of Algorithm 4, we can conclude that Algorithm 4 generates each flipping vector exactly once for sufficiently large N.

Second, we prove that when $i = k$, the vector $\tilde{v}^k$ generated by Algorithm 4 is indeed the flipping vector having the k-th smallest QD from q. For a $\tilde{v}^j$ dequeued from h_{min} before $\tilde{v}^k$, there are two cases. Case 1 is that $\tilde{v}^k$ is in h_{min} when $\tilde{v}^j$ is dequeued, implying that $dist(\tilde{v}^k) \geq dist(\tilde{v}^j)$. Case 2 is that $\tilde{v}^k$ is not in h_{min} when $\tilde{v}^j$ is dequeued, which implies that one ancestor of $\tilde{v}^k$ must be in h_{min}, otherwise $\tilde{v}^k$ cannot be generated later. Denote that ancestor as $\tilde{v}'$. According to Property 2 and the property of min-heap, we have $dist(\tilde{v}^k) \geq dist(\tilde{v}') \geq dist(\tilde{v}^j)$. Therefore, we can conclude that for any $\tilde{v}^j$ dequeued before $\tilde{v}^k$, $dist(\tilde{v}^k) \geq dist(\tilde{v}^j)$ holds. Using a similar analysis, we can also conclude that for any $\tilde{v}^j$ dequeued after $\tilde{v}^k$, $dist(\tilde{v}^j) \geq dist(\tilde{v}^k)$. This completes the proof of the correctness of Algorithm 4. The algorithm generates a flipping vector only when needed and calculates the QD of a child efficiently based on that of its parent.

The memory consumption of Algorithm 2 is also significantly lower than that of Algorithm 1 and Hamming ranking, as we do not need to store the QD of all buckets. Another nice property is that there are at most i elements in h_{min} for Algorithm 4 at iteration i. In most cases, both i and the size of the heap are small in the entire querying process. Note that the generation tree is common to all queries, we can code a flipping vector as an integer and store the generation tree (the corresponding integers) into an array. In this case, we do not need to actually conduct the *Append* and *Swap* operations, as fetching the corresponding elements from the array is more efficient.

We remark that QD and GQR are inspired to some extent by Multi-Probe LSH [28]. Beside the fact that our methods work for L2H with binary code and Multi-Probe LSH works for LSH with integer code, there are three more fundamental differences. The

Table 1: Statistics of datasets and linear search in seconds

Dataset	Dim#	Item#	Linear Search
CIFAR60K	512	60,000	31.34s
GIST1M	960	1,000,000	999.75s
TINY5M	384	5,000,000	1978.19s
SIFT10M	128	10,000,000	1247.07s

first difference lies in the definition of the distance metric. The *score* in Multi-Probe LSH is the sum of the squared difference between the projected value of a query and a bucket. In contrast, QD uses the sum of the absolute differences and introduces an additional exclusive-or operation to exclude the contribution (to flipping cost) of identical bits. Secondly, QD is also more general. The *score* indicates the similarity between the query and items in a bucket only when the hash vector follows Gaussian distribution. However, QD can provide a distance lower bound as long as the hash functions can be written in a matrix form. Thirdly, GQR can use a shared *generation tree* to optimize the querying process, but Multi-Probe LSH can not adopt such optimization as it works with integer code. Note that Multi-Probe LSH will also generate invalid buckets but this problem does not exist for GQR.

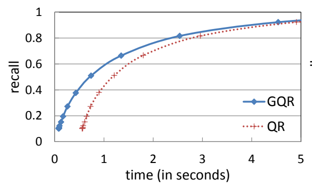
6 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our algorithms, QD ranking in Algorithm 1 and generate-to-probe QD ranking in Algorithm 2, denoted by **QR** and **GQR**, respectively. We compared our algorithms with the state-of-the-art methods under various settings. For fair comparison, we tuned other algorithms to achieve their best performance. All experiments were conducted using a machine with 64 GB Ram and a 2.10 GHz Intel Xeon E5-2620 Processor, running CentOS 6.5. All the querying algorithms and learning algorithms were implemented in C++ and MATLAB, respectively. We used MATLAB 9.1 and gcc 5.2.0 to compile the related source codes. We release all the source codes we implemented in `www.cse.cuhk.edu.hk/systems/hash/gqr`, while the source codes of other projects can be found in their websites.

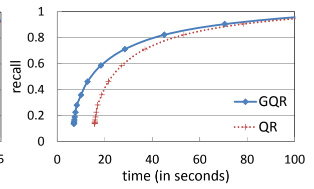
6.1 Datasets and Settings

We used four datasets with different sizes and number of features, as shown in Table 1. More results with 8 additional datasets are reported in the Appendix due to space limitation. CIFAR-10 [2] (denoted by CIFAR60K) is a small-scale dataset used as benchmark in many ANN search evaluations [9, 17, 20, 23, 25, 26]. Following the common practice in these work, we extracted a 512 dimensional GIST descriptor for each of the 60, 000 CIFAR-10 images and the GIST descriptors are treated as items in the dataset. For a middle-scale dataset, we used GIST1M [3], which contains one million vectors, each with 960 features. For large-scale datasets, we downloaded 5 million GIST descriptors of TINY images[4] with a dimensionality of 384, denoted as TINY5M. We also used SIFT10M[5], which contains 10 million SIFT descriptors with a dimensionality of 128.

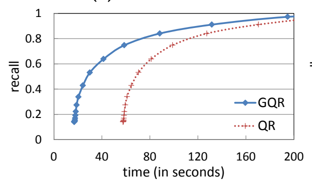
[2]https://www.cs.toronto.edu/ kriz/cifar.html
[3]http://corpus-texmex.irisa.fr/
[4]http://horatio.cs.nyu.edu/mit/tiny/data/
[5]http://corpus-texmex.irisa.fr/



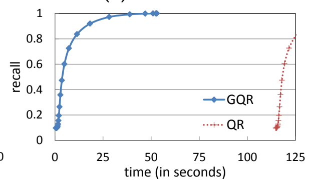
(a) CIFAR60K



(b) GIST1M



(c) TINY5M



(d) SIFT10M

Figure 6: Comparison between GQR and QR

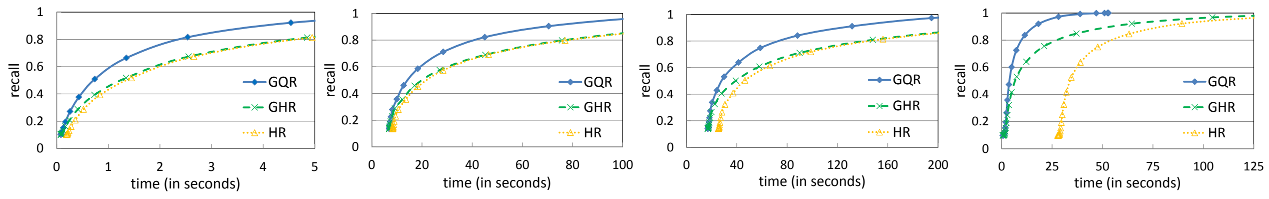
The hash functions were trained using ITQ [8, 9] unless otherwise stated, and kept the same for all querying methods. By default, we report the performance of 20-nearest neighbors search. For each dataset and querying method, we randomly sampled 1000 items as queries and measured the overall query processing time. We report the average recall or the number of items retrieved to evaluate query performance. We also present the running time of brute-force linear search as a benchmark in the last column of Table 1.

Code length. For L2H algorithms, such as ITQ [8, 9], SH [44] and PCAH [8, 43], a proper code length m is crucial for performance. If the code is too short, a querying method may lack enough information to distinguish favorable buckets from unfavorable ones and end up evaluating many irrelevant items. If the code is too long, the number of buckets will be large (equal to the number of items eventually) and *retrieval* will take excessive memory and time to sort the buckets. Following the methodology in [16] and [4], we fix the expected number of items inside a bucket, EP, to calculate the code length. Specifically, similar to [16] and [4], we set EP to 10 in our experiments, and m was set to an integer around $log_2(N/10)$, where N is the number of items in the dataset. Therefore, the default code length is 12, 16, 18 and 20 for CIFAR60K, GIST1M, TINY5M and SIFT10M, respectively. We will also experimentally verify that the above settings are almost optimal in the following subsections.

6.2 Comparison between QR and GQR

In this experiment, we compared the performance of *QR* and *GQR*. Both *QR* and *GQR* probe buckets according to their QD; thus, they probe buckets in the same order. Their difference lies in how the buckets are generated, i.e., *QR* calculates QD for all buckets and sorts them, while *GQR* generates the buckets on demand.

Figure 6 shows that *GQR* consistently outperforms *QR* on all four datasets. The performance gap is smaller for CIFAR60K but widens for larger datasets. For SIFT10M, the performance gap is substantial as reported in Figure 6d. This can be explained by the fact that *GQR* does not need to sort all the buckets at the start, thus is immune to the slow start problem of *QR*. In our experiment, CIFAR60K, GIST1M, TINY5M and SIFT10M have 3, 872, 57, 835, 218, 004 and 567, 753 buckets, respectively, and sorting these buckets takes 0.48,



(a) CIFAR60K (b) GIST1M (c) TINY5M (d) SIFT10M

Figure 7: Comparison between GQR and HR/GHR (best viewed in color)

8.82, 53.9 and 114.1 seconds. In many cases, the time spent on sorting is non-negligible, and thus *GQR* outperforms *QR*. For larger datasets, there are more buckets, and hence the performance gap widens with dataset size. However, the performance gap narrows at very high recall (near 100%), as a large portion of buckets needs to be probed and the time that *GQR* spends on generate-to-probe also increases and gradually approaches that of sorting.

As *GQR* consistently outperforms *QR*, we only compare *GQR* with existing querying methods in the following experiments.

6.3 Comparison between GQR and HR

In this experiment, we compared the performance of *GQR* with Hamming ranking, *HR*, which is extensively used in existing L2H work [4, 42]. We also implemented another algorithm named *generate-to-probe Hamming ranking* (*GHR* or hash lookup as commonly called) using an idea similar to *GQR*, i.e., the buckets to probe are generated in ascending order of their Hamming distances by manipulating the binary code of the query rather than sorting.

6.3.1 Recall-time. The recall-time curves for *GQR*, *GHR* and *HR* are plotted in Figure 7 for the four datasets. The result shows that *GHR* and *HR* have similar performance trade-off as *GQR* and *QR*, i.e., *GHR* consistently outperforms *HR* and the performance gap widens for large datasets, which proves again that sorting all buckets at the beginning will cause the slow start problem. However, the performance gained by generate-to-probe on *HR* is not as significant as on *QR*, because Hamming distance only takes discrete values and sorting is more efficient.

The result also shows that *GQR* achieves superior performance than both *HR* and *GHR* for all datasets, which can be explained by the fact that QD is a better indicator of similarity than Hamming distance. To show this, we plot the recall versus the number of retrieved items in Figure 8, which shows that for the same number of retrieved items, *GQR* always finds more true k-nearest neighbors than *HR* and *GHR* on all datasets. This demonstrates that QD can direct the evaluation to the more favorable (high quality) buckets, leading to higher efficiency. Moreover, the quality gap between the buckets generated by *GQR* and *GHR*/*HR* widens with the size of dataset. This can be explained by the fact that hamming distance can classify the buckets into only $m + 1$ categories, while QD can classify the buckets into up to 2^m categories. Larger datasets have longer code and the advantage of making a finer classification of the buckets is more obvious, i.e., favorable buckets can be more easily distinguished from unfavorable ones.

6.3.2 Querying Time. The recall gap between *GQR*, *GHR* and *HR* may seem moderate in Figure 7, but the speedup in querying time is actually significant as the recall-time curve is flat at high recall and a small improvement in recall may translate into a large time saving. To show this, we compare the running time of *GQR* with *HR* and *GHR* at a number of typical recalls reported in the experiments of existing work [28–30] in Figure 9. *GQR* has a minimum speedup of 1.6 over *HR* and *GHR*, and the speedup can be as much as 3 times, which means that *GQR* can achieve the same recall as *GHR* and *HR* with only half of the time in most cases.

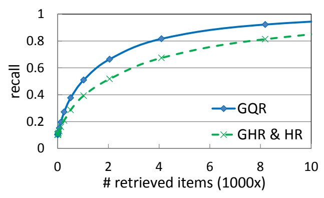
6.3.3 Effect of Code Length. In the previous experiments, we set the code length to an integer around $log_2(N/10)$. To show that there is no bias in our choice of code length, we tested different code lengths and plot the time taken to reach 90% recall for the two larger datasets in Figure 10. The result shows that the previously chosen code lengths, 18 for TINY5M and 20 for SIFT10M, are very close to the optimal code length of *HR* and *GHR*. The performance of all algorithms first increases with code length but then decreases due to the trade-off between the retrieval and evaluation costs. However, even at the optimal code length for *GHR* and *HR*, *GQR* still outperforms them.

6.3.4 Effect of k. We also report the speedup of *GQR* and *GHR* over *HR* in running time to achieve 90% recall for different number of target nearest neighbors (k). Figure 11 shows that *GQR* is significantly faster than *HR* and *GHR* for a wide range of values of k. The speedup is more significant for small k, and can be as much as 8 times over *HR* and 3.4 times over *GHR*.

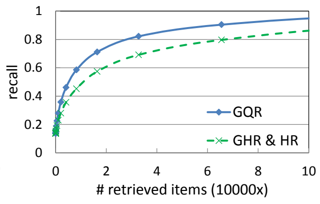
6.3.5 Effect of Multiple Hash Tables. Using multiple hash tables is a general strategy for improving the performance of similarity search [4, 16], but more memory space is needed. We demonstrate the performance advantage of *GQR* over *GHR* by comparing the performance of single-hash-table *GQR* with multi-hash-table *GHR*. Figure 12 shows that *GHR* needs about 30 hash tables to reach a similar performance to single-hash-table *GQR* on the SIFT10M dataset. However, *GHR* cannot reach the performance of *GQR* even with 30 hash tables on the TINY5M dataset. As using multiple hash tables incurs additional memory cost, using *GQR* instead of *GHR* can yield significant memory saving.

6.4 Compatibility with L2H Algorithms

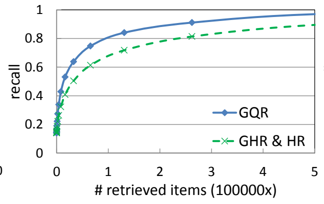
The experiments in the previous subsections were conducted using hash functions learnt by ITQ. However, there are many other L2H algorithms such as PCA hashing (PCAH) [8, 43] and spectral hashing



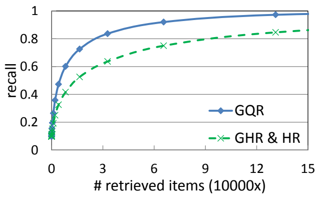
(a) CIFAR60K



(b) GIST1M

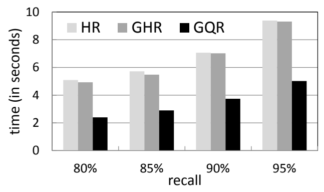


(c) TINY5M

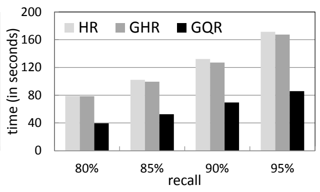


(d) SIFT10M

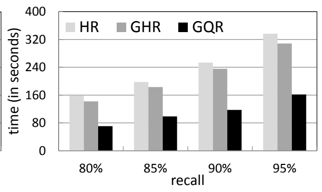
Figure 8: Recall-items curves



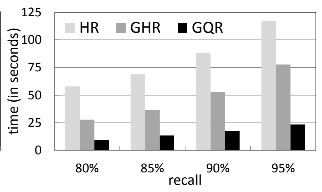
(a) CIFAR60K



(b) GIST1M

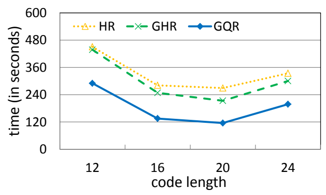


(c) TINY5M

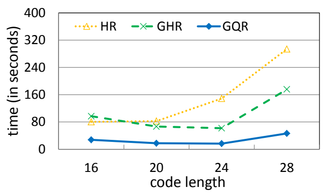


(d) SIFT10M

Figure 9: Querying time comparison between GQR and HR/GHR

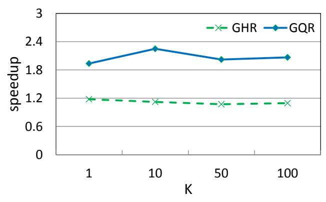


(a) TINY5M

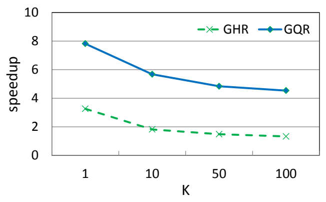


(b) SIFT10M

Figure 10: Effect of varying code length

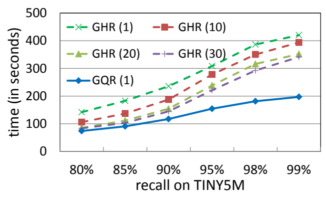


(a) TINY5M

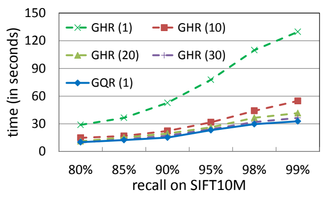


(b) SIFT10M

Figure 11: Speedup over HR for various *k*



(a) TINY5M



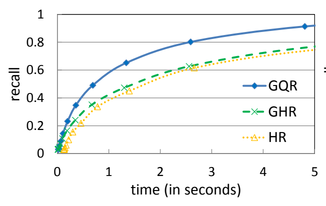
(b) SIFT10M

Figure 12: Effect of multiple hash tables

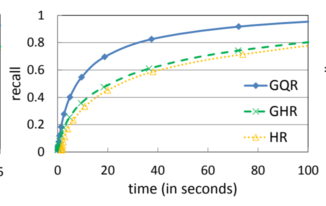
(SH) [44], which often show complicated performance trade-offs and different compatibility issues (e.g. datasets, dimensionality). A good querying method should be general enough to achieve good performance for a wide variety of L2H algorithms. In this experiment, we demonstrate the generality of *GQR* by reporting its performance with PCAH and SH. PCAH uses the eigenvectors of the covariance matrix of the dataset as hash functions, while SH uses the thresholded eigenvectors of the graph Laplacian as hash functions.

The recall-time curve for *GQR*, *GHR* and *HR* with PCAH is plotted in Figure 13. Similar to the cases of ITQ, *GQR* outperforms both *GHR* and *HR* consistently on the four datasets. We also plot the time taken to reach some typical recalls in Figure 14. The average speedups of *GQR* over GHR are 2.3, 2.8, 2.1 and 4.3 for CIFAR60K, GIST1M, TINY5M and SIFT10M, respectively. We plot the performance for SH in Figure 15 and Figure 16. The curves are quite similar to those of PCAH and ITQ. Note that there is also a trend that *GQR* achieves higher speedup on larger datasets.

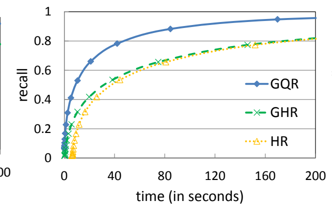
L2H algorithms have been extensively studied up to now, and to learn even better hash functions, new L2H algorithms are becoming increasingly complicated. However, complicated algorithms are expensive or even impractical for large datasets and their performance gains are sometimes only marginal. *GQR* is a general querying method that offers significant performance gains for a variety of L2H algorithms. Our experiments show that using *GQR* in combination with simple L2H algorithms may even outperform complicated L2H algorithms with *GHR*. For example, comparing Figure 7 and Figure 13, we can see that PCAH plus *GQR* offers better performance to ITQ plus *GHR*. However, ITQ adopts an iterative training process and is much more complex than PCAH. In cases that the complexity of hash function training is of concern, *GQR* can help by resolving to simple L2H algorithms.



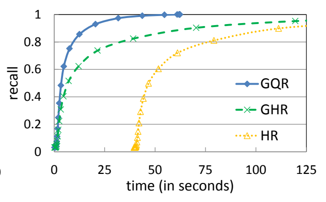
(a) CIFAR60K



(b) GIST1M

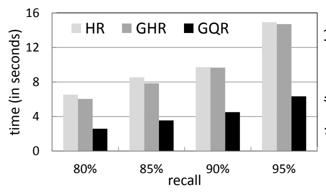


(c) TINY5M

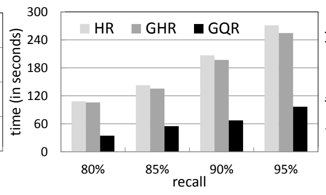


(d) SIFT10M

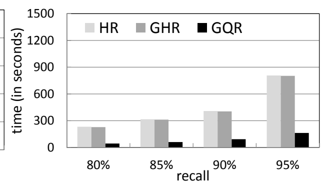
Figure 13: Comparison between GQR and HR/GHR with PCAH



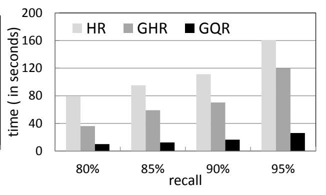
(a) CIFAR60K



(b) GIST1M

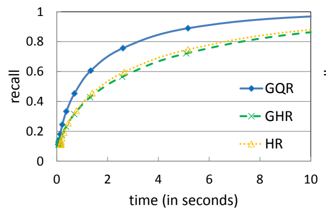


(c) TINY5M

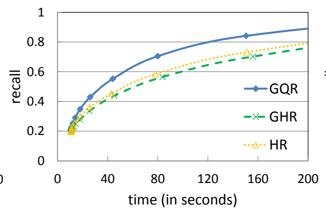


(d) SIFT10M

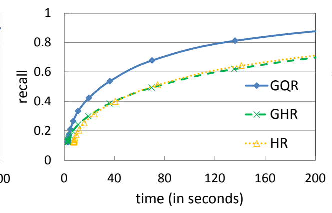
Figure 14: Querying time comparison between GQR and HR/GHR with PCAH



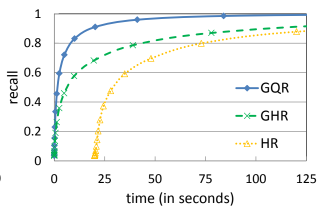
(a) CIFAR60K



(b) GIST1M

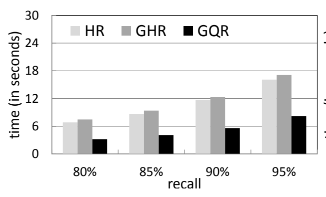


(c) TINY5M

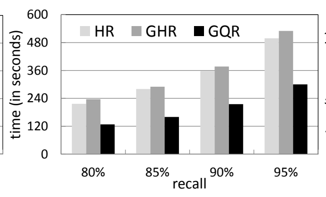


(d) SIFT10M

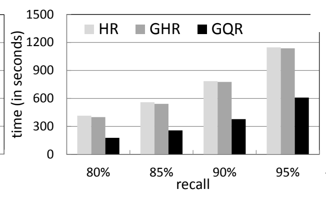
Figure 15: Comparison between GQR and HR/GHR with SH



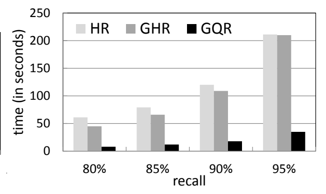
(a) CIFAR60K



(b) GIST1M

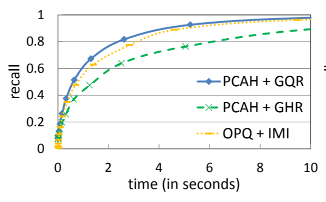


(c) TINY5M

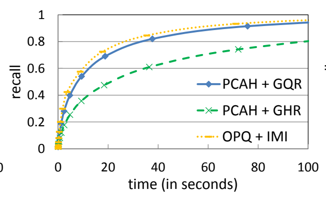


(d) SIFT10M

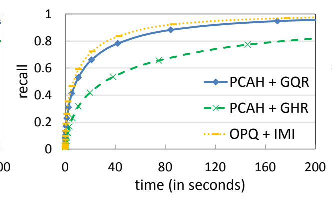
Figure 16: Querying time comparison between GQR and HR/GHR with SH



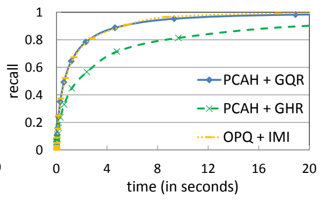
(a) CIFAR60K



(b) GIST1M



(c) TINY5M



(d) SIFT1M

Figure 17: Comparison between PCAH+HR, PCAH+GQR and OPQ+IMI

Table 2: Training cost

	Wall time (s)		CPU time (s)		Memory (GB)	
Dataset	OPQ	PCAH	OPQ	PCAH	OPQ	PCAH
CIFAR60K	43.9	12.0	295.6	12.4	1.2	1.0
GIST1M	2456.7	84.3	26166.7	153.8	23.1	15.5
TINY5M	10588.0	236.8	119391.0	464.5	54.5	30.4
SIFT1M	640.1	35.3	6096.8	40.0	4.7	2.5

6.5 Comparison with Vector Quantization

Vector quantization (VQ) based methods such as K-means hashing [11], OPQ [7] and CQ [50] were also proposed for similarity search, which represent the state-of-the-art similarity search methods. The VQ based methods usually learn one or more codebooks using K-means and quantize an item to its nearest codeword(s) in the codebook(s). Using a specially designed querying algorithm called Inverted Multi-Index (IMI) [1], the VQ based methods were reported to significantly outperform the L2H methods using Hamming ranking. In this experiment, we show that *GQR* can boost a binary hashing based L2H algorithm to achieve comparable query performance with OPQ, one of the state-of-the-art VQ based methods.

We used the NNS Benchmark [24, 32] to compare with OPQ+IMI. Figure 17 reports the time-recall curves of PCAH with Hamming ranking, PCAH with *GQR*, and OPQ with IMI. More results for another 8 datasets are reported in Figures 21-22 in the Appendix due to space limitation. The performance on SIFT10M was not reported as OPQ ran of memory in our machine in the training process. As a replacement, we used SIFT1M[6].

The results show that with Hamming ranking, there is a large performance gap between PCAH and OPQ. Remarkably, with *GQR*, the performance of PACH is improved to be comparable with that of OPQ. We emphasize that this result is significant as OPQ requires more memory and much longer time to train as reported in Table 2, where *Wall time* is the elapsed time and *CPU time* is the summation of the working time of all the CPU cores in the machine. By boosting the performance of simple binary hashing methods, *GQR* gives users more freedom in the selection of hashing methods in their applications, especially when the training overhead is a concern.

7 RELATED WORK

The nearest neighbors search problem has been extensively studied [4, 6, 11, 12, 27, 37–39, 41, 42, 51]. Initially, researchers attempted to find the exact nearest neighbors with the help of data structures such as R-tree [10] and k-d tree [2]. However, these methods suffer from the curse of dimensionality and are proved to perform even worse than linear scan for datasets with more than 20 features [48]. Since approximate nearest neighbors suffice in most applications, a large number of approximation algorithms are proposed. Tree based methods such as k-means tree and random k-d tree partition the entire space using a tree and search the tree with data structures such as priority queue to answer the query [29, 30, 35]. However, the preprocessing and querying efficiency is usually low as the tree is time-consuming to manipulate. Hashing based methods use hash tables and enjoy high querying efficiency. Among them, L2H has been shown to outperform LSH by experiments as L2H can train tailored hash functions for the datasets.

L2H consists of two aspects, hash function learning and query processing. Many hash function learning algorithms, including PCA hashing [8, 43], spectral hashing [44], K-means hashing [11], semi-supervised hashing [40] and others [4, 19, 42], were proposed and the goal is to preserve as much similarity as possible. Hash function learning is usually formulated as an optimization problem with the objective of minimizing quantization error or the difference between the distance (of item pairs) in the original feature space and the resultant binary space. For better query performance, some hash function learning algorithms are complicated, making them inefficient for large datasets [4]. For querying methods, Hamming ranking (HR) has been extensively used in existing L2H methods [4, 19].

While querying method was inadequately discussed for L2H, it has been extensively studied for LSH. Entropy LSH [36] generates random points around the query and probes the buckets these points hashed to. Multi-Probe LSH [28] adds a perturbation sequence to the hash code of the query to generate the buckets to probe. However, Entropy LSH and Multi-probe LSH cannot guarantee to scan the entire dataset (all buckets) and require multiple hash tables. *GQR* can use one hash table to save memory and avoid expensive de-duplication. Some LSH algorithms guarantee to enumerate all the items and can work with only one hash table. C2LSH [16] uses only one hash value for each hash table (i.e., $m = 1$) and dynamically expands the search space bi-directionally from $c(q)$. LSB-tree [39] and SK-LSH [27] probe buckets sharing the longest common prefix with $c(q)$ at first. These LSH algorithms work on external memory and can handle large datasets, but their query performance is generally worse than L2H methods in practice.

8 CONCLUSIONS

We proposed an efficient querying method, *QR*, for L2H, which uses a new similarity indicator called quantization distance to define a linear order for the buckets and probe the buckets sequentially. Based on the property of quantization distance, we also designed *GQR*, the generate-to-probe version of *QR*, which can generate buckets to probe on demand and avoid the slow start problem of *QR*. Experimental results show that *GQR* significantly outperforms the widely used HR querying method consistently across a variety of datasets, code lengths, numbers of hash tables, and numbers of target neighbors (k). We also show that *GQR* is a general querying method and can work with a wide variety of L2H algorithms. Remarkably, *GQR* can also boost the performance of a binary hashing based L2H algorithm to be comparable with that of the state-of-the-art vector quantization based method, OPQ, but using significantly less training overhead. In the future, we plan to extend *GQR* to the distributed setting on data-parallel systems such as LoSHa [21], Husky [45, 46] and others [15, 22].

Acknowledgments. We thank the reviewers for their valuable comments. We also thank Haoran Shu for his contribution in the early stage of this work. This work was supported in part by Grants (CUHK 14206715 & 14222816) from the Hong Kong RGC.

[6]http://corpus-texmex.irisa.fr/

REFERENCES

[1] Artem Babenko and Victor S. Lempitsky. 2012. The Inverted Multi-Index. In *CVPR*. 3069–3076.

[2] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. In *CACM*, Vol. 18. 509–517.

[3] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. 2015. Assembling Large Genomes With Single-Molecule Sequencing and Locality-Sensitive Hashing. In *Nature biotechnology*, Vol. 33. 623–630.

[4] Deng Cai. 2016. A Revisit of Hashing Algorithms for Approximate Nearest Neighbor Search. In *CoRR*, Vol. abs/1612.07545.

[5] Abhinandan Das, Mayur Datar, Ashutosh Garg, and Shyamsundar Rajaram. 2007. Google News Personalization: Scalable Online Collaborative Filtering. In *WWW*. 271–280.

[6] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-Sensitive Hashing Scheme Based on Dynamic Collision Counting. In *SIGMOD*. 541–552.

[7] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2946–2953.

[8] Yunchao Gong and Svetlana Lazebnik. 2011. Iterative Quantization: A Procrustean Approach to Learning Binary Codes. In *CVPR*. 817–824.

[9] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2013. Iterative Quantization: A Procrustean Approach to Learning Binary Codes for Large-Scale Image Retrieval. In *TPAMI*, Vol. 35. 2916–2929.

[10] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.

[11] Kaiming He, Fang Wen, and Jian Sun. 2013. K-Means Hashing: An Affinity-Preserving Quantization Method for Learning Binary Compact Codes. In *CVPR*. 2938–2945.

[12] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2017. Towards Automated Log Parsing for Large-Scale Log Data Analysis. In *TDSC*.

[13] Johannes Hoffart, Stephan Seufert, Dat Ba Nguyen, Martin Theobald, and Gerhard Weikum. 2012. KORE: Keyphrase Overlap Relatedness for Entity Disambiguation. In *CIKM*. 545–554.

[14] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. In *PVLDB*, Vol. 9. 1–12.

[15] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. 2018. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. In *PVLDB*.

[16] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[17] Wang-Cheng Kang, Wu-Jun Li, and Zhi-Hua Zhou. 2016. Column Sampling Based Discrete Supervised Hashing. In *AAAI*. 1230–1236.

[18] Brian Kulis and Kristen Grauman. 2009. Kernelized Locality-Sensitive Hashing for Scalable Image Search. In *ICCV*. 2130–2137.

[19] Learning to Hash. 2017. http://cs.nju.edu.cn/lwj/L2H.html.

[20] Cong Leng, Jiaxiang Wu, Jian Cheng, Xi Zhang, and Hanqing Lu. 2015. Hashing for Distributed Data. In *ICML*. 1642–1650.

[21] Jinfeng Li, James Cheng, Fan Yang, Yuzhen Huang, Yunjian Zhao, Xiao Yan, and Ruihao Zhao. 2017. LoSHa: A General Framework for Scalable Locality Sensitive Hashing. In *SIGIR*. 635–644.

[22] Jinfeng Li, James Cheng, Yunjian Zhao, Fan Yang, Yuzhen Huang, Haipeng Chen, and Ruihao Zhao. 2016. A Comparison of General-Purpose Distributed Systems for Data Processing. In *IEEE BigData*. 378–383.

[23] Wu-Jun Li, Sheng Wang, and Wang-Cheng Kang. 2016. Feature Learning Based Deep Supervised Hashing with Pairwise Labels. In *IJCAI*. 1711–1717.

[24] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2016. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. In *CoRR*, Vol. abs/1610.02455.

[25] Xuelong Li, Di Hu, and Feiping Nie. 2017. Large Graph Hashing with Spectral Rotation. In *AAAI*. 2203–2209.

[26] Wei Liu, Cun Mu, Sanjiv Kumar, and Shih-Fu Chang. 2014. Discrete Graph Hashing. In *NIPS*. 3419–3427.

[27] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Heng Tao Shen. 2014. SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search. In *PVLDB*, Vol. 7. 745–756.

[28] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *VLDB*. 950–961.

[29] Marius Muja and David G. Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *VISAPP*. 331–340.

[30] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. In *TPAMI*, Vol. 36. 2227–2240.

[31] Ankur Narang and Souvik Bhattacherjee. 2011. Real-time Approximate Range Motif Discovery & Data Redundancy Removal Algorithm. In *EDBT*. 485–496.

[32] NNS Benchmark. 2017. https://github.com/DBWangGroupUNSW/nns_benchmark.

[33] Mohammad Norouzi, Ali Punjani, and David J. Fleet. 2012. Fast Search in Hamming Space with Multi-Index Hashing. In *CVPR*. 3108–3115.

[34] Mohammad Norouzi, Ali Punjani, and David J. Fleet. 2014. Fast Exact Search in Hamming Space With Multi-Index Hashing. In *TPAMI*, Vol. 36. 1107–1119.

[35] OpenCV. 2017. http://opencv.org/.

[36] Rina Panigrahy. 2006. Entropy Based Nearest Neighbor Search in High Dimensions. In *SODA*. 1186–1195.

[37] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. 2010. Locality Sensitive Hashing: A Comparison of Hash Function Types and Querying Mechanisms. In *PRL*, Vol. 31. 1348–1358.

[38] Yuxin Su, Irwin King, and Michael R. Lyu. 2017. Learning to Rank Using Localized Geometric Mean Metrics. In *SIGIR*. 45–54.

[39] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and Efficiency in High Dimensional Nearest Neighbor Search. In *SIGMOD*. 563–576.

[40] Jun Wang, Ondrej Kumar, and Shih-Fu Chang. 2010. Semi-Supervised Hashing for Scalable Image Retrieval. In *CVPR*. 3424–3431.

[41] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for Similarity Search: A Survey. In *CoRR*, Vol. abs/1408.2927.

[42] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. 2017. A Survey on Learning to Hash. In *TPAMI*.

[43] Xin-Jing Wang, Lei Zhang, Feng Jing, and Wei-Ying Ma. 2006. AnnoSearch: Image Auto-Annotation by Search. In *CVPR*. 1483–1490.

[44] Yair Weiss, Antonio Torralba, and Robert Fergus. 2008. Spectral Hashing. In *NIPS*. 1753–1760.

[45] Fan Yang, Yuzhen Huang, Yunjian Zhao, Jinfeng Li, Guanxian Jiang, and James Cheng. 2017. The Best of Both Worlds: Big Data Programming with Both Productivity and Performance. In *SIGMOD*. 1619–1622.

[46] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. In *PVLDB*, Vol. 9. 420–431.

[47] Fan Yang, Fanhua Shang, Yuzhen Huang, James Cheng, Jinfeng Li, Yunjian Zhao, and Ruihao Zhao. 2017. LFTF: A Framework for Efficient Tensor Analytics at Scale. In *PVLDB*, Vol. 10. 745–756.

[48] Cui Yu. 2002. High-Dimensional Indexing: Transformational Approaches to High-Dimensional Range and Similarity Searches *(Lecture Notes in Computer Science)*, Vol. 2341. Springer.

[49] Fuzhen Zhang. 2011. *Matrix Theory: Basic Results and Techniques*. Springer Science & Business Media.

[50] Ting Zhang, Chao Du, and Jingdong Wang. 2014. Composite Quantization for Approximate Nearest Neighbor Search. In *ICML*. 838–846.

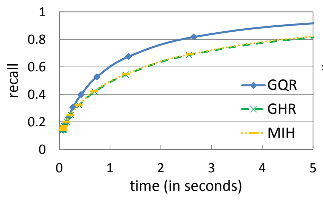
[51] Yuxin Zheng, Qi Guo, Anthony K. H. Tung, and Sai Wu. 2016. LazyLSH: Approximate Nearest Neighbor Search for Multiple Distance Functions with a Single Index. In *SIGMOD*. 2023–2037.

APPENDIX

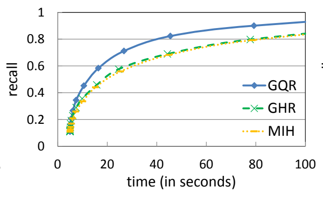
Comparison with Multi-Index Hashing

In this experiment, we conducted experiments by adopting the Multi-Index Hashing (*MIH*) algorithm [33, 34], which is an efficient algorithm to find all buckets within a certain Hamming distance from (the binary code of) the query. The performance of *MIH* on ITQ and PCAH, compared with *GQR* and *GHR*, is reported in Figures 18-19. The results show that *MIH* has slightly worse performance than *GHR*, while our method *GQR* significantly outperforms both *MIH* and *GHR*.

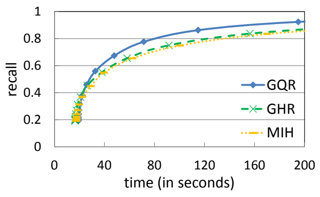
We explain why *MIH* has worse performance as follows. The idea of *MIH* is to chop the code into multiple blocks and build a hash table for each block. Searching is then conducted by merging the search results from the individual blocks, though additional de-duplication and filtering are required. The efficiency of *MIH* lies in its ability to avoid searching the empty buckets under long code length (e.g., 64 bit or 128 bit), in which the code space is much larger than the dataset cardinality and there are many empty buckets. However, long code usually results in poor recall-time performance when used as bucket index [4], and thus we used short code length in our experiments. Specially, we set the code length as an integer approximately equal to $\log_2(N)$, where N is the cardinality of the dataset. In the experiment, we found that the portion of empty buckets hardly exceeds 20%. In this case, *MIH* is



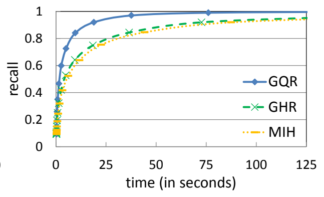
(a) CIFAR60K



(b) GIST1M

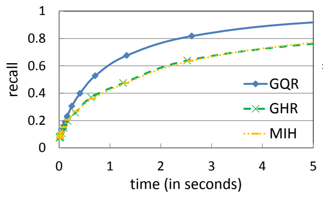


(c) TINY5M

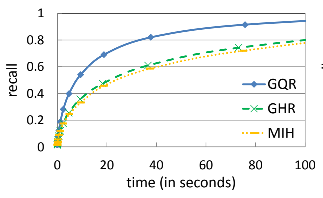


(d) SIFT10M

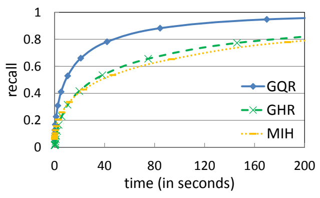
Figure 18: Comparison between GQR, GHR and MIH with ITQ



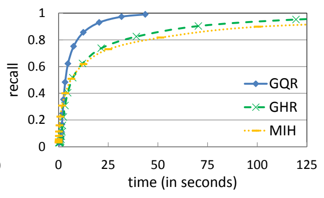
(a) CIFAR60K



(b) GIST1M

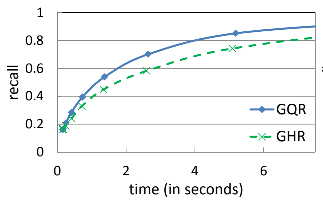


(c) TINY5M

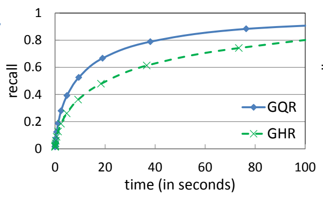


(d) SIFT10M

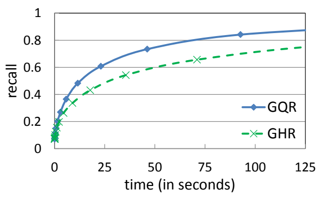
Figure 19: Comparison between GQR, GHR and MIH with PCAH



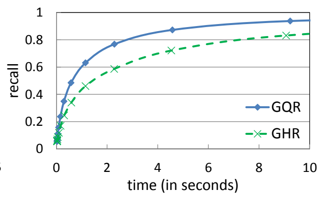
(a) CIFAR60K



(b) GIST1M



(c) TINY5M



(d) SIFT1M

Figure 20: Comparison between GQR and GHR with K-means Hashing

similar to but less efficient than hash lookup (i.e., *GHR*), since *MIH* needs to pay extra cost for de-duplication and filtering.

The result of this experiment shows that the superior performance of *GQR* over Hamming ranking mainly comes from the fine-grained similarity metric, i.e., QD. An efficient algorithm that searches the Hamming space, e.g., *MIH*, does not solve the coarse-grained problem of Hamming distance.

Compatibility with K-Means Hashing

Unlike existing hashing algorithms that normally adopt hyperplanes in quantization, K-means hashing [11] uses K-means to train the codewords for quantization. Since K-neans hashing adopts Hamming distance in query processing, GQR can also improve its performance as shown below.

The vector quantization algorithms usually learn one or more codebooks using K-means and quantize an item to its nearest codewords in the codebooks. For K-means hashing, we define the flipping cost of the i-th bit of query q as $dist(q, c_{q'}) - dist(q, c_q)$, where c_q is the codeword that q is quantized to, while $c_{q'}$ is the codeword that only differs from c_q in the i-th bit of their binary codes. Note that a codeword in K-means is a real-number vector, which is indexed by a binary code.

In the original K-means hashing paper [11], hash lookup (denoted by *GHR*) is used as the querying method. We compared the time-recall performance of *GHR* and *GQR* in Figure 20. The performance on SIFT10M is not reported because the training of K-means hashing ran out of memory in our machine. The result shows that *GQR* can outperform *GHR* by a large margin for K-means hashing.

Comparison with OPQ and More Datasets

In this experiment, we used 8 additional datasets of different types from the NNS Benchmark [24, 32] to compare OPQ, ITQ and PCAH. OPQ uses IMI and ITQ/PCAH uses *GQR* to accelerate similarity search. We randomly sampled 1000 queries for datasets DEEP1M [7], MSONG1M [8], GLOVE1.2M [9], and GLOVE2.2M [9]. We directly used the 200 queries provided together with the corresponding datasets by the NNS Benchmark [24, 32] for AUDIO50K [10], NUSWIDE0.26M [11], UKBENCH1M [12], and IMAGENET2.3M [13]. Some statistics of the 8 datasets are listed in Table 3.

[7]https://yadi.sk/d/I_yaFVqchJmoc

[8]http://www.ifs.tuwien.ac.at/mir/msd/download.html

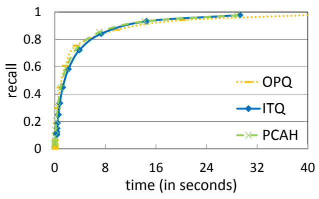
[9]https://nlp.stanford.edu/projects/glove/

[10]http://www.cs.princeton.edu/cass/audio.tar.gz

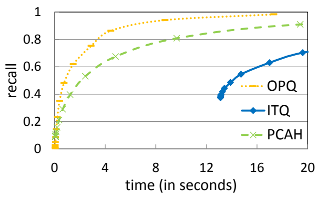
[11]http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm

[12]http://vis.uky.edu/ stewe/ukbench/

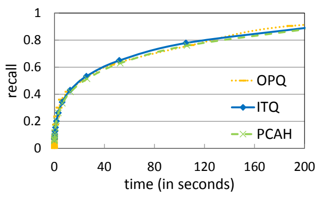
[13]http://cloudcv.org/objdetect/



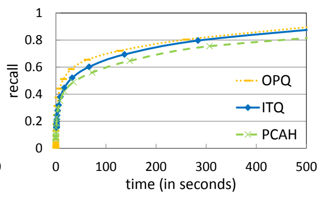
(a) DEEP1M



(b) MSONG1M

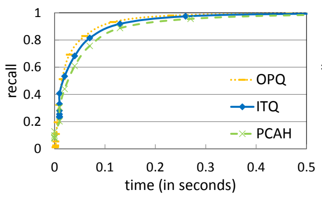


(c) GLOVE1.2M

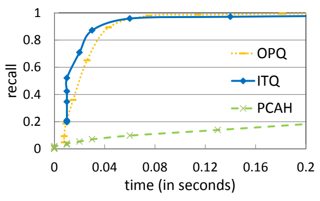


(d) GLOVE2.2M

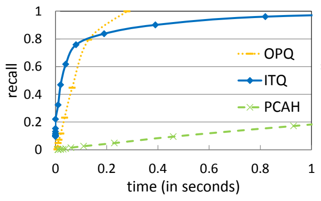
Figure 21: Comparison between (ITQ/PCAH + GQR) and (OPQ + IMI)



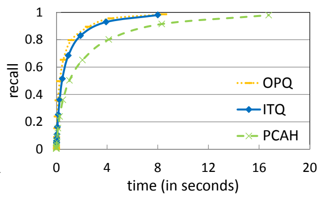
(a) AUDIO50K



(b) NUSWIDE0.26M



(c) UKBENCH1M



(d) IMAGENET2.3M

Figure 22: Comparison between (ITQ/PCAH + GQR) and (OPQ + IMI)

Table 3: Additional Datasets

Dataset	Dim#	Item#	Code Length	Type
DEEP1M	256	1,000,000	16	Image
MSONG1M	420	994,185	16	Audio
GLOVE1.2M	200	1,193,514	16	Text
GLOVE2.2M	300	2,196,017	18	Text
AUDIO50K	192	53,387	12	Audio
NUSWIDE0.26M	500	268,643	14	Image
UKBENCH1M	128	1,097,907	16	Image
IMAGENET2.3M	150	2,340,373	16	Image

We report the performance of PCAH/ITQ using *GQR* and OPQ using IMI, for another 8 datasets in Figures 21-22. The results show that in the majority of the cases *GQR* can boost the performance of PCAH and/or ITQ to be comparable with that of OPQ, while in other cases there is no clear winner.