# Scalable and Efficient Query Methods for Learning to Hash

Liu Jie[1] and Song Yang[2], Supervisor: Prof. James Cheng[3]

**Abstract**

Learning to hash (L2H) is proven to be an effective solution to the approximate *k*-nearest neighbor search problem, as it can learn similarity-preserving hash functions for the given dataset. However, current researches on L2H are mainly focusing on learning good hashing functions from a given dataset and hamming distance is used as the default similarity indicator to measure the similarity between the query and items in the dataset. In this report, we show that the coarse-grained nature of hamming distance limits the querying efficiency, and we propose that we can improve the query performance of binary hashing based L2H methods by using another similarity indicator, quantization distance. Then we present an efficient querying method Generate-to-probe Quantization Ranking based on quantization distance. Our proposed method can work with various L2H algorithms. We also implement the improved query method in a distributed setting to handle large datasets.

## I. Approximate *k*-nearest Neighbor Search

Similarity search in high-dimensional spaces has become increasingly important in databases, data mining, and search engines, particularly for content-based search of feature-rich data such as audio recordings, images, videos. For high-dimensional spaces, there are often no known algorithms for nearest neighbor search that are more efficient than simple linear search [1]. As linear search is too costly for most applications, approximate algorithms are used to provide large speedups. For example, when we use Google to search for something we are interested in, we would not wait for several hours before the "optimal" answer finally comes out. Rather, we would like to get a reasonable answer within a few seconds. Approximate algorithms can be orders of magnitude faster than linear search, while still achieving good accuracy.

In a typical approximate *k*-nearest neighbor search problem, we are usually given a large dataset consisting of millions of data objects. Each data object is typically represented as a high-dimensional feature vector. Depending on different datasets, the dimension in each vector may vary from hundreds to millions. When the dimension of each vector is high (e.g., millions of dimensions), the vector is referred to as sparse vector, for most of the value of each dimension will be zero and only some dimensions have meaningful data. When the dimension of each vector is low (e.g., hundreds of dimensions), the vector is referred to as dense vector, for most of the value of each dimension will be non-zero. The vectors in the given dataset are called item vectors. We need to look for near neighbors for a set of query vectors in the whole dataset. The number of query vectors is not very large (e.g., 1000) compared to the size of database.

In order to find the nearest neighbors for each query vector, we need to first define the distance between query vectors and item vectors. There are some frequently used measure metrics, such as Euclidean distance, angular distance and Jaccard distance, etc.

For each query, we need to develop an algorithm to find its *k*-nearest neighbors in the item vector set, according to the defined distance. As we explain previously, the *k*-nearest neighbors we found, in most cases, are not the exact *k*-nearest neighbors and they are usually near-optimal solutions. In practice, people find that it is sufficient to find approximate *k*-nearest neighbors and the most important

*We also receive a lot of support from Prof. James Cheng's Ph.D. students Jinfeng Li and Xiao Yan

[1]Liu Jie, final-year student, Department of Computer Science and Engineering, The Chinese University of Hong Kong (CUHK), Shatin, N.T., Hong Kong the Chinese University of Hong Kong, Jerry.liujie95 at gmail.com

[2]Song Yang, final-year student, Department of Computer Science and Engineering, The Chinese University of Hong Kong (CUHK), Shatin, N.T., Hong Kong the Chinese University of Hong Kong, sy568386042 at gmail.com

[3]James Cheng, assistant professor, Department of Computer Science and Engineering, The Chinese University of Hong Kong (CUHK), Shatin, N.T., Hong Kong, jcheng at cse.cuhk.edu.hk

concern for approximate $k$-nearest neighbor search is efficiency, i.e., retrieving a large portion of the true $k$-nearest neighbors in a short time.

## II. RELATED WORK

The current solutions to approximate $k$-nearest neighbors search problem can be classified into two categories, tree-based methods and hashing-based methods. Tree-based methods include R-tree [2], $k$-$d$ tree [3], SR-tree [4], random $k$-$d$ tree [5], [1], hierarchical $k$-means tree [5], [1]. They use tree structure to divide the space into multiple partitions and assign each item vector to different partitions. At each level of the tree, the data is split by a hyper-plane in $k$-$d$ tree while k-means clustering is utilized in hierarchical $k$-means tree. Nearest neighbors are then searched by performing a backtracking or priority search in this tree. The drawbacks of tree-based methods are that they may suffer a lot more from complicated preprocessing (especially for $k$-means tree method) and querying when the dimension of vectors increases. They are proved to perform worse than linear scan for datasets with more than 20 dimensions [6].

On the other hand, currently more research interests are focusing on hashing-based methods. Compared to tree-based methods, hashing-based methods use hash tables and enjoy high querying efficiency. Performance advantage of hashing-based algorithms over tree-based algorithms has already been widely recognized. There are generally two categories of hashing-based methods, locality sensitive hashing (LSH) and learning to hash (L2H). The key difference between them is whether the hash functions are dataset-dependent or not.

The main idea of LSH is that, by using a family of predefined hash functions, similar items are hashed to the same bucket with higher probability and one only needs to check a small number of buckets for items similar to a query. LSH can be implemented on relational databases and has sub-linear querying time complexity regardless of the distribution of data/queries [7]. Some common LSH families include min-wise permutation for Jaccard distance, random projection for angular distance, and 2-stable projection for Euclidean distance. Multiple hash tables are generated, and multiple hash values are concatenated and used as a whole for each hash table in order to enlarge the gap of to-same-bucket probability between similar items and dissimilar items. Compared to L2H methods, the disadvantage of LSH is pretty obvious. Since it uses predefined hash functions, it is completely blind and does not look at the data at all, as is shown in Figure 1.

L2H learns tailored hash functions for the dataset. Although an additional stage of training the hashing functions is required, L2H significantly outperforms LSH in terms of query performance. Currently there

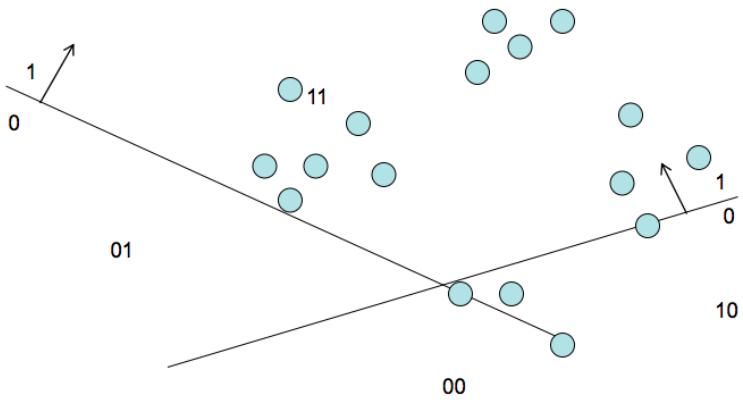



Fig. 1: Locality sensitive hashing: Completely blind, not looking at the data at all

are two main catogories of learning to hash methods: vector quantization(VQ) based methods and binary hashing methods. Vector quantization(VQ) based methods such as K-means hashing [8] and OPQ [9] represents the state-of-art learning to hash methods and they were reported to outperform simple binary hashing based methods (using hamming distance) by large margin. The VQ based methods usually learn one or more codebooks using K-means and quantize an item to its nearest codeword in the codebook. The superiority of VQ can be illustrated using the Figure 2 and Figure 3.

The vector quantization (VQ) methods have better performance over binary hashing methods (with Hamming ranking) due to better model flexibility. However, the binary hashing methods also have

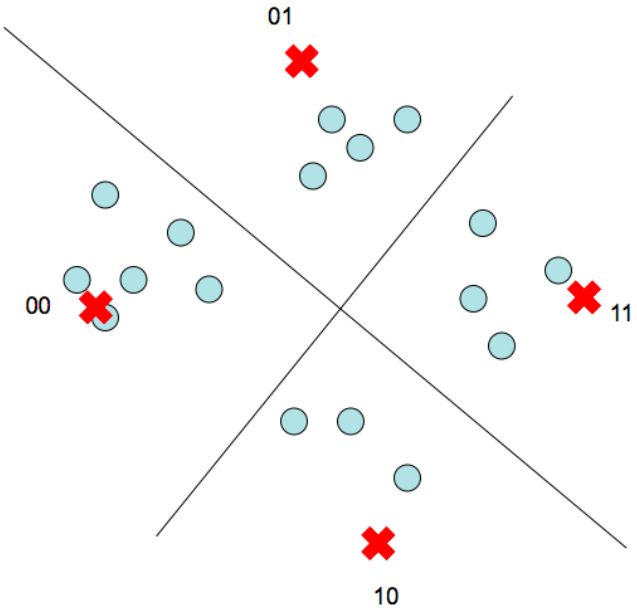



Fig. 2: Binary hashing: use Iterative quantization (ITQ) method to illustrate, first use PCA for dimensionality reduction, then use orthogonal projections, pick the best rotation of the data (or of the hypercube) to minimize quantization errors

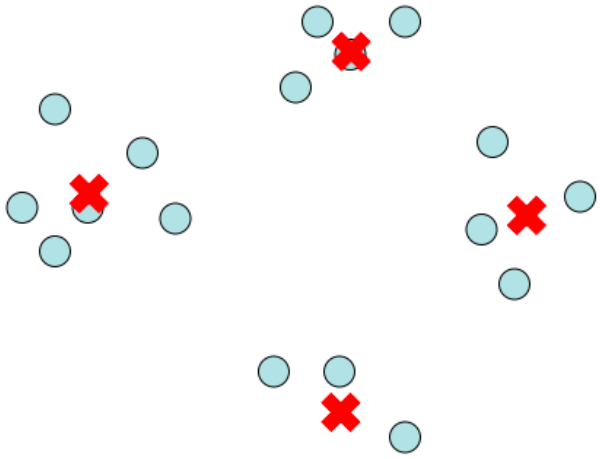


Fig. 3: Vector quantization: instead of restricting to a hypercube, pick best k locations in the space to quantize to – i.e. k-means

important use cases. Firstly, the training complexity of the VQ methods is significantly higher than binary hashing methods as they invoke K-means multiple times. Moreover, binary hashing algorithms have been successfully adapted to a number of scenarios, while how to apply VQ based methods in these scenarios is not clear. For example, we can learn the hash functions for binary hashing in an online fashion and also handle very large dataset in a distributed manner for binary hashing and there are a number of researches about how to do binary hashing with supervision. In the aforementioned scenarios and when the training complexity is of concern, the binary hashing methods may be used instead of the VQ based methods.

We mainly focus on binary hashing based learning to hash method in the following chapters. Later we will show that the method we proposed is possible to boost the binary hashing method to achieve a performance comparable with that of vector quantization(VQ) methods. This result can be significant because VQ needs to use much longer time and also more memory.

## III. LEARNING TO HASH

Learning to Hash (L2H) has two major steps, learning and querying. Accordingly, the performance of L2H depends on both the hash function and the querying method. In the learning stage, it will try to learn good hash functions from the given dataset. Currently, almost all L2H researches focus on learning good hash functions and many learning algorithms have been proposed, including spectral hashing (SH) [10], principal component analysis hashing (PCAH) [11], iterative quantization (ITQ) [11], semi-supervised hashing (SSH) [12], and etc. While advanced hash function learning algorithms have become increasingly complicated but lead to only marginal gain in query performance, design of querying method is somehow ignored. There is still much space for improving the current querying method mainly based on hamming distance.

In our work, we mainly focusing on improving the querying process and we used Iterative Quantization (ITQ) as the default learning method. In the following, we will briefly introduce the hashing and querying in L2H.

### *A. Hashing*

Suppose that we have a *d*-dimensional vector, we want to use hashing to map it to a binary code with length *m* (normally *m* < *d*). Our goal is that similarity is well preserved in the mapping, that is, if vectors are close to each other in the orginal d-dimensinal space, then there is a high probability that they have same or at least similar m-dimensional binary codes. The hashing is achieved in two operations, projection and quantization.

**Projection** Assume that we have $m$ learned hash functions ( $h_1,\ h_2,\ ...,\ h_m$ ) and a vector *v*. Then, applying the *m* hash functions to *v*, we can generate *m* real number, which can form a *m*-dimensional real value vector, i.e., **projected vector**. The projected vector is $p(v) = (h_1(v),\ h_2(v),\ ...,\ h_m(v))$. The value in each dimension of the projected vector can be represented as $p_i$, where $i = 1, ..., m$. We can take each hash function as a hyperplane in the space, and the hashed value is the distance between the query vector and the hyperplane.

**Quantization** After getting the projected vector, we will convert the $m$-dimensional projected vector to a binary code with length $m$. The binary code is $c(q)$ = ( $quan(p_1)$, $quan(p_2)$, ... , $quan(p_m)$ ) and it is also called **signature**. The quantization function, $quan()$, will convert $p_i \geq 0$ to 1 and $q_i < 0$ to 0.

After quantization, we get the signature (binary codes) of the original vector and each signature corresponds to a bucket id in the hash table. A hash tables has many buckets, each bucket contains the vectors having the same signature. A hash table and its buckets are determined by the hash functions. We can even have multiple hash tables. For example, if we have $n$ hash tables, then we will need $n \times m$ hash functions. The hash functions will be divided into $n$ groups. Each group contains $m$ hash functions and determines a hash table.

### B. Querying

Given a query vector $q$, we pass it to hash functions groups and then get its signatures. With these signatures, predefined querying method is used to determine how to probe buckets in each hash table. We can adopt either **single-probing** strategy or **multi-probing** strategy. Single-probing method is the basic method which uses a family of locality-sensitive hash functions to hash nearby objects in the high-dimensional space into the same bucket. It hashes a query object into a bucket, and only uses the data objects in that bucket as the candidate set of the results, and then ranks the candidate objects using the distance measure of the similarity search. To achieve high search accuracy, the LSH method needs to use multiple hash tables to produce a good candidate set. Experimental studies show that this basic LSH method needs over a hundred and sometimes several hundred hash tables to achieve good search accuracy for high-dimensional datasets. Since the size of each hash table is proportional to the number of data objects, the basic approach does not satisfy the space efficiency requirement [13]. The main idea of multi-probing is to build on the basic single-probing method, but to use a carefully derived probing sequence to look up multiple buckets that have a high probability of containing the nearest neighbors of a query object. Then we retrieve the item vectors in each bucket we visited and evaluate the distances between the query vector and retrieved item vectors. We keep updating top-$k$ nearest neighbors by ranking the distances. This retrieving-evaluating-updating iteration will not be terminated until a predefined number of nearest neighbor are fetched.

Querying method determines which bucket to probe first and which to probe next. Obviously, if we could decide which bucket is most likely to contain nearest neighbors and probe it at the beginning, query performance would be quite good. The existing work on learning to hash are all using hamming ranking as their querying method and we will discuss more details in later sections.

## IV. HAMMING RANKING

Hamming distance can measure the difference between two equal length strings. It indicates the number of different characters in two strings. For example, the hamming distance between 'cat' and 'car' is 1, since the third character of two strings differs. The hamming distance between 'ate' and 'tea' is 3 though they have the same set of alphabets.

### A. Method Overview

Assume that we only have one hash tables, after computing the signature of the query vector, the algorithm will start to probe buckets in the hash table. As each bucket has a signature, then the algorithm will calculate the hamming distances of bucket signature to the query signature and sorted the buckets in ascending order of their hamming distances.

In the next step, the algorithm will begin to probe the bucket according to their hamming distance. First, it will probe the bucket with 0 hamming distance. Fetch the candidate item vectors in that bucket and if the algorithm get $k$ item vectors, it will stop and output the result. If it does not find $k$ item vectors, it will keep the item vectors as part of the output and continue to probe the buckets with 1 hamming distance to the query. If in the probed buckets, the algorithm still haven't find $k$ item vectors, it will keep the item vectors as part of the output and continue to probe the buckets with 2 hamming distance to the query. It will continue this process until it find $k$ item vectors. The details of this algorithm are in Algorithm 1.

**Algorithm 1** Hamming Ranking

**Input:**
The query vector signature, $q$ ;
$n$ buckets, their signatures are $b_1, b_2, ..., b_n$;
$k$, number of nearest neighbor need to find.
**Output:** approximate $k$-nearest neighbors to $q$

Set output candidate vector set $C = \emptyset$
$i = 0$
**while** $|C| < k$ **do**
    Find the buckets whose hamming distance to the query vector is $i$
    Fetch the item vectors in those buckets and put them in the output set $C$
    $i = i + 1$
**end while**
Calculate the similarity between candidate vectors in $C$ and the query vector $q$
Sorted the candidate vectors in $C$ according to their similarity
Output the top-$k$ in $C$

### *B. Drawback*

The problem of hamming ranking algorithm is that if the algorithm fail to find $k$ item vectors in the first bucket, it will need to probe a large number of buckets in later iterations. For example, if the code length is 4 and the query signature is 1111, then in the second iteration, it will need to probe bucket 1110, bucket 1101, bucket 1011 and bucket 0111 (refer to Figure 4). As the code length can be even longer, the algorithm may need to probe a lot more buckets, which may result in a huge waste in computation resources. If the code length is 32, then when probing the buckets of 2 as their hamming distance to the query, we will need to probe $\binom{32}{2}$.

This drawback indicates that the hamming ranking algorithm is too coarse-grained and we need a more fine-grained algorithm during the probing, which should be able to help us determine the probing order among the buckets with equal hamming distance (HD).

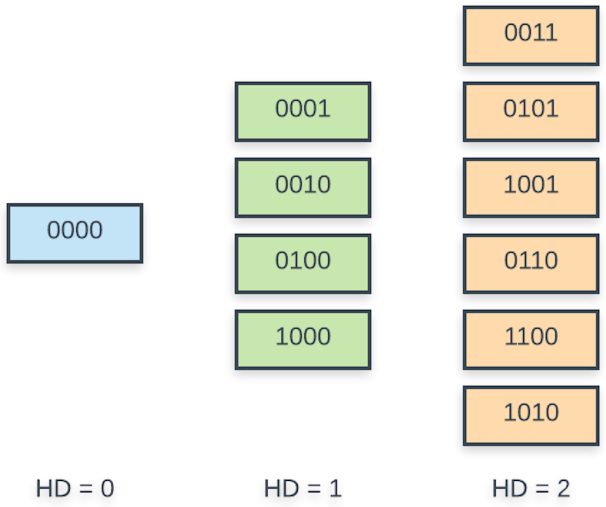



Fig. 4: The buckets need to probe will increase as HD increase

## V. Quantization Distance Ranking

From the last section, we can see that hamming ranking algorithm still needs to be improved, as it is not efficient in probing buckets. To probe buckets more efficiently, we propose the quantization distance ranking algorithm.

### A. Quantization Distance

For a given query $q$, whose signature is $c(q)$ and projected vector is $p(q)$, and a bucket, whose signature is $b$, the quantization distance between $q$ and $b$ is defined 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 $b_i$ represents the $i^{th}$ bit of the signature for query $q$ and bucket $b$, respectively; $p_i(q)$ represents the $i^{th}$ entry of $q$'s projected vector; and $\oplus$ is the exclusive-or operation. The quantization distance measures the minimum change that the projected vector $p(q)$ needed to be quantized to bucket $b$. Only when $b_i \neq c_i(q)$, $|p_i(q)|$ will be added to $dist(q, b)$.

For example, if $b$ is $0000$ and $p(q)$ is $(-0.1, 0.3, -0.5, 0.7)$, then $c(q)$ is $0101$ and $dist(q, b)$ is $0.3 + 0.7 = 1$. If $p(q)$ is $(-0.1, -0.3, -0.5, 0.7)$, then $c(q)$ is $0001$ and $dist(q, b)$ is $0.7$.

The quantization distance will keep the similarity information that is lost during the quantization. For each query, we generate a list of real values after projection operation, and the value on each bit not only indicates which side it falls into, but also give us the information how likely its neighbor falls into the other side. For example, on ith bit, the projected value of query q is 0.05 and that of its nearest neighbor is -0.05. They have different hash values in spite of the fact that they are very close to each other. The smaller the absolute value of projected value on that bit, the more likely the similar items have different binary values. With the help of quantization distance, when we are probing buckets, we can sort all buckets first and then probe them one by one according to their quantization distance to query $q$.

### B. Quantization Distance Ranking Algorithm

In this section, we introduce a naive algorithm using quantization distance to replace hamming distance in Algorithm 1.

**Algorithm 2** Quantization Distance Ranking

**Input:**
The query vector signature, $q$ ;
$n$ buckets, their signatures are $b_1, b_2, ..., b_n$;
$k$, number of nearest neighbor need to find.

**Output:** approximate $k$-nearest neighbors to $q$

1: Set output candidate vector set $C = \emptyset$
2: Calculate the quantization distance between each bucket and the query $q$
3: Sort the buckets in ascending order according to their quantization distance, represent the sorted set as $B$
4: $i = 1$
5: **while** ($|C|$ < $k$) and ($i \leq |B|$) **do**
6: Fetch the item vectors in $i^{th}$ bucket of $B$ and add them to the output set $C$
7: $i = i + 1$
8: **end while**
9: Calculate the similarity between candidate vectors in $C$ and the query vector $q$
10: Sorted the candidate vectors in $C$ according to their similarity
11: Output the top-$k$ in $C$

However, in Algorithm 2, when the signature is long, we will have even more computations to do in calculating quantization distances and sorting buckets. Assume that the signature has 32 bits, then it could have $2^{32}$ buckets in total. Calculating the quantization distance for $2^{32}$ buckets will be very slow and sort all these quantization distance will let our algorithm run even slower. To solve this problem, we will introduce more improvements in next section.

### C. Comparison with score in Multi-Probe LSH

Quantization distance is inspired to some extent by the score in Multi-Probe LSH [13], but there are a few fundamental differences. The first difference lies in their definitions. The score 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. Moreover, the score works with integer code, while QD works with binary code.

## VI. GENERATE-TO-PROBE QUANTIZATION DISTANCE RANKING (GQR)

In the last section, We introduced a more fine-grained similarity indicator quantization distance for querying to determine which bucket to probe. However, we still need a practical algorithm to use quantization distance without introducing too much additional computation.

We want to probe buckets in ascending order according to their quantization distances. But we do not want to introduce more computation to querying process, for calculating all quantization distance between each bucket and query $q$ is obviously too expensive. Usually, the length of signature is set to more than 10. For some datasets with high dimensions and large number of items, we may need even longer (20-30) signature to filter the nearest neighbors from datasets. Let's say, the length of signature is 30 in this case, then the number of buckets would be $2^{30} = 1073741824$. That is to say, before the iterative processing, we need to sort the quantization distances between the bucket that query belongs to and these 1073741823 buckets, which requires a huge amount of computing. Moreover, among these 1073741823 buckets, most of them actually do not contain any items inside it since the distribution of items in buckets is very sparse, so there is no sense in calculating all the quantization distance. Though quantization distance is a more fine-grained similarity indicator compared to hamming distance, we could not put it into practice if the performance is much worse than the hamming ranking. We always strive for a balance between efficiency and precision.

To solve this problem, we propose a Generate-to-Probe method based on Min-Heap. Before we talk about our algorithm, we need to introduce some concepts.

### A. Flipping Vector

After projection and quantization, we have a projected query vector and an original signature of the query at hand. What we need to do next is to generate buckets that are very likely to contain nearest neighbors to probe first. We apply flipping vectors to original signature to generate the candidate buckets. Assume that the code length is $m$. 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 $1 \leq i \leq m$.

Flipping vector is just a binary code with length $m$ and it can be used to generate new candidate bucket IDs based on the query signature, i.e. the original bucket id. Each bit of a flipping vector indicates whether to flip the corresponding bit in the query signature. For example, given original signature 1010 and the flipping vector 1000, we need to flip the first bit of signature and keep remaining bits unchanged, thus the resulting bucket id is 0010. There are $2^m - 1$ flipping vectors for each query. Applying these flipping vectors to each query signature, we can convert the query signature to any bucket.

### B. Sorted Flipping Vector

In order to find the nearest neighbors efficiently, we should probe the bucket differing on the bit that has smallest projected value first. We sort the absolute value of projected vector for each query first. Then every time we need to generate a new bucket to probe, we flip one bit of original signature of the query according to the sorted projected vector so that we can probe the bucket with higher probability first.

Assume that we have a projected vector $p(q)$. We can sort the absolute value of each element in $p(q)$, which is to sort element in $(|p_1(q)|, |p_2(q)|, ..., |p_m(q)|)$, and then we get a new vector, the sorted projected vector $\tilde{p}(q)$.

The sorting process define 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 denote sorted flipping vector as $\tilde{v}$. It indicates the entries to flip in the sorted projected vector.

The Algorithm 3 can help us find the corresponding bucket signature from a given sorted flipping vector. Sorted flipping vector has many nice properties, which lead to the design of an efficient algorithm. We will talk about this in the next section.

**Algorithm 3** Sorted Flipping Vector to Bucket Signature

**Input:** sorted flipping vector $\tilde{v}$, binary code $c(q)$, code length $m$
**Output:** bucket to probe $b$

```
b = c(q), i = 1
while i ≤ m do
    if ṽ_i == 1 then
        l = f(i)
        b_l = 1 − c_l(q)
    end if
    i = i + 1
end while
Return b
```

### C. Min-Heap based Generate-to-Probe

*1) Swap and Append:* We need to define two operations on a sorted flipping vector (or just consider this as a operation to a bit vector) $\tilde{v}$, *Swap* and *Append*.

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

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

*Append* adds a 1 entry to the right-hand side of the rightmost 1, while *Swap* exchanges the positions of the rightmost 1 and the 0 on its right hand side. For example, if $\tilde{v} = 1000$, then $\tilde{v}^+ = 1100$ and $\tilde{v}^- = 0100$ respectively. More examples are given in Figure 5.

*2) Generation Tree:* Let $\tilde{V}$ be the set of all bit vectors with length $m$. Let $\tilde{v}^r$ be a bit vector in $\tilde{V}$, where $\tilde{v}^r_1 = 1$ and $\tilde{v}^r_i = 0$ for $i < i \leq m$. Let $\tilde{v}^0$ be an all-zero bit vector in $\tilde{V}$, i.e., $\tilde{v}^0_i = 0$ for $1 \leq i \leq m$. With *Append* and *Swap*, we define the generation tree as follows.

**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 two operations *Append* and *Swap*, such that the right child of an

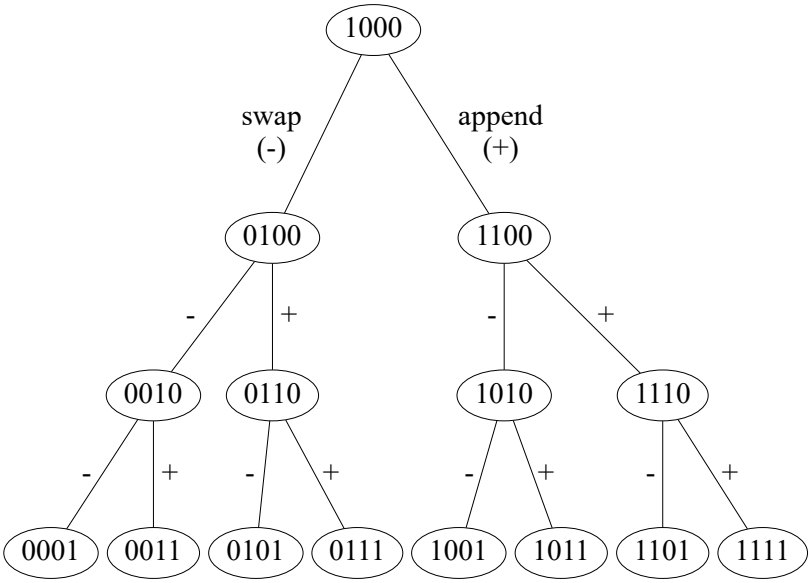



Fig. 5: Example of a Generation Tree

internal node is generated by applying *Append* on the node, while the left child is generated by applying *Swap* on the node.

The generation tree has two important properties. As query $q$ is clear in the context, in the following discussion we simply denote the quantization distance of flipping vector $v$ from $q$ as $dist(v)$, and that of sorted flipping vector $\tilde{v}$ from $q$ as $dist(\tilde{v})$.

**Property 1. One-to-one mapping** Let $T$ be the generation tree with root $\tilde{v}^r$. There is a one-to-one mapping from each node in $T$ to each bit vector in $\tilde{V} \setminus \tilde{v}^r$.

This property shows that all possible sorted flipping vectors can be obtained from the generation tree exactly once.

**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 $dist(\tilde{v}) \leq dist(\tilde{v}^-)$.

This property shows that in the generation tree, the quantization disatance of a child is no smaller than its parent. If we apply the generation tree to generate sorted flipping vector, then when a new sorted flipping vector is generated, its quantization distance can be easily obtained. To be specific, we have $dist(\tilde{v}^+) = dist(\tilde{v}) + \tilde{p}_{i+1}(q)$ and $dist(\tilde{v}^-) = dist(\tilde{v}) + \tilde{p}_{i+1}(q) - \tilde{p}_i(q)$.

*3) Generate-To-Probe Quantization Distance Ranking (GQR):* Based on the above two properties, we use a min-heap, $h_{min}$, to generate sorted flipping vectors. Each element in $h_{min}$ is a tuple $(\tilde{v}, dist(\tilde{v}))$, where $dist(\tilde{v})$ is the key. We initialize $h_{min}$ by inserting the vector $\tilde{v}^r = (1, 0, ...0)$. When we need a bucket to probe, we dequeue the top element, $\tilde{v}$, from $h_{min}$, and use it to generate the bucket described in Algorithm 4. We also apply *Swap* and *Append* operations on $\tilde{v}$ to generate two new tuples, which correspond to the two children of $\tilde{v}$ in the next generation tree, and then we insert the new tuples into $h_{min}$.

**Algorithm 4** Generate-to-Probe Quantization Distance Ranking

**Input:** query $q$, code length $m$, and the number of candidates to collect $N$
**Output:** approximate $k$-nearest neighbors of $q$

Set the candidate set $C = \emptyset$
Compute the binary code $c(q)$ and the sorted projected vector $\tilde{p}(q)$
Collect items in bucket $c(q)$ into $C$
$h_{min}.push(\tilde{v}^r = (1, 0, ..., 0),\ |\tilde{p}_1(q)|)$
i = 1
**while** $i \leq 2^m - 1$ and $|C| < N$ **do**
    $\tilde{v} = h_{min}.del\_min()$
    $\tilde{v}^+ = Append(\tilde{v})$
    Set $j$ as the index of the rightmost 1 in $\tilde{v}$
    $h_{min}.push(\tilde{v}^+,\ dist(\tilde{v}) + \tilde{p}_{j+1}(q))$
    $\tilde{v}^- = Append(\tilde{v})$
    $h_{min}.push(\tilde{v}^-,\ dist(\tilde{v}) + \tilde{p}_{j+1}(q) - \tilde{p}_j(q))$
    Use $\tilde{v}$ to calculate the bucket $b$ to probe by Algorithm 3
    Collect items in $b$ into $C$
    $i = i + 1$
**end while**
Re-rank the items in $C$ by their Euclidean distances to $q$ in ascending order
Return the top-$k$ items

From property 1 we can make sure that Algorithm 4 will not probe a same bucket multiple times, as all sorted projected vector in the generation tree can be generated exactly once. From property 2 and min-heap, we can make sure that the vector $\tilde{v}$ generated by Algorithm 4 in $i^{th}$ iteration is the $i^{th}$ smallest quantization distance from $q$.

The memory consumption of Algorithm 4 is also less than the hamming ranking algorithm, as we do not need to store the hamming distances of all buckets.

## VII. EVALUATION

In this section, we evaluate the performance of our algorithms, QD ranking in Algorithm 2 and generate-to-probe QD ranking in Algorithm 4, denoted by QR and GQR, respectively. We compared our algorithms with the state-of-the-art methods.

### *A. Datasets and setting*

CIFAR-10 (denoted by CIFAR60K) is a dataset used as benchmark in many ANN search evaluations. Following the common practice in these work, we extracted a 512 dimensional GIST descriptor for each of the 60000 CIFAR-10 images and the GIST descriptors are treated as items in the dataset.

In terms of other experiment setting, the hash functions were trained using ITQ, 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 (query items are not used in the hash function learning) and report the average performance. The default code length is set to 12 for CIFAR60K dataset.

### *B. 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 the given dataset.

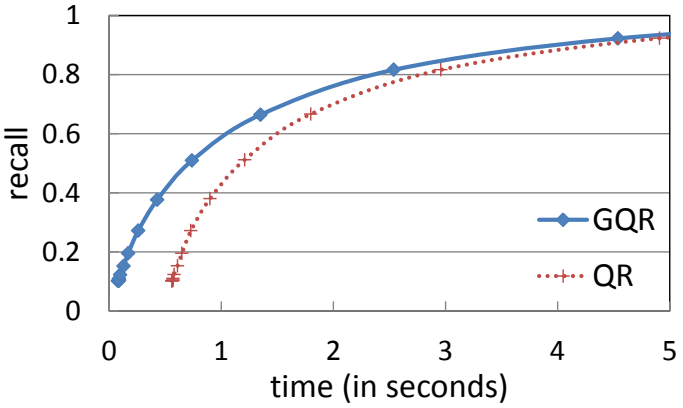
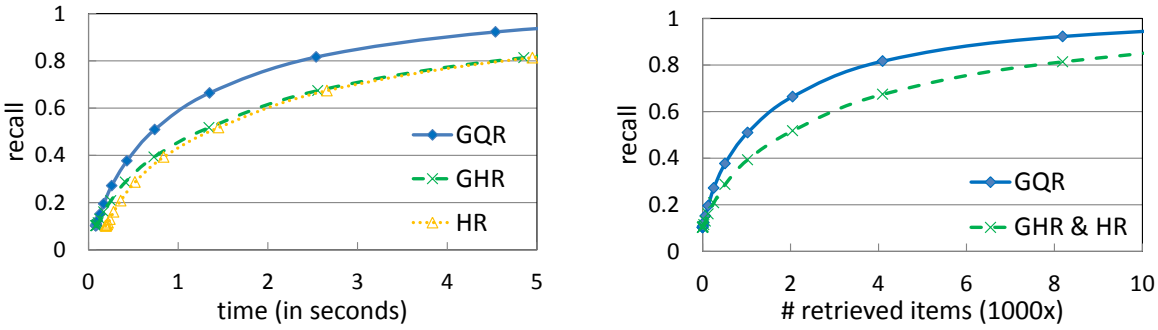



Fig. 6: CIFAR60K: GQR VS QR




(a) Recall (b) Precision

Fig. 7: Recall & Precision

### C. 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. We also implemented another algorithm called 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.

The figure 7 shows that GQR outperforms both HR and GHR on precision as well as the recall time, which can be explained by the fact that QD is a more fine-grained and better similarity indicator than Hamming distance.

## VIII. DISTRIBUTED IMPLEMENTATION

In 2012, Facebook ingested 500 TB/day. Google served > 1.2 billion queries per day on more than 27 billion items.In the era of big data, We are even producing more data than we able to store. Specific to LSH, it often needs to deal with large-scale data, such as billions of tweets or images. In most cases, tweets or images will be converted to high dimensional vectors for LSH computation. With billions of high dimensional vectors, we can hardly put the data into the main memory of a single machine. We turn to distributed computing platform in order to scale up processing for big data. A distributed implement of Generate-To-Probe Quantization Distance Ranking (GQR) algorithm is more scalable.

### A. Major Challenges

*1) Single-Probing VS Multi-Probing in Distributed Implementation:* We have introduced the differences between single-probing method and multi-probing method in previous sections. Most of the distributed

solutions are single-probing LSH only, whose query processing is simple and easy to implement. For each query, all we need to do is to hash it to a bucket and then retrieve all the candidate items in the same bucket. To guarantee high search quality, the single-probing LSH scheme needs a rather large number of hash tables. In the distributed setting, this entails a large space requirement, and in the distributed setting, with each query requiring a network call per hash bucket look up, this also entails a big network load [14]. But multi-probing could be also kind of troublesome. We also need to determine a unique probing sequence for each query vector and check whether the number of candidates has satisfied our predefined requirements. Since the signatures can have a fairly long length, determining which bucket to probe next can cause unnecessary computing cost if we do not implement distributed multi-probing algorithm properly. To program multi-probing LSH, users need to define a complicated dataflow consisting of many steps. Each step involve multiple operations, and users have to carefully select suitable operators (e.g., which join operator to use) and consider many other implementation details (e.g., how to avoid unnecessary re- computation by data caching on Spark). In order to make the implementation efficient, we need to do much debugging and performance tuning work among many steps, which is a time-consuming job to do.

*2) Difficulties in Using General Computing Framework:* When it comes to implement some algorithm in a distributed environment, the first thought of most people may be to utilize the existing popular distributed computing framework, such as Hadoop and Spark. Currently, there is a lack of programming framework for LSH algorithms in either Hadoop or Spark. With a general programming framework, everything would become a lot easier. For example, if we have some work to do in graph processing, we could turn to Pregel, which is Google's scalable and fault-tolerant platform with an API that is sufficiently flexible to express arbitrary graph algorithms. We would not choose to implement some framework from scratch since the implementation cannot be very efficient without the problem-specific optimization.

We noticed that most of the works used the batched processing system MapReduce and adopted external-memory implementations, which have limited query throughput and long delay due to the computing capacity of a single machine. Defining complex dataflows cannot be avoided when working on these platforms and this could be inflexible to use and difficult in debugging and testing. Also, it needs much performance tuning efforts to make it really efficient.

Moreover, using existing computing framework, the items and queries will always be needed to transfer through the network, which will cause inefficiency. As a result, the memory consumption and network traffic will be very high.

In a word, multi-probing LSH is inherently difficult to be implemented in a distributed setting and the existing popular general frameworks cannot give us enough support when we are to implement an efficient distributed LSH algorithm. Hence, in our implementation we used another distributed computing framework – Husky.

### *B. Programming Framework for Multi-probing LSH*

To address the above problem, a task of top priority is to design a programming paradigm so that users can express different types of LSH algorithms (especially multi-probing LSH) easily and concisely. We make effort on searching for an appropriate level of abstraction. In other words, we want to hide the unnecessary details from users and give them enough flexibility when they implement their LSH algorithms at the same time.

*1) Abstractions: Hash functions, Query, Bucket and Item:* We have three data abstractions: Query, Bucket and Item. We first load all the item vectors from datasets to create a set of Item objects. And we also create a set of Bucket objects which has its id represented as bId. Item object has its own signature after hashing. If the signature of the Item Object is the same as bId of one Bucket object, we say this Item object belongs to this Bucket. Query object is for the input query vector. Queries can be preprocessed in a similar way to create a set of Query objects.

We will first give an overview of how these three abstractions interact and how the entire system works. Since we are not doing single-probing, we need multiple iterations to probe nearby buckets and fetch items in them. In each iteration, each Query needs to find a set of potential buckets in order to retrieve similar items. We denote this function as "query" function. The "query" function can be customized to implement various query methods, providing adequate programming flexibility for users. After the set of potential Bucket objects is finalized, we would go to these Bucket objects and see what's inside. Since we have processed entire datasets to know which Item belongs to which Bucket, finding those Bucket objects means that we could easily find the similar Item objects. Bucket object forwards the information of Query object to the Item object. Then Item object have the information about the query vector and its own vectors, and thus it can compute the distance between the target query vector and itself. It returns the item id and distance information to Query object. We call this process "answer" function. That is basically how one iteration works. In the next iteration, we would try different Bucket objects according to querying scheme. This query-answer scheme is pretty easy to understand and implement. In this programming framework, users only need to define customized "query" and "answer" function and the underlying complex dataflow is properly hidden.

*2) Initialization:* We have a pre-processing stage when we train the hash functions from the given data and store all the learned parameters for later use. When running our application, we simply read all the hash functions from files and use it to calculate the signature of input vectors.

After initialization of the hash functions, read the dataset chunk by chunk. We will create an Item object list in each worker and store all item vectors read by that worker in local Item object list. We will also load the query vectors in a similar manner. As the total number of query vectors is usually small when compared to the number of item vectors, we could keep a local copy of all the query vectors in each worker to facilitate the iterative query processing. However, query vectors may be splitted and loaded by several workers and these workers will create a Query object list to store a set of query vectors that they read from HDFS. We can aggregate all the query vectors from all workers and make it available to every worker before the iterative processing. We call this procedure "query broadcasting". Then every worker would have a local copy of all the query vectors. This is extremely useful when later we calculate the distance between item and query vectors, as we do not need to send item or query vector through the network anymore, which save a lot of memory and bandwidth.

With hash functions, Item object lists and Query object lists all ready, now we can start to initialize buckets. Input each item vector from the Item object list to the hash functions to generate signatures for it. If the signature points to an existing bucket, then we just add the item ID to that item ID list in that bucket (each bucket maintains a list storing all item ID belonging to it). If the signature points to a non-existing bucket, a new corresponding Bucket object would be automatically created and the item ID would then be added to that bucket.

*3) Implement Multi-Probing in Iterative Manner:* In this section, we will present our easy-to-use query-answer scheme in iterative processing. The workflow is as follows: Query object calls query() function to find the Bucket object to probe and send personal information to it; Bucket object forward the information about the corresponding Query object to other Item object in this Bucket; Item object receives information about Query object first and then calls answer() function to calculate the distance and send all these data back to Query.

```
void initialize(const LSHFactory<ItemIdType, ItemElementType>& fty, Tree* tree) {
    int numTables = fty.getBand();
    handlers_.reserve(numTables);

    auto projs = fty.calProjs(this->getQuery());
    assert(projs.size() == numTables);
    for (unsigned t = 0; t < numTables; ++t) {

        vector<bool> hashBits(projs[t].size());
        for (int idx = 0; idx < hashBits.size(); ++idx) {
            if (projs[t][idx] >= 0 ) {
                hashBits[idx] = 1;
            } else {
                hashBits[idx] = 0;
            }
            projs[t][idx] = fabs(projs[t][idx]);
        }

        handlers_.emplace_back(TSTable(hashBits, projs[t], tree));
    }
}
```

Fig. 8: GQR Initialization

In the first iteration, we need to find the original bucket for each query and keep the information about the projected vector so that we will not apply hash functions to query vectors again. We call this process "Initialization". The corresponding code is shown above. Note that it requires a flipping vector tree to help decide which bucket to probe next. We only need to set up the flipping vector tree once and it works for all queries, as long as the length of query signature is the same as the number of levels in flipping vector tree. This property turns out to be surprisingly good in distributed implementation. The size of flipping vector tree can be pretty large considering that number of flipping vectors is exponential to the length of single flipping vector. We save a lot of space by sharing one copy of flipping vector tree. Therefore, in customized query function, we initialize flipping vector tree as follows.

```
// initliaze fvs
std::call_once(fvs_flag, [&fty]() {
        GLOBAL_Tree = new Tree(fty.getRow());
});
```

Fig. 9: GQR Initialize Flipping Vectors - 1

std::call_once() ensures that this piece of code would be executed exactly once, even if called concurrently, from several threads. Here we define Tree class in a separated header file.

The overall logic of our query function in the first iteration is as follows. Since Query objects in Query list will call query() function in every iteration and we do not need to calculate the signature after the first iteration, we add a "hasInited" tag to avoid redundant computing. We apply hash functions to query vector to get its signature, i.e. bucket ID of the bucket that it belongs to. Then the ID of this Query object would be sent to the specific Bucket object which is associated with a set of Item object (this step has been completed in the initialization phase before the iterative processing).

```
// initliaze fvs
std::call_once(fvs_flag, [&fty]() {
        GLOBAL_Tree = new Tree(fty.getRow());
});

if (!hasInited) {
    initialize(fty, GLOBAL_Tree);
    hasInited = true;
    this->queryMsg = this->getItemId();

    for (auto& bId : fty.calItemBuckets(this->getQuery())) {
        this->sendToBucket(bId);
    }
}
```

Fig. 10: GQR Initialize Flipping Vectors - 2

Bucket receives the messages from Query and then it knows which Query object belongs to it. It is responsible for forwarding the information of the Query to Item objects that are in the same bucket. Bucket plays a role as bridge, linking the similar Query objects and Item objects. In every iteration, what it does is the same, receiving and forwarding. The code is as follows.

```
// execute buckets
husky::list_execute(bucket_list,
    {&query2BucketCH}, {&bucket2ItemCH},
    [&factory, &query2BucketCH, &bucket2ItemCH](BucketType& bucket) {

        for (auto& msg : query2BucketCH.get(bucket)) {
            // forward query, should do message reduction
            for (auto& itemId : bucket.itemIds_) {
                bucket2ItemCH.push(msg, itemId);
            }
        }
});
```

Fig. 11: GQR Function: list_execute()

After the first two steps, we finally find the potential neighbors of the Query we are processing. Item object collect the Query IDs from buckets and evaluate the distance (we use Euclidean distance in our case) between the Query object and itself. This is the answer step in "query-answer" scheme. In our implementation, we use a unordered_set to contain the Query objects and we only calculate the distance between the two if the Query object has not been processed before. This guarantees no duplicate candidate items appearing in the answer. Finally, we will send the item id as well as the distance information back to Query object. This is the end of the first iteration.

In the second iteration or subsequent iterations, only the query() function we run is different from the one in the first iteration. We will manipulate the bucket id generated in the first iteration to probe more buckets to probe.

At the beginning of second iteration, we first receive the information sent from Item objects and thus know the ID of candidate item vectors as well as the exact distances. Then we need to check if the number of candidates has satisfied the requirement. If not, we will continue our querying process. If yes, we will stop here and do a sorting based on retrieved items and output the result. The checking should be conducted every iteration except the first iteration since there is no information sent from Item objects at that time. In fact, we implement two kinds of checking. First, each time we receive (id, distance) pair from items, we could append it to a container. And by checking the size of container, we would know whether we have retrieved enough candidates. In the last round, we do a sorting among all

these candidates and output the final result. Second, we use a min-heap structure to keep a constant number of the candidates in the heap so that we do not need to keep all the (id, distance) pairs. In this way, we could reduce used memory in the runtime and thus increase the scalability.

After the candidate checking, we would make use of projected vector (which has been calculated in the first iteration) and flipping vector tree to generate buckets on demand. The related code is as below. What moveForward() function does is basically popping the bucket with smallest quantization distance from the min heap and then inserting two children of it which are guaranteed to have larger quantization distances into the heap. In this way, we only keep a small number of (distance, id) pair in the min heap, which indeed saves a lot of memory compared sorting all quantization distances of buckets.

```
vector<int> sig(1);
for (int tb = 0; tb < handlers_.size(); ++tb) {
    if (handlers_[tb].moveForward()) {
        unsigned long long tmp = handlers_[tb].getCurBucket();
        sig[0] = (int)tmp;
        this->sendToBucket(sig, tb);
    }
}
```

Fig. 12: GQR: Probe Next Bucket

```
bool moveForward() {
    const unsigned& idx = minHeap_.top().index_;
    const float& score = minHeap_.top().score_;
    const unsigned& lastOnePos = tree_->getLastOne(idx);

    if (idx <= upperIdx) {
        // shift
        unsigned shiftIdx = 2 * idx + 1;
        const bool* shiftFV = tree_->getFV(shiftIdx);
        // float shiftScore = calScore(shiftFV);
        float shiftScore =
            score - posLossPairs_[lastOnePos].second + posLossPairs_[lastOnePos + 1].second;
        minHeap_.push(ScoreIdxPair(shiftScore, shiftIdx));

        // expand
        unsigned expandIdx = 2 * idx + 2;

        const bool* expandFV = tree_->getFV(expandIdx);
        // float expandScore = calScore(expandFV);
        float expandScore =
            score + posLossPairs_[lastOnePos + 1].second;
        minHeap_.push(ScoreIdxPair(expandScore, expandIdx));
    }
    minHeap_.pop();

    return !minHeap_.empty();
}
```

Fig. 13: GQR Function: moveForward()

Suppose that now we have popped a (distance, index)pair from the min heap, we could easily locate the corresponding flipping vector in the tree with the given index. Then we just apply the flipping vector to the original signature (i.e. original bucket ID) to generate a new potential bucket and send the query information to the specific bucket just as what we did in the first iteration.

We repeat the above operation until we have found enough number of candidates. Once we have done that, we will write the final result to HDFS. And that's the end of LSH.

*4) Sorting Objects and Messages:* In our implementation, each worker is processed by one thread. Workers will send messages to each other to communicate, and each worker maintains multiple message buffers for other workers.

During GQR computation, a worker will need to pass the messages that sent to its receiving objects by IDs. This can be implemented by hash Join, but it may cause high overhead due to random access. As we can see from Figure X, as a message list and a object list could be very large, when the worker try to merge all the messages to one object, it will be an expensive operation. In our implementation, we will pre-sort object IDs for each worker. And then, when messages come, we will sort these messages again depending on their target object IDs. Now the join operation will only take linear time and can improve the application performance.


Fig. 14: Without Sorting VS Sorting

*5) Bucket Compression:* Assume we have L hash tables and each hash table has B buckets (k is the total bit number for a signature, $B = 2^k$), then the total number of buckets will be $L * B$. In our implementation, we want to reduce the space complexity and in order to do that, we merge the $L * B$ buckets into B buckets. Those buckets having the same signature from different hash tables will be merged into one bucket. In the new bucket, it will maintain multiple list (at most L list) for each hash table. Furthermore, for each list, we will group the item IDs in it depending on which worker it is stored at, and then, we will sort the item IDs in each group.

Using this implementation mainly has the following advantages. First, message sent to multiple buckets with a same signature will only sent to one bucket now. Here, we reduce the message sent among the network. Second, when the application need to forward messages to item object, it can directly get the item ID list on different workers and send the message to the worker only once. Third, when a worker receives messages from multiple buckets, it can directly merge those message, which can easily remove those duplicate queries to each item. As the Item IDs are sorted, a simple linear scan can help to quickly finish the merge operation.

*6) Message Reduction:* One major source of messages is sending query vector to various buckets and then forward them to different items. To handle this issue, we simply broadcast query vectors on all machine, which can help us prevent to send the actual query vector again and again. In our application, we only need to send the query ID through the network, which greatly reduce the communication cost.

When sending query ID to the corresponding buckets, we can also do some message reduction. A query may send messages to multiple buckets in multiple workers. If these workers are in the same physical machine, then, our application will pack the messages into one and only send one message, which

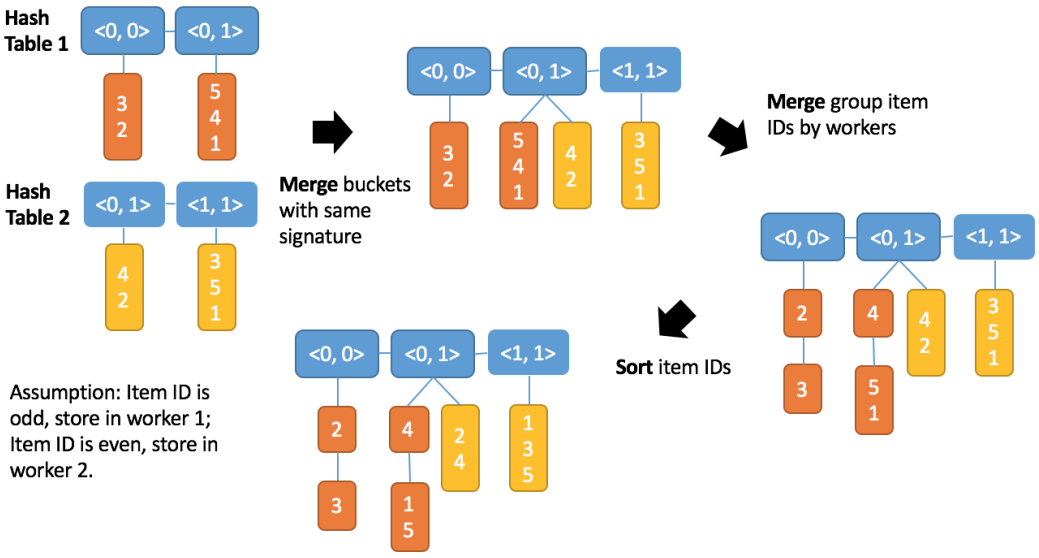



Fig. 15: Bucket Compression

contains a list of target bucket IDs and target worker IDs, to target machine. When the corresponding machine receive the message, it will create a new thread to unpack the message and send the message to multiple buckets in multiple workers.

Similarly, when forwarding message from bucket to item object, we implement a two-level combiner. Assuming that we have multiple buckets on a same machine will forward a message to the same item. We will first combine the messages send to the same item object in worker level. Then, we will further combine the messages for the same query in machine level. And finally, we will combine the messages that are sent to multiple items in multiple workers in the same machine.

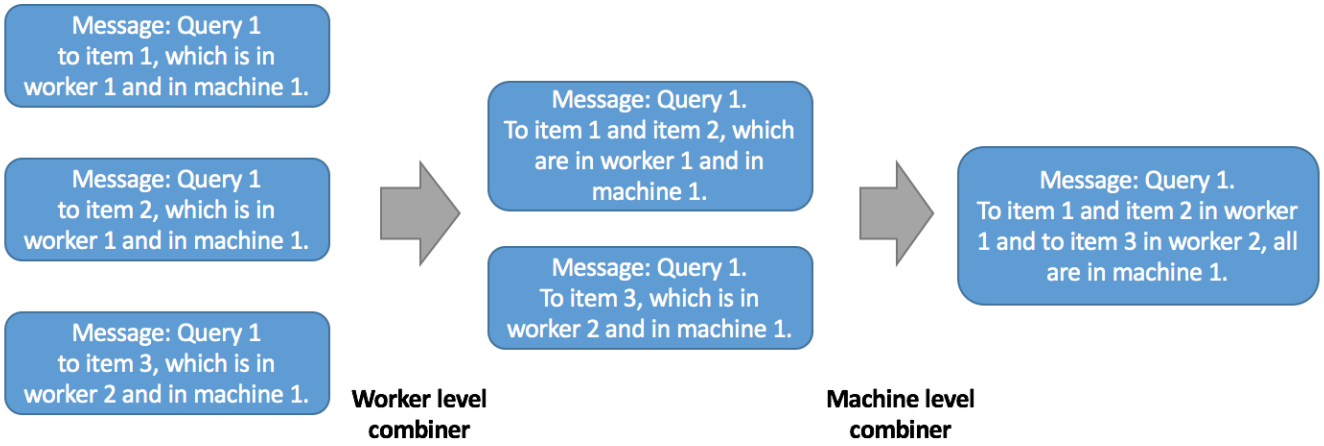



Fig. 16: Message Reduction

### C. *Experiments*

We have shown that GQR outperforms both Hamming Ranking on precision as well as the recall time on small datasets, which can be explained by the fact that QD is a more fine-grained and better similarity indicator than Hamming distance. In this section, we lay emphasis on verifying the scalability

of our distributed implementation. We used three datasets with different sizes and number of features, as shown below.

| Datasets | Dimensions | Number of data objects |
|---|---|---|
| GIST1M | 960 | 1,000,000 |
| TINY5M | 384 | 5,000,000 |
| SIFT10M | 128 | 10,000,000 |

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. 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. For each worker, we set the number of threads to 20.

| workers | GIST1M | TINY5M | SIFT10M |
|---|---|---|---|
| 1 worker | 215s | fail | fail |
| 3 workers | 112s | 482s | fail |

We double the number of expected candidate buckets after each iteration. For example, in the first iteration, we fetch one bucket. And we will fetch 2 buckets in next iteration and 4 buckets in third iteration. When the number of buckets in one iteration is very large, the worker will have difficulty in allocating the memory for large vectors. However, as our algorithm has been implemented on distributed platforms, we could easily scale up processing. We noticed that one worker cannot handle TINY5M datasets and it will fail before last round. We assign more workers to share the workload. Because of the optimization work we have done on data communication, taking in more workers does not introduce high communication cost, thus boost the querying performance. When the size of dataset increase to 10M, three workers is no longer sufficient, then we could use a larger cluster, in this case, 10 workers, SIFT10M could be handled by 10 workers in 7 minutes. Here we can see that the size of datasets would not be a big problem in distributed given the scalability of this programming framework.

## ACKNOWLEDGMENT

This is a joint work with Jinfeng Li, Xiao Yan and Prof. James Cheng. The main idea of the work has been accepted into SIGMOD 2018. We would like to express our special thanks of gratitude to them who gave us the golden opportunity to join this research work.

## REFERENCES

[1] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP*, pages 331–340, 2009.

[2] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. volume 18, pages 509–517, 1975.

[4] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[5] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. volume 36, pages 2227–2240, 2014.

[6] Cui Yu. *High-Dimensional Indexing: Transformational Approaches to High-Dimensional Range and Similarity Searches*, volume 2341 of *Lecture Notes in Computer Science*. Springer, 2002.

[7] Jinfeng Li, James Cheng, Fan Yang, Yuzhen Huang, Yunjian Zhao, Xiao Yan, and Ruihao Zhao. Losha: A general framework for scalable locality sensitive hashing. In *SIGIR*, pages 635–644, 2017.

[8] Kaiming He, Fang Wen, and Jian Sun. K-means hashing: An affinity-preserving quantization method for learning binary compact codes. In *CVPR*, pages 2938–2945, 2013.

[9] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, pages 2946–2953, 2013.

[10] Yair Weiss, Antonio Torralba, and Robert Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.

[11] Yunchao Gong and Svetlana Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, pages 817–824, 2011.

[12] Jun Wang, Ondrej Kumar, and Shih-Fu Chang. Semi-supervised hashing for scalable image retrieval. In *CVPR*, pages 3424–3431, 2010.

[13] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[14] Bahman Bahmani, Ashish Goel, and Rajendra Shinde. Efficient distributed locality sensitive hashing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2174–2178. ACM, 2012.