

PQTable: Fast Exact Asymmetric Distance Neighbor Search for Product Quantization using Hash Tables

Yusuke Matsui Toshihiko Yamasaki Kiyoharu Aizawa
 The University of Tokyo, Japan

{matsui, yamasaki, aizawa}@hal.t.u-tokyo.ac.jp

Abstract

We propose the product quantization table (PQTable), a product quantization-based hash table that is fast and requires neither parameter tuning nor training steps. The PQTable produces exactly the same results as a linear PQ search, and is 10^2 to 10^5 times faster when tested on the SIFT1B data. In addition, although state-of-the-art performance can be achieved by previous inverted-indexing-based approaches, such methods do require manually designed parameter setting and much training, whereas our method is free from them. Therefore, PQTable offers a practical and useful solution for real-world problems.

1. Introduction

With the explosive growth of multimedia data, compressing high-dimensional vectors and performing approximate nearest neighbor (ANN) searches in the compressed domain is becoming a fundamental problem in handling large databases. Product quantization (PQ) [13], and its extensions [17, 7, 2, 24, 21, 11, 5, 25], are popular and successful methods for quantizing a vector into a short code. PQ has three attractive properties: (1) PQ can compress an input vector into an extremely short code (e.g., 32 bit) that enables it to handle typically one billion data points in memory at once; (2) the approximate distance between a raw vector and a compressed PQ code can be computed efficiently (the so-called *asymmetric distance computation* (ADC) [13]), which is a good estimate of the original Euclidean distance; and (3) the data structure and coding algorithms are surprisingly simple. By quantizing database vectors into short codes in advance, vectors similar to a given query can be found from the database codes by a linear comparison of the query with each code using ADC (see the linear ADC scan in Fig. 1a).

Although a linear ADC scan is simple and easy to use, ANN search by linear ADC scanning is efficient only for small datasets because the search is exhaustive, i.e., the computational cost is at least $O(N)$ for N PQ codes. To

handle large (e.g., $N = 10^9$) databases, *short-code-based inverted indexing systems* [1, 23, 7, 15, 3, 4] have been proposed, which are currently state-of-the-art ANN methods (see Fig. 1b). Such systems operate in two stages: (1) coarse quantization and (2) reranking via short codes. A database vector is first assigned to a bucket using a coarse quantizer (e.g., k-means [13], multiple k-means [23], or Cartesian products [1, 12]), then compressed to a short code using PQ [13] or its extensions [17, 7] and finally stored as a posting list¹. In the retrieval phase, a query vector is assigned to the nearest buckets by the coarse quantizer, then associated items in corresponding posting lists are traversed, with the nearest one being reranked via ADC². These systems have been reported as being fast, accurate, and memory efficient, by being capable of holding one billion data points in memory and conducting a retrieval in milliseconds.

However, such inverted indexing systems are built by a process of carefully designed manual parameter tuning. The processing time and recall rates strongly depend on the number of space partitions in the coarse quantizer, and its optimal size needs to be decided manually, as shown in Fig. 2. For example, this simple example shows that $k = 1024$ is the fastest setting for $N = 10^7$, but the slowest for $N = 10^8$. These relationships do not become clear until several parameter combinations for the coarse quantizers are examined. However, training and testing coarse quantizers is computationally expensive. For example, we found that, to plot a single dot of Fig. 2 for $N = 10^9$, the training took around a day and building the inverted index structure took around three days, using 16-core machines. This is particularly critical for a recent per-cell training ANN system [3, 15], which requires even more computation to build the system. It should be pointed out that achieving state-of-the-art performances by ANN systems depends largely on special tuning for the testbed dataset, e.g., SIFT1B [14]. There is no guarantee that we can always achieve the best performance by such systems. In real-world applications,

¹In practice, the residual difference between the query vector and the codeword is encoded.

²There are smarter versions of this computation [3].

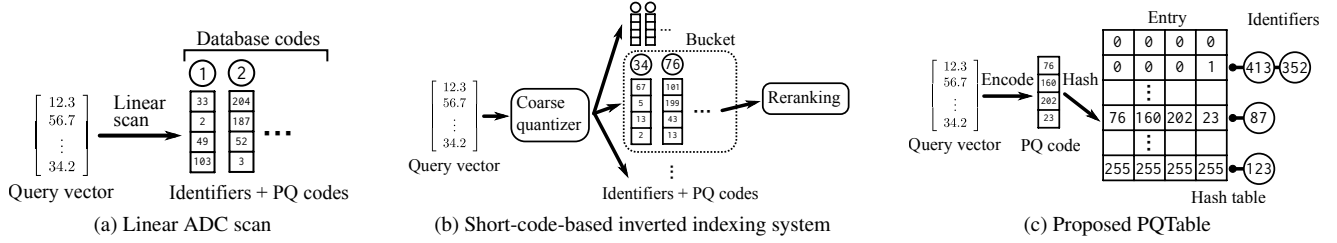


Figure 1: Data structures of ANN systems: linear ADC scan, short-code-based inverted indexing systems, and PQTable.

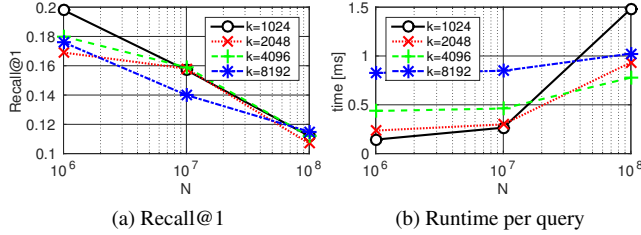


Figure 2: Performance of a short-code-based inverted indexing system (IVFADC [13]), showing the relationship between the database size, N , and Recall@1 or runtime per query for various k , which is the number of space partitions for the coarse quantizer. The search range w is set to one. Note that the search range is also an important parameter, which has been chosen manually.

it often happens that cumbersome trial-and-error-based parameter tuning is required, which might well turn ordinary users away from these systems.

To achieve an ANN system that would be suitable for practical applications, we propose the *PQTable*, an exact, nonexhaustive, NN search method for PQ codes (see Fig. 1c). We do not employ an inverted index data structure, but find similar PQ codes directly from the database. This achieves exactly the same accuracy as a linear ADC scan, but requires a significantly shorter time (10.3 ms, instead of 18 s, for the SIFT1B data using 64-bit codes). In other words, this paper proposes an efficient ANN search scheme to replace a linear ADC scan when N is sufficiently large. As discussed in Section 4, the parameter values required to build the PQTable can be calculated automatically. Note that, as discussed in Section 5, PQTable requires larger memory footprints than storing PQ codes linearly.

The main idea of the PQTable is the use of a hash table, where the PQ code itself is used directly as an entry in the hash table (see Fig. 1c). That is, identifiers of PQ codes in the database are directly associated with the hash table by using the PQ code itself as an entry. Given any new query vector, it can be PQ coded and identifiers associated with the entry are retrieved.

There are advantages and disadvantages in the use of inverted indexing systems as well as for our proposed method.

Inverted indexing systems are accurate and fast, but they do require manual parameter tuning and time-consuming training. The PQTable can achieve the same recall rates as a linear ADC, but its performance (accuracy, runtime, and memory usage) may lag the latest inverted indexing systems with the best parameters. The PQTable does not require any parameter tuning or training steps for the coarse quantizers, which are important aspects of real-world applications.

Relationship with previous methods

Hamming-based ANN systems: As an alternative to PQ-based methods, another major approach to ANN are the Hamming-based methods [22, 8, 20], in which two vectors are converted to bit strings whose Hamming distance approximates their Euclidean distance. Comparing bit strings is faster than comparing PQ codes, but is usually less accurate for a given code length [10].

In Hamming-based approaches, bit strings can be linearly scanned by computing their Hamming distance, which is similar to linear ADC scanning in PQ. In addition, a multitable algorithm has been proposed to facilitate fast ANN search [18], which computes exactly the same result as a linear Hamming scan but is much faster. However, a similar querying algorithm and data structure for fast computing have not been proposed for the PQ-based method to date. Our work therefore extends the idea of such a multitable algorithm [18, 9] to PQ codes, where a Hamming-based formulation cannot be applied directly.

Extensions to PQ: Recent reports [24, 2, 5, 25] have proposed a new encoding scheme that is regarded as a generalized version of PQ. Because the data structure used in these approaches is similar to PQ, our proposed method can also be applied to such approaches. In this paper, we discuss only the core of the PQ process for simplicity.

2. Background: Product Quantization

In this section, we briefly review PQ [13].

Product quantizer: Let us denote any $\mathbf{x} \in \mathbb{R}^D$ as a concatenation of M subvectors: $\mathbf{x} = [\mathbf{x}^1, \dots, \mathbf{x}^M]$, where we assume that the subvectors have an equal number of dimensions D/M , for simplicity. A product quantizer $q(\cdot)$ is

defined as a concatenation of subquantizers:

$$\mathbf{x} \mapsto \mathbf{q}(\mathbf{x}) = [\mathbf{q}^1(\mathbf{x}^1), \dots, \mathbf{q}^M(\mathbf{x}^M)], \quad (1)$$

where each subquantizer $\mathbf{q}^m(\cdot)$ is defined as

$$\mathbf{x}^m \mapsto \mathbf{q}^m(\mathbf{x}^m) = \arg \min_{\mathbf{c}^m \in \mathcal{C}^m} \|\mathbf{x}^m - \mathbf{c}^m\|. \quad (2)$$

Each subcodebook \mathcal{C}^m comprises K D/M -dim centroids learned by k-means [16]. Therefore, $\mathbf{q}(\mathbf{x})$ maps an input \mathbf{x} to a codeword $\mathbf{c} = [\mathbf{c}^1, \dots, \mathbf{c}^M] \in \mathcal{C} = \mathcal{C}^1 \times \dots \times \mathcal{C}^M$.

Because \mathbf{c} is encoded as a tuple (i^1, \dots, i^M) of M subcentroid indices, where each $i^m \in \{0, \dots, K-1\}$, it can be represented by $M \log_2 K$ bits. Typically, K is set as a power of 2, making $\log_2 K$ an integer. In this paper, we set K as 256 (this is the typical setting used in many papers, for which \mathbf{c}^m is represented by 8 bits), and M is set as 1, 2, 4, or 8, which yields a length for the code of 8, 16, 32, or 64 bits, respectively.

ADC: Distances between encoded vectors and a raw vector can be computed efficiently. Suppose that there are N data points $\mathcal{Y} = \{\mathbf{y}_j \in \mathbb{R}^D | j = 1, \dots, N\}$ that have been encoded and stored. Given a new query vector $\mathbf{x} \in \mathbb{R}^D$, the squared Euclidean distance from \mathbf{x} to a point \mathbf{y} is approximated by asymmetric distance (AD):

$$d_{AD}(\mathbf{x}, \mathbf{y})^2 = \sum_{m=1}^M d(\mathbf{x}^m, \mathbf{q}^m(\mathbf{y}^m))^2. \quad (3)$$

This can be computed as follows. First, \mathbf{x}^m is encoded by its corresponding subcodebook, \mathcal{C}^m , thereby generating a distance table whose size is $M \times K$, where each (m, k) entry denotes the squared Euclidean distance between \mathbf{x}^m and the k -th subcentroid in \mathcal{C}^m , namely \mathbf{c}_k^m . Suppose that a subvector \mathbf{y}^m of each \mathbf{y} is encoded in advance as \mathbf{c}_k^m , i.e., $\mathbf{q}^m(\mathbf{y}^m) = \mathbf{c}_k^m$. It follows that an (m, k) in the distance table means $d(\mathbf{x}^m, \mathbf{c}_k^m)^2 = d(\mathbf{x}^m, \mathbf{q}^m(\mathbf{y}^m))^2$. We can then compute Eq. (3) by simply looking up the distance table M times and summing the distances. The computational cost of the whole process is $O(KD + NM)$, which is fast for small N but still linear in N .

3. PQTable

As shown in Fig. 1c, the proposed system involves a hash table within which the PQ code itself is used as an entry. In the retrieval phase, a query vector is PQ coded and identifiers associated with the entry are retrieved. This is very straightforward, but there are two problems to be solved.

The empty-entries problem: Suppose that a new query is PQ coded and hashed. If the identifiers associated with the entry of the query are not present in the table, the hashing fails. To continue the retrieval, we would need to find candidates for the query by some other means.

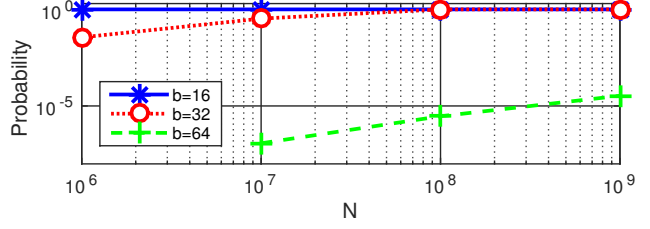


Figure 3: The probability of finding the first nonempty entry (the first identifier). Plots are computed over 50 queries from the SIFT1B data for various code lengths b .

The long-code problem: Even if we can find the candidates and continue querying, the retrieval is not efficient if the length of the codes is very long compared to the size of the database, e.g., 64-bit codes for 10^9 vectors. Fig. 3 shows the relationship between the number of database vectors and the probability of finding the first nonempty entry. If the length of the codes b is 16 or 32 bits, there is a high probability that the first identifiers can be found by the first hashing. However, for the case where $b = 64$, candidate entries need to be generated 10^5 times to find the first nonempty entry. This is caused by the wide gap between the number of possible candidates, 2^b , which measures the number of entries in the table, and the number of database vectors. This imbalance causes almost all entries to be empty³.

To handle these two issues, we first present a **candidate generation scheme**, which is mathematically equivalent to a multisequence algorithm [1], to solve the empty-entries problem. Then we propose **table division and merging** to handle the long-code problem.

We first show the data structure and a querying algorithm for a single table. To be able to use a single table, we need to handle the empty-entries problem, so we present a candidate-generation scheme to solve it. Then, because a single table cannot handle large codes (the long-code problem), we propose table division and merging to handle codes of any length.

3.1. Single Hash Table

The basic idea of the proposed method is to use a hash table, as shown in Fig. 4a. We prepare a hash table comprising the PQ codes themselves, i.e., a table entry is a tuple of 8-bit integers. During offline processing in advance, the j -th database vector \mathbf{y}_j is compressed by PQ. The resultant PQ code (i_j^1, \dots, i_j^M) is used directly as an entry in the hash table and its identifier j is associated with the entry. If several vectors are associated with the same entry, as shown for “413” and “352” in Fig. 4a, both of them are stored. Note that the distances from the query to these vectors are the same because they are quantized as the same code. In on-line processing, an input query vector \mathbf{x} is compressed to

³This is a similar problem to that of designing a coarse quantizer for an inverted indexing-based ANN system, but we found an optimal solution empirically, as discussed later.

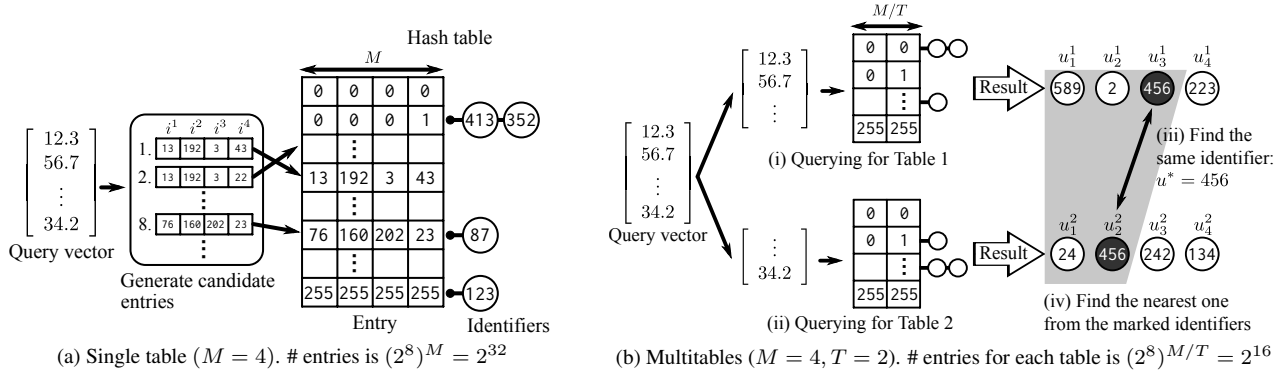


Figure 4: Overview of the proposed method.

(i^1, \dots, i^M) by PQ, and identifiers associated with the PQ code of the query are searched.

If an identifier is found ($\exists j, i^m = i_j^m$ for all m), it is obviously the nearest entry in terms of AD because the code-word that is the nearest to the query is (i^1, \dots, i^M) , by definition. A problem arises if there are no identifiers associated with the PQ code of the query, e.g., $(13, 192, 3, 43)$ is the PQ code of a query for which there is no associated identifier in the hash table in Fig. 4a. This is the empty-entries problem described above. In such cases, we need to find a candidate entry for the PQ code, which will be the second nearest to the query in terms of AD. To find the top K nearest identifiers, the search continues for candidates until K identifiers are found, giving a set of results that are sorted automatically in terms of AD.

Given a query vector, how can candidate entries be generated? We present a candidate-generation scheme (the “Generate candidate entries” box in Fig. 4a). Given a query vector, it produces candidate codes that are sorted in ascending order of AD. This scheme is mathematically equivalent to the higher-order multisequence algorithm proposed by Babenko and Lempitsky [1]. They used this algorithm originally to divide the space into Cartesian products for coarse quantization. We found that it can be used to enumerate all possible PQ code combinations in ascending order of AD. We give the pseudocode for the multisequence algorithm used in our implementation in the supplementary materials.

Pseudocode for an algorithm that queries a single table is shown in Alg. 1, where x denotes the input vector, $table$ is the PQTable filled with database PQ codes, and K is the user-specified number of output identifiers. The output $u[]$ is a sequence of identifiers ordered in terms of AD between the query and the database vectors. $cand$ is a priority queue composed of PQ code, and used for the multisequence algorithm. The norm $|\cdot|$ of a sequence means the number of elements of the sequence. $Hash(\cdot)$ fetches items using the input code as an entry and returns a set of associated identifiers. $Insert(\cdot)$ simply adds items to the end of the ar-

Algorithm 1: Querying for a single table.

Input: $x, table, K$
Output: $u[]$: ranked identifiers

```

1  $u[] \leftarrow \emptyset$  // array
2  $cand \leftarrow Init(x)$  // priority queue
3 while  $|u| < K$  do
4    $code \leftarrow NextCode(cand)$ 
5    $\{u'\} \leftarrow table.Hash(code)$ 
6    $u.Insert(\{u'\})$ 

```

ray. We omit d_table and returned $cand$ from $NextCode$, for simplicity. Please refer to the implementation details for the multisequence algorithm ($Init$ and $NextCode$) [1] in the supplementary materials.

3.2. Multiple Hash Table

The single-table version of PQTable may not work when there are long PQ codes, such as $8 \leq M$ (64 bits). This is the long-code problem described above, where the number of possible entries is too large for efficient processing. The number of entries is $(2^8)^M$ for an $8M$ bit code, and the number of database vectors can be at most one billion. Therefore, most of the entries will be empty if M is 8 or greater ($(2^8)^8 = 2^{64} \sim 1.8 \times 10^{19} \gg 10^9$).

To solve this problem, we propose a table division and merging method. The table is divided as shown in Fig. 4b. If an $8M$ -bit code table is divided into two $4M$ -bit code tables, the number of the entry decreases, from $(2^8)^M = 256^M$ for one table to $(2^8)^{M/2} = 16^M$ for two tables. By properly merging the results from each of the small tables, we can obtain the correct result.

If we have b -bit PQ codes for the database vectors, we prepare T b/T -bit tables. During offline processing in advance, the j -th PQ code is divided by T and inserted in the corresponding table. In online processing, we first compute ranked identifiers from each table in the same way as for the single table case (see Fig. 4b (i, ii)).

Denote the ranked identifiers from the t -th table as $\{u_n^t\}$, where $t = 1, \dots, T$, and $n = 1, 2, \dots$. We can then check the identifiers from the head ($n = 1$) until the same identifier u^* is found from all of the tables, i.e., $u_{n_1}^1 = u_{n_2}^2 = \dots = u_{n_T}^T \equiv u^*$, where n_t^* means u^* is found in the n_t^* -th element of the t -th ranked identifiers (see Fig. 4b (iii)). If the same identifier is found, we pick up all of the checked identifiers, i.e., $u_1^t, \dots, u_{n_t^*-1}^t$ for all t , referred to as the **marked identifiers** (shaded gray in Fig. 4b) and compute their AD to the query. It is guaranteed that all identifiers whose AD is less than $d_{AD}(\mathbf{x}, \mathbf{y}_{u^*})$ will be included among the marked identifiers (see the supplementary material for proof). Therefore, by simply sorting the marked identifiers in terms of AD, we can find all of the nearest identifiers in the database with AD less than $d_{AD}(\mathbf{x}, \mathbf{y}_{u^*})$ (see Fig. 4b (iv)). This process is repeated until the desired number of output identifiers is obtained.

Pseudocode for querying multiple PQTables is shown in Alg. 2. Unlike the single-table case, $tables[]$ is an array of tables. $u_dist[]$ and $marked[]$ are each an array of tuples, comprising an identifier and its AD to the query. $u_dist[]$ is the final result and $marked[]$ is used as a buffer. $Split(\cdot, T)$ divides the priority queues into T smaller priority queues. $Check(\cdot)$ and $UnCheck(\cdot)$ are used to record how many times identifiers have appeared in the process. They are implemented by an associative array in our method. Because each identifier is included once in each table, and therefore can be checked at most T times. An identifier that has appeared in all tables can be detected by seeing if it has been checked T times or not. $PartialSortByDist(\cdot, K)$ sorts an array of tuples in terms of distance, obtaining the top K results at a computational cost of $O(N \log K)$, where N is the length of the array. The function $array.Less(d)$ returns those elements whose distances are less than d . Note that $d_{AD}(i)$ is an abbreviation of $d_{AD}(\mathbf{x}, \mathbf{y}_i)$.

4. Empirical Analysis for Parameter Selection

In this section, we discuss how to determine the value of the parameter required to construct the PQTable. Suppose that we have N PQ-coded database vectors of code length b . To construct the proposed PQTable, we need to decide one parameter value T , which is the number of dividing tables. If $b = 32$, for example, we need to select the data structure as being either a single 32-bit table, two 16-bit tables, or four 8-bit tables, corresponding to $T = 1, 2$, and 4, respectively. We first show that the statistics of the assigned vectors are strongly influenced by the distribution of the database vectors, and present an indicative value, $b/\log_2 N$, as proposed by previous work on multitable hashing [9, 18], that estimates the optimal T .

Because the proposed data structure is based on a hash table, there is a strong relationship between N , the number of entry 2^b , and the computational time. If N is small,

Algorithm 2: Querying for multitable.

Input: $\mathbf{x}, tables[], K$
Output: $u_dist[]$: ranked identifiers with distances

```

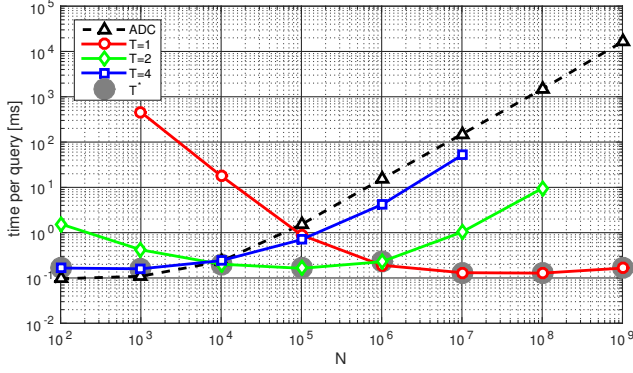
1  $u\_dist[] \leftarrow \emptyset$  // array
2  $T \leftarrow |tables|$ 
3  $cands[] \leftarrow Split(Init(\mathbf{x}), T)$  // priority queues
4  $marked[] \leftarrow \emptyset$  // array
5 while  $|u\_dist| < K$  do
6   for  $t \leftarrow 1$  to  $T$  do
7      $code \leftarrow NextCode(cands[t])$ 
8      $\{u^t\} \leftarrow tables[t].Hash(code)$ 
9     for  $u \in \{u^t\}$  do
10       $Check(u)$ 
11      if  $u$  has been checked  $T$  times then
12         $\{i'\} \leftarrow \text{Pick up checked identifiers}$ 
13        for  $i \in \{i'\} \cup u$  do
14           $marked.Insert(tuple(i, d_{AD}(i)))$ 
15           $UnCheck(i)$ 
16         $PartialSortByDist(marked, K)$ 
17         $u\_dist.Insert(marked.Less(d_{AD}(u)))$ 

```

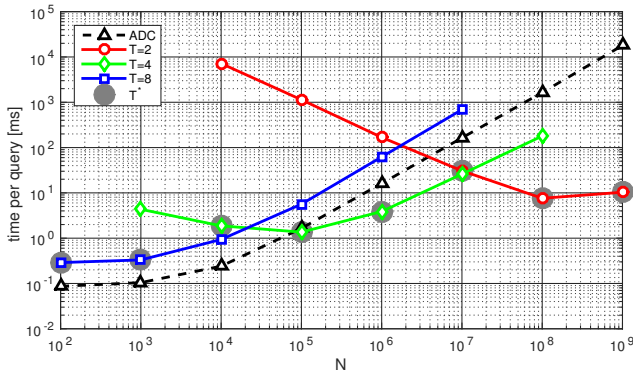
almost all of the entries in the table will not be associated with identifiers, and generating candidates will take time. If N is appropriate and the entries are well filled, the search is fast. If N is too large compared with the size of the entry, all entries are filled and the number of identifiers associated with each entry is large, which can again cause slow fetching. We show the relationship between N and the computational time for 32-bit codes and $T = 1, 2$, and 4 in Fig. 5a, and for 64-bit codes and $T = 2, 4$, and 8 in Fig. 5b. Note that these graphs are log-log plots. For example, the table with $T = 1$ computes 10^5 times faster than a linear ADC scan when $N = 10^9$ in Fig. 5a. From these figures, each table has a “hot spot.” In Fig. 5a, for $N = 10^2$ to 10^3 , 10^4 to 10^5 , and 10^6 to 10^9 , $T = 4, 2$, and 1 are the fastest, respectively. Given N and b , we need to decide the optimal T without actually constructing tables.

We can show that the statistics in the table are strongly influenced by the distribution of the input vectors. Consider the average number of candidates generated s in finding the nearest neighbor. For the case of a single table, this represents the number of iterations of the **while** loop in Alg. 1. If all vectors are equally distributed among the entries, we can estimate the average number \bar{s} by a fill factor $p \equiv N/2^b$, which shows how well the entries are filled. Because the probability of finding the identifiers for the first time in the s -th step is $(1 - p)^{s-1}p$ when $p < 1$, the expected value of s for finding the item for the first time is

$$\bar{s} = \begin{cases} \sum_{s=1}^{\infty} (1 - p)^{s-1} p s = \frac{1}{p} & (\text{if } p < 1) \\ 1 & (\text{if } p \geq 1). \end{cases} \quad (4)$$



(a) 32-bit PQ codes from the SIFT1B data.



(b) 64-bit PQ codes from the SIFT1B data.

Figure 5: Runtime per query of each table.

This estimated number and the actual number required to find the identifiers, evaluated for the SIFT1B data, are shown in Table 1. There is a very large gap between the estimated and actual number (roughly 10 times) for all N . In addition, the table gives the number of assigned items for the entry when the nearest items are found, which corresponds to $\{u'\}$ in Alg. 1 when hashing returns identifiers for the first time. This number is surprisingly large when N is very large. For example, for the case of $N = 10^9$, the first candidate usually has associated identifiers (s is actually 1.1), with 1.1×10^4 identifiers being assigned to that entry. This is surprising because, if the identifiers were distributed equally, the estimated step s would be 4.3, and the number of associated identifiers would be less than 1 for each entry. This heavily biased behavior, caused by the distribution of the input vectors, prevented us from theoretically analyzing the relationship between the various parameters. Therefore, we chose to employ an empirical estimation procedure from the previous literature, which was both simple and practical.

The literature on multitable hashing [18, 9] suggests that an indicative number, $b/\log_2 N$, can be used to divide the table. Our method is different from those in previous studies because we are using PQ codes. However, this indicative number can provide a good empirical estimate of the optimal T in our case. Because T is a power of two in the pro-

Table 1: Estimated and actual number of steps to find an item for the first time, and actual number of retrieved items for the first entry, for 32-bit codes from the SIFT1B data.

	N	10^4	10^5	10^6	10^7	10^8	10^9
# steps	Actual	2.8×10^4	2.5×10^3	1.7×10^2	25	6.0	1.1
	Estimate(\bar{s})	4.3×10^5	4.3×10^4	4.3×10^3	4.3×10^2	43	4.3
# items	Actual	1.1	2.1	12	1.2×10^2	1.2×10^3	1.1×10^4

Table 2: Estimated and actual T for the SIFT1B data.

	N	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
$b = 32$	Actual	4	4	2	2	1	1	1	1
	T^*	4	4	2	2	2	1	1	1
$b = 64$	Actual	8	8	8	4	4	4	2	2
	T^*	8	8	4	4	4	2	2	2

posed table, we quantize the indicative number into a power of two, and the final optimal T^* is given as

$$T^* = 2^{Q(\log_2(b/\log_2 N))}, \quad (5)$$

where $Q(\cdot)$ is the rounding operation. A comparison with the actual number is shown in Table 2. In many cases, the estimated T^* is a good estimation of the actual number, and the error was small even if the estimation failed, such as in the case of $b = 32$ and $N = 10^6$ (see Fig. 5a). Selected T^* values are plotted as gray dots in Fig. 5a and Fig. 5b.

5. Experimental Results

We present our experimental results using the proposed method. The evaluation was performed using the SIFT1B data from the BIGANN dataset [14], which contain one billion 128-D SIFT features and provide learning and querying vectors, from which we used 10 million for learning and 1,000 for querying. The centers of the PQ codes were trained using the learning data (this learning process is required for all PQ-based methods), and all database vectors were converted to 32- or 64-bit PQ codes. Note that we selected T using Eq. (5). All experiments were performed on a notebook PC with a 2.8 GHz Intel Core i7 CPU and 32 GB RAM. Results using another data, GIST1M, are presented in the supplementary materials, which shows the same tendency as for SIFT1B.

Speed: Figure 6 shows the runtimes per query for the proposed PQTable and a linear ADC scan. The results for 32-bit codes are presented in Fig. 6a and Fig. 6b. The ADC runtime depends linearly on N and required 16.4 s to scan $N = 10^9$ vectors, but the runtime for the PQTable was less than 1 ms for all cases. When the database size N was small, the ADC was faster. Note that the time to encode the query vector was more significant than the retrieval itself for $N \leq 10^2$. The proposed method was faster for the case where $N > 10^5$, and was approximately 10^5 times faster than the ADC for $N = 10^9$, even for the 1-, 10-, and 100-NN cases. Fig. 6c and Fig. 6d show the results for 64-bit PQ codes. The speedup was less dramatic than for the 32-bit cases, but was still 10^2 to 10^3 times faster than the ADC,

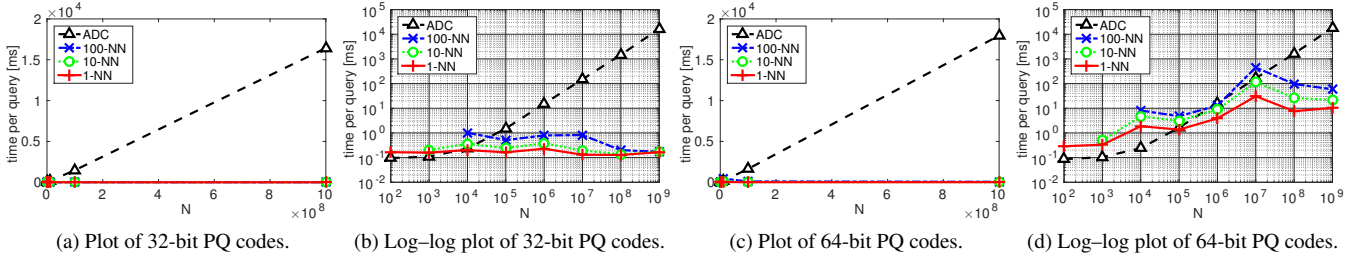


Figure 6: Runtimes per query for the proposed PQTable with 1-, 10-, and 100-NN, and a linear ADC scan (SIFT1B data).

Table 3: Speed-up factors for $N = 10^9$.

Speed-up factors for k-NN vs. ADC				
b	1-NN	10-NN	100-NN	ADC
32	1.0×10^5	9.7×10^4	9.5×10^4	16.4 s
64	1.7×10^3	8.4×10^2	3.1×10^2	18.0 s

particularly for $N = 10^8$ to $N = 10^9$, where the runtime was 10.3 ms for $N = 10^9$ in the 1-NN case. We report the speedup factors with respect to the linear ADC in Table 3.

Accuracy: Accuracy results for the proposed method are: Recall@1=0.001, @10=0.019, and @100=0.071, for 32-bit codes, with Recall@1=0.067, @10=0.249, and @100=0.569, for 64-bit codes, for $N = 10^9$. Note that all values are the same as for the linear ADC scan case.

Memory usage: We show the estimated and actual memory usage of the PQTable in Fig. 7. For the case of a single table, the theoretical memory usage involves the identifiers (4 bytes for each) in the table and the centroids of the PQ codes, a total of $4N + 4KD$ bytes. For multi-table cases, each table needs to hold the identifiers, and the PQ codes are also required for computing d_{AD} . Here, the final memory usage is $(4T + b/8)N + 4KD$ bytes. These are theoretical lower-bound estimates. As a reference, we also show the cases when PQ codes are linearly stored for the linear ADC scan ($bN/8 + 4KD$ bytes) in Fig. 7.

For the $N = 10^9$ with $b = 64$ case, the theoretical memory usage is 16 GB, and the actual cost is 19.8 GB. This difference comes from an overhead for the data structure of hash tables. For example, for 32-bit codes in a single table, directly holding 2^{32} entries in an array requires 32 GB of memory, even if all of the elements are empty. This is because a NULL pointer requires 8 bytes with a 64-bit machine. To achieve more efficient data representation, we employed a sparse direct-address table [18] as the data structure, which enabled the storage of 10^9 data points with a small overhead and provided a worst-case runtime of $O(1)$.

When PQ codes are linearly stored, it requires only 8 GB for $N = 10^9$ with $b = 64$. Therefore, we can say there is a trade-off among the proposed PQTable and the linear ADC scan in terms of runtime and memory footprint (18.0 s with 8GB v.s. 10.3 ms with 19.8 GB).

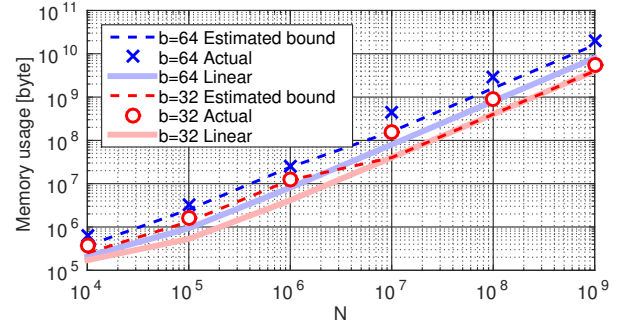


Figure 7: Memory usage for the tables using the SIFT1B data. The dashed lines represent the theoretically estimated lower bounds. The circles and boxes represent the actual memory consumption for 32- and 64-bit tables. We also show the linearly stored case for the ADC scan.

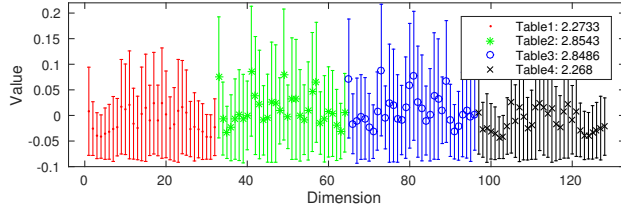
Distribution of each component of the vectors: We investigated how the distribution of vector components affects the search performance, particularly for the multi-table case. We prepared the SIFT1M data, using principal-component analysis (PCA), to have the same dimensionality (128). Then both the original SIFT data and the PCA-aligned SIFT data were zero-centered and normalized to enable a fair comparison. Finally, the average and standard deviation for all data were plotted, as shown in Fig. 8. In both cases, a PQTable with $T = 4$ was constructed and dimensions associated with each table were plotted using the same color. Note that the sum of the standard deviation for each table is shown in the legends.

As shown in Fig. 8a, the values for each dimension are distributed almost equally, which is the best case for our PQTable. On the other hand, Fig. 8b shows a heavily biased distribution, which is not desirable because elements in Tables 2, 3, and 4 have almost no meaning. In such a situation, however, the search is only two times slower than for the original SIFT. From this, we can say the PQTable remains robust for heavily biased element distributions. Note that the recall value is lower for the PCA-aligned case because PQ is less effective for biased data [13].

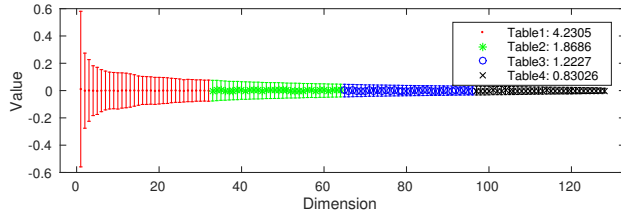
Comparative study: A comparison with existing systems for the SIFT1B data with 64-bit codes is shown in Table 4. We compared the proposed PQTable with two short-

Table 4: A performance comparison for different methods using the SIFT1B data with 64-bit codes. The bracketed values are from [4]. We show recall values involving similar computational times to those for PQTable.

System	#Cell	List len (L)	R@1	R@10	R@100	Time [ms]	Memory [Gb]	Required params	Required training step
PQTable	-	-	0.067	0.249	0.569	10.3	19.8	None	None
IVFADC [13]	2^{13}	8 million	0.104 (0.112)	0.379 (0.343)	0.758 (0.728)	111.6 (155)	(12)	#Cell, L	Coarse quantizer
OMulti-D-OADC-Local [4]	$2^{14} \times 2^{14}$	10000	(0.268)	(0.644)	(0.776)	(6)	(15)	table-order, k , L	Coarse quantizer, local codebook



(a) Original SIFT data. Runtime per query: 6.45 ms. Recall@1=0.25



(b) PCA-aligned SIFT data. Runtime per query: 12.94 ms. Recall@1=0.17

Figure 8: The average and standard deviation for the original SIFT vectors and the PCA-aligned vectors.

code-based inverted indexing systems, IVFADC [13] and OMulti-D-OADC-Local [4, 3, 15]. IVFADC is the simplest system, and can be regarded as the baseline, where the coarse quantizer uses a k -means method. OMulti-D-OADC-Local is a current state-of-the-art system, where the coarse quantizer involves PQ [1], the space for both the coarse quantizer and the short code is optimized [7], and the quantizers are per-cell learned [3, 15]. Note that the T for the PQTable is automatically set to two by using Eq. (5).

From the table, IVFADC has slightly better accuracy, but is usually slower than PQTable. IVFADC requires two parameters to be tuned, namely the number of space partitions #Cell and the length of the returned list L . The best-performing system, OMulti-D-OADC-Local, can produce superior results. It can achieve better accuracy and memory usage than PQTable. Regarding memory, as discussed in the previous subsection, the memory overhead of PQTable is not so small. As for computational cost, both are similar, but OMulti-D-OADC-Local is better with the best parameters. However, it has three parameters to be tuned, namely table-order, k , and L . All have to be tuned manually. In addition, several training steps are required, namely learning the coarse quantizer and constructing local codebooks, both of which are usually time-consuming.

From the comparative study, we can say there are advantages and disadvantages for both the inverted indexing

systems and our proposed PQTable. **Static vs. dynamic database:** For a large static database where users have enough time and computational resources for tuning parameters and training quantizers, the previous inverted indexing systems should be used. On the other hand, if the database changes dynamically, the distribution of vectors might vary over time and parameters might need to be updated often. In such cases, the proposed PQTable would be the best choice.

Ease of use: The inverted indexing systems produce good results but are difficult to handle by novices because they require several tuning and training steps. The proposed PQTable is conceptually simple and much easier to use, because users do not need to decide on any parameters. This would be useful if users were using an ANN method simply as a tool for solving problems in some other domain, such as fast SIFT matching using ANN for 3D reconstruction [6].

Limitations: The PQTable is no longer efficient for ≥ 128 -bit codes with the SIFT1B data ($T = 4$, $N = 10^9$), taking 3.4 s per query, which is still ten times faster than for the ADC, but slower than for the state-of-the-art system [4]. This is caused by the inefficiency of the “checking” process for longer bit codes. Handling these longer codes is proposed for future work. Note that ≤ 128 -bit codes are practical for many applications, such as the use of 80-bit codes for state-of-the-art image-retrieval systems [19].

Another limitation is the heavy memory usage of PQTable. Constructing a memory efficient data structure for Hash tables remains a future work.

6. Conclusion

We propose the PQTable, a nonexhaustive search method for finding the nearest PQ codes without need of parameter tuning. The PQTable is based on a multiindex hash table, and includes candidate code generation and merging of multiple tables. From an empirical analysis, we showed that the required parameter value T can be estimated in advance. An experimental evaluation using the SIFT1B data showed that, the PQTable could compute results 10^2 to 10^5 times faster than the ADC-scan method. The disadvantage of the PQTable is its heavier memory overhead compared to the state-of-the-art systems.

Acknowledgements: This work was supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) and JSPS KAKENHI Grant Number 257696 and 15K12025.

References

- [1] A. Babenko and V. Lempitsky. The inverted multi-index. In *Proc. CVPR*. IEEE, 2012. 1, 3, 4, 8
- [2] A. Babenko and V. Lempitsky. Additive quantization for extreme vector compression. In *Proc. CVPR*. IEEE, 2014. 1, 2
- [3] A. Babenko and V. Lempitsky. Improving bilayer product quantization for billion-scale approximate nearest neighbors in high dimensions. *CoRR*, abs/1404.1831, 2014. 1, 8
- [4] A. Babenko and V. Lempitsky. The inverted multi-index. *IEEE TPAMI*, 2014. 1, 8
- [5] A. Babenko and V. Lempitsky. Tree quantization for large-scale similarity search and classification. In *Proc. CVPR*. IEEE, 2015. 1, 2
- [6] J. Cheng, C. Leng, J. Wu, H. Cui, and H. Lu. Fast and accurate image matching with cascade hashing for 3d reconstruction. In *Proc. CVPR*. IEEE, 2014. 8
- [7] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization. *IEEE TPAMI*, 36(4):744–755, 2014. 1, 8
- [8] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE TPAMI*, 35(12):2916–2929, 2013. 2
- [9] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. In *Proc. FOCS*. IEEE, 1994. 2, 5, 6
- [10] K. He, F. Wen, and J. Sun. K-means hashing: an affinity-preserving quantization method for learning binary compact codes. In *Proc. CVPR*. IEEE, 2013. 2
- [11] J.-P. Heo, Z. Lin, and S.-E. Yoon. Distance encoded product quantization. In *Proc. CVPR*. IEEE, 2014. 1
- [12] M. Iwamura, T. Sato, and K. Kise. What is the most efficient way to select nearest neighbor candidates for fast approximate nearest neighbor search? In *Proc. ICCV*. IEEE, 2013. 1
- [13] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE TPAMI*, 33(1):117–128, 2011. 1, 2, 7, 8
- [14] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proc. ICASSP*. IEEE, 2011. 1, 6
- [15] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proc. CVPR*. IEEE, 2014. 1, 8
- [16] S. P. Lloyd. Least squares quantization in pcm. *IEEE TIT*, 28(2):129–137, 1982. 3
- [17] M. Norouzi and D. J. Fleet. Cartesian k-means. In *Proc. CVPR*. IEEE, 2013. 1
- [18] M. Norouzi, A. Punjani, and D. J. Fleet. Fast exact search in hamming space with multi-index hashing. *IEEE TPAMI*, 36(6):1107–1119, 2014. 2, 5, 6, 7
- [19] E. Spyromitros-Xioufis, S. Papadopoulos, I. Y. Kompatsiaris, G. Tsoumakas, and I. Vlahavas. A comprehensive study over vlad and product quantization in large-scale image retrieval. *IEEE TMM*, 16(6):1713–1728, 2014. 8
- [20] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014. 2
- [21] J. Wang, J. Wang, J. Song, X.-S. Xu, H. T. Shen, and S. Li. Optimized cartesian k-means. *IEEE TKDE*, 2014. 1
- [22] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Proc. NIPS*, 2008. 2
- [23] Y. Xia, K. He, F. Wen, and J. Sun. Joint inverted indexing. In *Proc. ICCV*. IEEE, 2013. 1
- [24] T. Zhang, C. Du, and J. Wang. Composite quantization for approximate nearest neighbor search. In *Proc. ICML*, 2014. 1, 2
- [25] T. Zhang, G.-J. Qi, J. Tang, and J. Wang. Sparse composite quantization. In *Proc. CVPR*. IEEE, 2015. 1, 2