Supplementary Material for "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

1. Generate candidate scheme using multisequence algorithm

We review the multisequence algorithm [1] using our efficient implementation in App. Alg. 1. It consists of two steps: initialization (Init) and code generation (NextCode).

Initialization: For the initialization, given a query vector $\boldsymbol{x}$ and codewords $\mathcal{C}$, $\boldsymbol{x}$ is divided into subvectors $\boldsymbol{x}^m$, and coded by corresponding subcodewords $\mathcal{C}^m$. When coding, the distances from the $\boldsymbol{x}^m$ to each subcodeword $\boldsymbol{c}_k^m \in \mathcal{C}^m$ are recorded in $d\_table[m]$, and sorted. Note d_table is an $M \times K$ 2D array consisting of tuples. Each tuple is composed the ID[1] of the centroids and a distance to the query.

Then, the indices of the closest subcodewords are collected and used to construct the nearest code, which is then kept in $cand$. Note $cand$ is a priority queue composed of tuples. Each tuple consists of a code and its distance to the query, sorted by the distance, where the distance is automatically computed by looking up d_table when pushing[2]. $cand$ holds candidates of the final output. The computational cost of the initialization is $O(K(D + \log K))$ for encoding and sorting, which is negligible for a large database.

Code generation: To generate a code, the nearest code to the query is popped from the priority queue, which consists of candidates of the nearest code.

For generation of the next-nearest candidates, new candidates of codes are computed as follows. Suppose the popped $code$ is constructed by IDs, from $(d\_table[1][k_1], \ldots, d\_table[M][k_M])$. We construct M new_codes, where each of them is constructed by IDs from $(d\_table[1][k_1], \ldots, d\_table[m][k_m + 1], \ldots, d\_table[M][k_M])$, for $m = 1 \ldots M$. These M new_codes are new candidates, and kept in the priority queue, $cand$. The NextCode( ) function is called whenever a new candidate is to be generated.

Appendix algorithm 1: Multisequence algorithm [1] with our efficient implementation.

Function Init
  Input: $\boldsymbol{x} = [\boldsymbol{x}^1, \ldots, \boldsymbol{x}^M]$ // query vector
    $\mathcal{C} = \mathcal{C}^1 \times \cdots \times \mathcal{C}^M$ // codewords
  Output: $cand$ // priority queue
    $d\_table[\,][\,]$ // 2D array
  $cand \leftarrow \emptyset$
  $d\_table[\,][\,] \leftarrow \emptyset$
  for $m \leftarrow 1$ to M **do**
    for $k \leftarrow 1$ to K **do**
      $\boldsymbol{c}_k^m \leftarrow k$-th center from $\mathcal{C}^m$
      $d\_table[m][k] \leftarrow \text{tuple}(k, d(\boldsymbol{x}^m, \boldsymbol{c}_k^m)^2)$
    SortByDist($d\_table[m][\,]$)
  $code \leftarrow$ Collect IDs from $d\_table[m][1]$
    for all m and construct code.
  $cand$.Push($code$)
Function NextCode
  Input: $cand$, $d\_table[\,][\,]$
  Output: $cand, code$
  $code \leftarrow cand$.Pop( )
  Suppose $code$ is constructed by collecting
    IDs from $d\_table[m][k_m]$ for all m.
  for $m' \leftarrow 1$ to M **do**
    $next\_code \leftarrow$ Collect IDs from
      $d\_table[m][k_m + a]$ for all m and construct
      code where $a \leftarrow 1$ if $m = m'$, else 0.
    $cand$.Push($next\_code$)

[1] To avoid confusion, we use "ID" to represent the ID-th centroids of subquantizers and "identifier" to denote database vectors.

[2] Note that this priority queue does not contain duplicate codes as a result of checking the code when pushing a new tuple into the priority queue, using another supplemental hash table. This "checking" step is a different implementation to the original method [1], where they required an M-dimensional array to handle this, requiring much more memory. Note that the results of the two algorithms are exactly the same.

2. Proof that required identifiers are included in the marked identifiers

We prove that all identifiers where their asymmetric distance (d_{AD}) is less than $d_{AD}(\boldsymbol{x}, \boldsymbol{y}_{u^*})$ are included in the marked identifiers.

Hereafter, we denote the ADC from the t-th sequence as $d^t_{AD}(\boldsymbol{x}, \boldsymbol{y})$, which leads to

$$\sum_{t=1}^{T} d^t_{AD}(\boldsymbol{x}, \boldsymbol{y})^2 = d_{AD}(\boldsymbol{x}, \boldsymbol{y})^2. \quad (1)$$

Here, we introduce a proposition.

Proposition: Given a query vector y, suppose there are sequences of identifiers from each table, where each sequence is sorted in ascending order by its distance to the query (d^t_{AD}). If we find the same identifier u^* from all of the sequences, then any vectors $\boldsymbol{y}_u$ in the database where $d_{AD}(\boldsymbol{x}, \boldsymbol{y}_u) < d_{AD}(\boldsymbol{x}, \boldsymbol{y}_{u^*})$ have been already included in the marked identifiers.

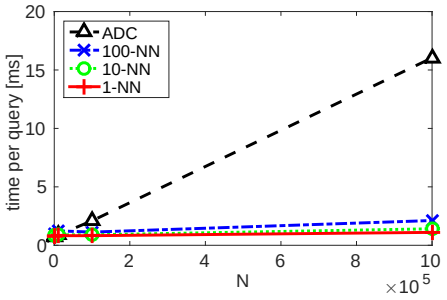
Proof: The proposition is proved by contradiction. Suppose there is an identifier $\bar{u}$, where $d_{AD}(\boldsymbol{x}, \boldsymbol{y}_{\bar{u}}) < d_{AD}(\boldsymbol{x}, \boldsymbol{y}_{u^*})$, and it is not included in the marked identifiers. Because $\bar{u}$ is not included in the marked identifiers, $\bar{u}$ must appear after u^* in each sequence. This leads to $d^t_{AD}(\boldsymbol{x}, \boldsymbol{y}_{u^*}) < d^t_{AD}(\boldsymbol{x}, \boldsymbol{y}_{\bar{u}})$ for all t, because each sequence is sorted in ascending order. By summing up all t, it leads to $d_{AD}(\boldsymbol{x}, \boldsymbol{y}_{u^*}) < d_{AD}(\boldsymbol{x}, \boldsymbol{y}_{\bar{u}})$, which contradicts the premise.

3. Experimental Results using GIST1M data

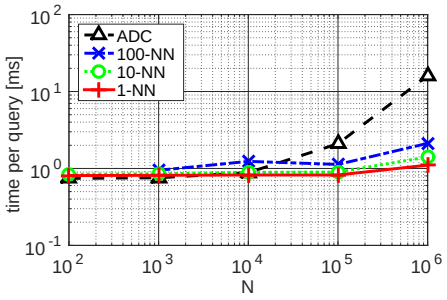
We show the performance evaluation using GIST1M data from the BIGANN dataset [2]. The GIST1M data contain one million 960-D GIST features in the database and provide vectors for learning and querying, from which we used 500,000 for learning and 1,000 for querying.

Fig. 1 shows the runtimes per query for the proposed method and the ADC for the GIST1M dataset. We observed the same tendencies as for the SIFT1B case. Because the size of the GIST1M database vectors was only one million, and our method is most efficient for much larger N, the speedup factors were not as high as for the SIFT1B dataset, but they were still always faster than the ADC for all parameter settings with $N = 10^6$.

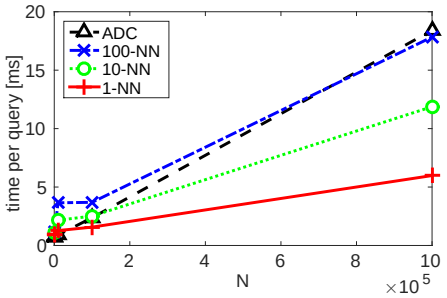
Note that accuracy of the proposed method is the same as for ADC: Recall@1=0.031, Recall@10=0.065, and Recall@100=0.173, for 32-bit codes, and Recall@1=0.056, Recall@10=0.125, and Recall@100=0.338, for 64-bit codes.



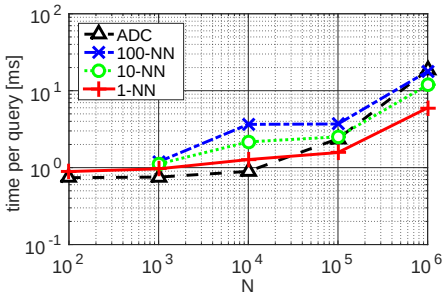
(a) Plot of 32-bit PQ codes.



(b) Log–log plot of 32-bit PQ codes.



(c) Plot of 64-bit PQ codes.



(d) Log–log plot of 64-bit PQ codes.

Figure 1: Runtimes per query for the proposed PQTable with 1-, 10-, and 100-NN, and a linear ADC scan (GIST1M dataset).

References

[1] A. Babenko and V. Lempitsky. The inverted multi-index. In *Proc. CVPR*. IEEE, 2012. 1

[2] H. Jègou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with souce coding. In *Proc. ICASSP*. IEEE, 2011. 2