Efficient Vector Data Search Using Sorting Transformation with Lookup Tables

Hongzhi Wang [1] **Tanveer Syeda-Mahmood** [1]

Abstract

For vector data, a sorting transformation sorts the elements of a vector by permuting the locations of its elements. It can be shown that among all permutation transformations, the sorting transformation minimizes L2 distance and maximizes similarity measures such as cosine similarity and Pearson correlation for vector data. Applying sorting transformation with vector/product quantization can substantially reduce compression errors when the same codebook size is applied, with a mild storage overhead for saving the sorting permutations for each compressed vector. For nearest neighbor search, the sorting transformation may produce false positive nearest neighbors when directly applied with the product quantization search algorithm. This problem was initially addressed via a re-ranking approach. In this work, we gave more in depth analysis to show that the re-ranking approach may be inadequate, especially for evenly distributed data. To address this problem, we adapted the lookup table approach for nearest neighbor search using sorting transformation based product quantization and showed its effectiveness on both real and simulated data.

1. Introduction

A sorting transformation is a permutation transformation that permutes a vector such that the elements of the permuted vector are in sorted order. Among all permutation transformations, the sorting transformation minimizes L2 distance and maximizes similarity measures such as cosine similarity and Pearson correlation for vector data (Wang & Syeda-Mahmood, 2024).

Vector quantization is a commonly used compression method based on nearest neighbor representation (Gray, 1984; Arya & Mount, 1993; Li & Salari, 1995). This technique divides a vector space into clusters, e.g. via K-means clustering, and the centroid of each cluster is used to represent every sample within that cluster. Each centroid is called a codeword and the collection of codewords is called a codebook. Given a sample to be compressed, its nearest neighbor codeword is chosen to represent the sample. The compression error produced by vector quantization can be quantified as the distance between the original sample and its nearest neighbor codeword.

Since sorted vectors have smaller distances comparing to their unsorted counterparts, sorting transformation based vector quantization can achieve comparable compression performance as standard vector quantization, but with substantially smaller codebooks. As a result, it substantially improves compression and decompression speeds comparing to standard vector quantization. The speed improvement comes at a cost of mild storage overhead for storing sorting permutations for compressed data, which as shown in (Wang & Syeda-Mahmood, 2024) is an overall beneficial tradeoff.

A key advantage of vector quantization is its high efficiency in approximate nearest neighbor search, e.g. through the product quantization method (Jegou et al., 2010; Ge et al., 2013), which has been widely used for retrieval-based applications (Lewis et al., 2020; Radford et al., 2021). When directly applying sorting transformation with product quantization for nearest neighbor search, the sorting transformation produces false positive nearest neighbors as nearest neighbor vectors obtained after sorting transformation may not be actual nearest neighbors in their original form. To address this problem, a re-ranking method was applied such that the retrieved nearest neighbors are inversely transformed back to the unsorted format and compared with original query vector to remove false positive nearest neighbors (Wang & Syeda-Mahmood, 2024). In this work, we show the weakness of the re-ranking approach and propose to adapt the lookup table method for nearest neighbor search using sorting transformation based product quantization.

Note that previous works on vector quantization have made progress on optimizing codebook generation (Qian, 2006; Ge et al., 2013; Kalantidis & Avrithis, 2014; Karri & Jena, 2016) and reducing the searching cost (Arya & Mount, 1993; Li & Salari, 1995). The sorting transformation approach

[1]IBM Almaden Research Center. Correspondence to: Hongzhi Wang <hongzhiw@us.ibm.com>.

Proceedings of the 1st Workshop on Vector Databases at International Conference on Machine Learning, 2025.

complements these methods and can be applied jointly with them.

2. Method

To be self-contained, we first describe sorting transformation and its application to product quantization for compression and search. In section 2.3, we describe the weakness in re-ranking based search and adapt the lookup table method for nearest neighbor search using sorting transformation based product quantization.

2.1. Reducing Vector Distance by Sorting Transformation

Let $X_i = \{x_i^1, ..., x_i^d\}$ be a vector of size d. The L2 norm between two vectors is:

$$|X_i - X_j|_2 = \sum_{k=1}^{d} \left(x_i^k - x_j^k\right)^2 \quad (1)$$

To reduce the distance between two vectors, we consider applying a permutation transformation to each vector. Let $(x_i^{\pi_i(1)}, ..., x_i^{\pi_i(d)})$ and $(x_j^{\pi_j(1)}, ..., x_j^{\pi_j(d)})$ be a permuted version of X_i and X_j, respectively. Under this consideration, (1) is a special case with identity permutation. Our goal is to find the permutation transformations that minimize the distance, i.e.

$$\pi_i^*, \pi_j^* = \text{argmin}_{\pi_i,\pi_j} \sum_m \left[x_i^{\pi_i(m)} - x_j^{\pi_j(m)}\right]^2 \quad (2)$$

Note that the distance can be minimized by permuting one vector and keeping the other vector unchanged. However, the optimal permutation will be based on the unchanged vector. For more general and simpler discussion, we consider applying permutations on both vectors.

It can be proven that the sorting permutations minimizes the above L2 distance (Wang & Syeda-Mahmood, 2024). A sorting permutation for a vector sorts the members of the vector. Without loss of generality, we only consider ascending sorting as the sorting permutation. Let π be the ascending sorting permutation for X. Then $X^{\pi(k)} \leq X^{\pi(m)}$ for $k < m$.

Sorting transformation reduces vector distance by projecting vectors from the space of unsorted vectors into a smaller space of sorted vectors. For a d-dimensional vector with no duplicated values, a sorted vector can be produced from sorting $d!$ unique vectors. Hence, the volume of sorted vectors is $\frac{1}{d!}$ of the volume of unsorted vectors. Fig. 1 illustrates the space of sorted vectors for 2D and 3D data.

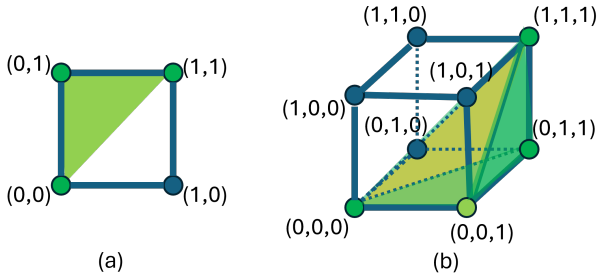

Figure 1. Illustrations of the space of sorted vectors in 2D (a) and 3D (b). For variables in $[0, 1]$, regular vectors are distributed in a square and a cube for 2D and 3D, respectively. In contrast, sorted vectors are distributed in the shaded triangle and pyramid for 2D and 3D, respectively.

Algorithm 1 SortPQ - Codebook Generation

Input: training data $\{x_i\}_{i=1}^N$, segment size d_m, number of segments M, codebook encoding bit n_b
for $i = 1$ **to** N **do**
  for $m = 1$ **to** M **do**
    Sort m_{th} segment of x_i.
  end for
end for
for each segment m=1 to M **do**
  Apply K-means to produce 2^{n_b} codewords using all sorted training data. Let C_m be the resulting codebook.
end for
Output: $\{C_m\}_{m=1}^M$

2.2. Sorting Transformation based Product Quantization

Since sorting transformation can greatly reduce vector distance, applying it to vector quantization can improve the compression accuracy. Here we apply it to product quantization. Product quantization is a variant of vector quantization that applies Cartesian product to generate codewords efficiently for high dimensional vectors. In an algorithm view, it divides high dimensional vectors into multiple low-dimensional segments and applies vector quantization to compress each vector segment independently (Jegou et al., 2010).

Algorithm 1, 2, 3 show how codebook generation, encoder and decoder work when applying sorting transformation with product quantization (SortPQ). The key difference from the standard product quantization (PQ) algorithm is that both training and testing data need to be sorted within each vector segment prior to codebook generation or encoding. Furthermore, the sorting permutations are stored as part of compression encoding for data compression.

Algorithm 2 SortPQ - Encoder

Input: vector x and codebooks $\{C_m\}_{m=1}^{M}$.
for each segment m=1 to M **do**
  Step 1 Sort m_{th} segment of x, let $p_m(x)$ be the sorting permutation.
  Step 2 Find the closest codeword $c_m(x)$ for the sorted segment of x from C_m.
end for
Output: $\{c_m(x), p_m(x)\}_{m=1}^{M}$

Algorithm 3 SortPQ - Decoder

Input: $\{c_m(x), p_m(x)\}_{m=1}^{M}$
for each segment m=1 to M **do**
  Inversely permute codeword $c_m(x)$ using $p_m(x)$. Let the result be $\hat{x}_m$.
end for
Output: $\hat{x} = \{\hat{x}_m\}_{m=1}^{M}$

The encoding cost for the sorting permutation of a vector segment of size d_m is bounded by $\lceil \log_2(d_m!) \rceil \sim d_m \log_2(d_m)$, which is an extra storage cost over standard PQ. Hence, applying SortPQ works best for low-dimensional segment sizes. As shown in (Wang & Syeda-Mahmood, 2024), the overhead for storing sorting permutations is well compensated by performance improvement for compression.

2.3. Nearest Neighbor Search with SortPQ

Algorithm 4 shows how data should be ingested for nearest neighbor search using SortPQ. Again, the key difference from standard PQ ingestion is that all ingested data are sorted within each segment and the sorting permutations are stored.

Re-ranking Search When both query and searched vectors are sorting transformed, the nearest neighbors in the sorted format may not be the nearest neighbors in the original unsorted format. Hence, applying sorting transformation may introduce false positive nearest neighbors. To address this problem, a re-ranking approach was applied in (Wang & Syeda-Mahmood, 2024), as shown in Algorithm 5. First, an initial set of nearest neighbors are retrieved for a sorting transformed query vector. Then, the decoded vectors of the initially retrieved nearest neighbors are used to re-calculate their actual distances to the original unsorted query vector, from which the top nearest neighbors are returned.

Algorithm 4 SortPQ - Ingestion

Input: vector set $\{x_i\}_{i=1}^{N}$ and codebooks $\{C_m\}_{m=1}^{M}$.
for $i = 1$ **to** N **do**
  Encode x_i using algorithm 2 and obtain encoding $\{c_m(x_i), p_m(x_i)\}_{m=1}^{M}$.
end for
Output: $\{\{c_m(x_i), p_m(x_i)\}_{m=1}^{M}\}_{i=1}^{N}$

Algorithm 5 SortPQ - Re-ranking Search

Input: vector x, codebooks $\{C_m\}_{m=1}^{M}$ and ingested data $\{\{c_m(x_i), p_m(x_i)\}_{m=1}^{M}\}_{i=1}^{N}$, k and l.
Step 1 Sort each segment of x and let the result be $s(x, m)$.
Step 2 Retrieve the top l nearest neighbors $\{ns(x)_1, ..., ns(x)_l\}$ for $s(x, m)$ from the ingested data using standard PQ search.
Step 3
for $i = 1$ **to** l **do**
  Decode $ns(x)_i$ using algorithm 3 and recalculate its distance to x using the decoded vector.
end for
Output: Top k nearest neighbors obtained in Step 3.

Algorithm 6 SortPQ - Lookup Table Search

Input: vector x, codebooks $\{C_m\}_{m=1}^{M}$ and ingested data $\{\{c_m(x_i), p_m(x_i)\}_{m=1}^{M}\}_{i=1}^{N}$, k.
for $m = 1$ **to** M **do**
  Create a table of size $2^{n_b} \times d_m!$ to store the distance between m_{th} segment of x and each permutation of every codeword for segment m.
end for
for $i = 1$ **to** N **do**
  Calculate the distance between x and x_i by summing up distances retrieved from corresponding lookup tables using $\{c_m(x_i), p_m(x_i)\}_{m=1}^{M}$.
end for
Output: Top k nearest neighbors based on the calculated distances.

Drawback of the re-ranking approach The re-ranking approach worked well for top-1 nearest neighbor search for SIFT and GIST datasets (Wang & Syeda-Mahmood, 2024). However, as shown below it is not an efficient solution for data that are evenly distributed in the vector space.

For a d-dimensional vector with 1 segment, let V be the entire vector space. Recall that the space of sorted vectors is $\frac{1}{d!}$ of the entire vector space. In other words, the entire vector space can be evenly divided into $d!$ equal volume subspaces, where each of the $d!$ subspaces corresponds to a unique permutation for a d-dimensional vector (see Fig. 1 for examples with $d = 2, 3$). Let $\{V_1, ..., V_{d!}\}$ be the $d!$ subspaces and V_1 be the sorted vector subspace for a query vector. The sorting transformation establishes a one-to-one map between each of the $d!$ subspaces and V_1, through which a one-to-one map is formed for each pair of subspaces

as well. Let q and V_q be a query vector and the original subspace it belongs to, respectively. Let $a(q,k) \in V_q$ be q's k_{th} nearest neighbor[1]. Let $b(q,k)$ be the hyper-ball centered at q with radius equal to the distance between q and its k_{th} nearest neighbor. The sorting transformation maps $d!$ hyper-balls from all subspaces to $b(q,k)$. Since k neighbors fall into $b(q,k)$, if vectors are evenly distributed across the entire vector space, the expected number of vectors falling in each hyper-ball that matches $b(q,k)$ is k as well. Since all those vectors are transformed into $b(q,k)$ through sorting transformation, to retrieve the top k nearest neighbor for a query vector, the initially retrieved list for re-ranking is expected to include $k(d!-1)$ false nearest neighbors.

For a vector of size d containing m equal size segments, i.e. segment size $d_m = \frac{d}{m}$, since sorting is performed within each segment, the vector space is divided into $(\frac{d}{m}!)^m$ equal volume subspaces and each pair of such subspaces form a one-to-one map through the sorting transformation. To retrieve the top k nearest neighbors for a query vector from the sorted vector space, the initially retrieved list for re-ranking is expected to include $k((\frac{d}{m}!)^m - 1)$ false nearest neighbors, which grows exponentially with respect to segment size and the number of segments. Due to the large number of false positive nearest neighbors produced by sorting transformations, the re-ranking approach is inefficient for evenly distributed datasets.

Lookup Table Search To address the false positive nearest neighbor problem caused by sorting transformation, we adapt the lookup table approach. In the standard PQ search, one lookup table is created for each segment to store the distance between a query vector and each codeword in that segment. Let $n_c = 2^{n_b}$ be the size of each codebook. Then all lookup tables contain mn_c precomputed values. Since each ingested vector in the searched dataset is represented by the nearest codeword in each segment, its distance to the query vector can be calculated by summing up the distances between its codewords to the query vector using the lookup tables. The lookup table approach is efficient and has been adapted by variants of vector quantization methods such as (Jegou et al., 2010; Babenko & Lempitsky, 2014) for nearest neighbor search.

In SortPQ, each vector in the ingested dataset is represented by the nearest codeword, which is sorted, and its sorting permutation in each segment (see Algorithm 2). A vector can be represented by inversely applying its sorting permutations to its nearest codewords (see Algorithm 3). Hence, to efficiently calculate the distance between a query vector and all ingested sorting transformed vectors, a lookup table can be created for each segment by storing the distance between the query vector and all permuted versions of each codeword (as shown in Fig. 2). The lookup table for each segment contains $d_m!n_c$ values and all lookup tables contain $md_m!n_c$ values. The lookup table search algorithm is summarized in Algorithm 6.

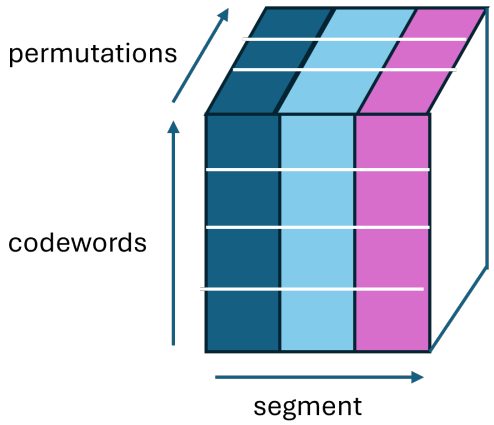

Figure 2. Illustration of the lookup tables for vector distance computation using SortPQ encoding.

Computational Complexity PQ search has three components: 1) creating lookup tables; 2) calculating distances to each vector in the ingested dataset; 3) nearest neighbors retrieval based on the calculated distances. For SortPQ, the computational complexity for the three components are $O(md_m!n_c)$, $O(mN)$, and $O(N + klogN)$, respectively [2]. N is the number of ingested vectors and k is the number of retrieved nearest neighbors. The standard PQ has the same computational cost for the last two components, but has lower complexity of $O(mn_c)$ for lookup table creation. In practice, usually small segment size d_m and large ingested dataset size N are employed, which makes the additional computational cost by SortPQ in creating the lookup tables a small fraction of the overall computational cost.

Furthermore, the computational cost for lookup table creation for SortPQ can be reduced by using another lookup table process. For $L2$ distance (1), calculating the distance between two vectors of size d_m needs $2d_m - 1$ addition/subtraction operations and d_m multiplications. The total number of variables stored in lookup tables for SortPQ is $md_m!n_c$. Directly computing such lookup tables using (1) needs $d_m md_m!n_c$ multiplications and $(2d_m - 1)md_m!n_c$ addition/subtraction operations. However, for each code-

[1]Without loss of generality, we assume that top nearest neighbors live in the same subspace as the query vector for simple discussion.

[2]The complexity for nearest neighbor retrieval is based on heap sort. The complexity for building a heap is $O(N)$. The complexity for retrieving the top k nearest neighbors and re-heapify the heap after each neighbor is retrieved is $O(klogN)$.

word produced by SortPQ, $d_m!$ vectors are derived from applying permutations to it for lookup table generation. There are substantial repeated computations in calculating the distances between the $d_m!$ derived codewords with the same query vector. To avoid such repeated computations, $t_{ij} = \left(w^i - q^j\right)^2$ can be precomputed and stored for reuse for $0 \leq i, j \leq d_m$. w is a codeword produced by SortPQ and q is the corresponding vector segment of a query vector. With this approach, The lookup tables can be created for SortPQ with $d_m^2 m n_c$ multiplications and $d_m^2 m n_c + (d_m - 1) m d_m! n_c$ addition/subtraction operations in total, which uses $\frac{1}{(d_m-1)!}$ multiplications and half addition/subtraction operations of the method without reusing repeated calculations. Hence, reusing the lookup table $\{t_{i,j}\}_{i,j=1}^{d_m}$, the computational cost can be reduced by more than 50%. Given the fact that the best multiplication algorithm (Harvey & Van Der Hoeven, 2021) has complexity of $O(blogb)$ for numbers with b-bit representation and has complexity of $O(b)$ for addition/subtraction, the actual reduction of computation can be greatly over 50%.

3. Experiments

3.1. Data

We tested our method using the SIFT and GIST descriptor data (Jegou et al., 2010). Both data sets were derived from natural images. Each SIFT sample has dimension of 128, while each GIST sample has dimension of 960. Both data sets have one million vectors for training. SIFT and GIST have 10000 and 1000 samples for query testing, respectively, In addition to the above two datasets derived from real data, we also generated a simulated random dataset with 1000000 vectors for training and 1000 vectors for testing. The random vectors have dimension of 384. The values of the vectors were randomly generated with value range in $[0, 1)$. Unlike the SIFT and GIST data, the random vectors are evenly distributed in the vector space.

3.2. False positive nearest neighbor analysis for SortPQ

The analysis in section 2.3 shows the expected number of false positive nearest neighbors produced by applying sorting transformations to evenly distributed data. In this section, we conducted experimental study to demonstrate such effects.

Given a testing vector and a training dataset from which the nearest neighbors are retrieved for the testing query vector, we define that the nearest neighbors retrieved for a query vector using the original uncompressed data are the ground truth nearest neighbors for the query data. To test the effect of sorting transformation at a specific segment size on nearest neighbor retrieval, sorting transformations were applied to both testing and training data such that the elements within each segment are sorted. For each testing vector, the nearest neighbor ranking position of its ground truth k_{th} nearest neighbor after applying sorting transformation indicates the number of false positive nearest neighbors that have lower ranks than the ground truth neighbor. Table 1 shows the average nearest neigbor ranking positions of the top 10 ground truth nearest neighbors for all testing data after applying sorting transformations. For this study, we tested with segment sizes 2 and 4, respectively.

Consistent with our earlier analysis, more false positive nearest neighbors were produced with the larger segment size 4 for all three datasets. For evenly distributed random data, the sorting transformation produced many false positive nearest neighbors. For segment size 2, on average there are over 1136 false positives for the top 1 nearest neighbor and there are over 3600 false positives for the 10_{th} nearest neighbor. For segment size 4, the number of false positives increased substantially. There were over 27000 false positives for top 1 nearest neighbor and over 46000 false positives for 10_{th} nearest neighbor. These results show that the re-ranking search approach will not work well for SortPQ nearest neighbor search on this evenly distributed dataset.

On the other hand, for real dada, i.e. SIFT and GIST, the sorting transformation only mildly produced false positive nearest neighbors. This result indicates that these two datasets are far away from being evenly distributed. It is consistent with the nearest neighbor search performance produced by re-ranking based sortPQ search reported in (Wang & Syeda-Mahmood, 2024).

3.3. Nearest neighbor retrieval

To test the performance of nearest neighbor retrieval, for each testing query data, its top nearest neighbors among the training data set were retrieved using the following three methods: 1) original values used for both query and ingested samples; 2)-3) original values used for query samples, while the ingested samples are compressed using product quantization (PQ) and sorting transformation based product quantization (SortPQ), respectively. For SortPQ, both the re-ranking based search (SortPQ-rerank) and the lookup table search (SortPQ-lookup) were tested. For SortPQ-rerank, $l = 128$ was applied for initially retrieved set for re-ranking. Our SortPQ method was implemented based on the PQ implementation (Matsui, 2023).

To obtain a complete performance profile, combinations of various segment sizes and codebook sizes were tested. Since large segment sizes usually lead to large compression errors, and poor nearest neighbor retrieval performance, we focused our test on small segment sizes $\{2, 4\}$. For codebook size, we tested $\{64, 128, 256, 512, 1024, 2048, 4096\}$, which correspond to $\{6, 7, 8, 9, 10, 11, 12\}$ bits codeword

Table 1. The average nearest neighbor rankings of top $k \in [1, 10]$ nearest neighbors after applying sorting transformations, which sort the data with each segment. The tested segment sizes are either 2 or 4, which is shown after the dataset name.

	TOP-1	TOP-2	TOP-3	TOP-4	TOP-5	TOP-6	TOP-7	TOP-8	TOP-9	TOP-10
SIFT-2	2.1	4.9	7.6	10.7	12.9	17.1	18.0	21.4	22.7	26.7
SIFT-4	38.4	75.2	110.6	117.0	134.4	186.0	185.8	239.7	260.4	234.8
GIST-2	7.2	12.6	18.1	22.8	33.9	37.7	37.9	43.2	47.3	51.5
GIST-4	164.5	182.5	255.9	303.2	450.5	454.7	491.9	451.2	514.7	516.4
RANDOM-2	1137.1	1626.2	2299.1	2404.2	2809.9	2762.1	3273.9	3509.9	3267.5	3625.1
RANDOM-4	27096.1	29960.3	39090.0	41336.5	40559.1	43569.3	45543.9	43413.5	43768.2	46877.7

encoding cost for each segment, respectively.

For evaluation, the retrieval results produced by PQ methods were respectively compared with the retrieval results produced using original data, under the assumption that better retrievals should have higher overlaps with retrievals produced on original data.

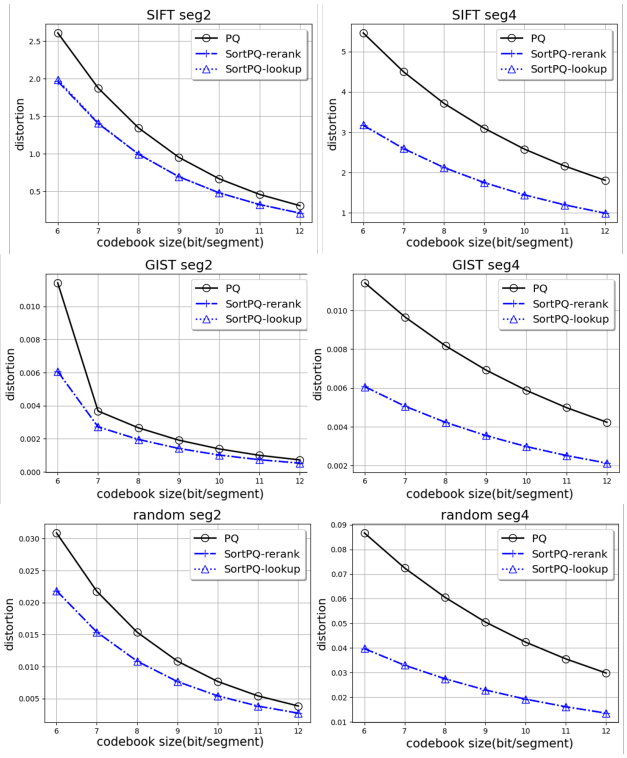

Figure 3. Distortion (mean absolute distance) produced by PQ and SortPQ. The x-axis shows the codebook size for each segment in bit/segment.

Fig. 3 summarizes the compression accuracy produced by each method. Since the two variants of the SortPQ methods share the same compression algorithm, they produced similar compression distortions, which are consistently better than those produced by PQ. Please refer to (Wang & Syeda-Mahmood, 2024) for in-depth discussion on the trade-off between computational and storage cost for SortPQ and PQ. In summary, the storage overhead of SortPQ for storing sorting permutations is well compensated by the reduction of computational cost.

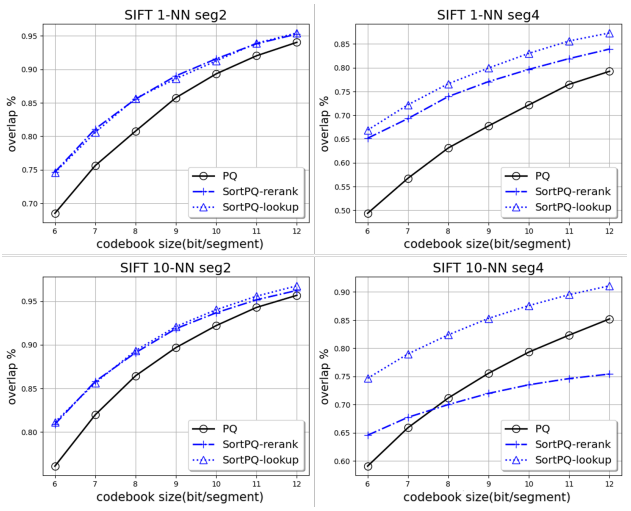

Figure 4. Nearest neighbors retrieval results for the SIFT dataset. Overlaps between retrieval on original data and retrieval using PQ and SortPQ, respectively.

Fig.4 shows the nearest neighbor retrieval results for the SIFT dataset. When segment size 2 was applied, both re-ranking search and lookup table search produced comparable results for both top-1 and top-10 nearest neighbor search results. This result is consistent with the results reported in Table 1 as only a small number of false positives were introduced by sorting transformation. Using $l = 128$ for re-ranking is adequate for removing false positives. When segment size 4 was applied, lookup table search showed clear advantage over re-ranking search, with more substantial advantage for top-10 neighbor search. SortPQ with re-ranking search updperformed PQ search when segment size 4 and at least 8-bit codebook encoding were used.

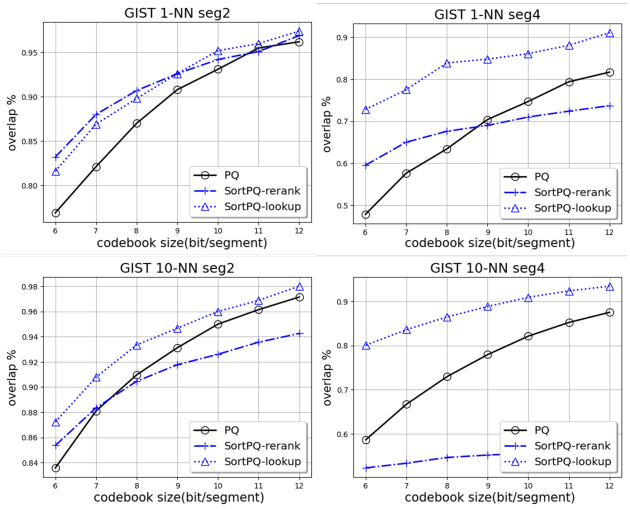

Figure 5. Nearest neighbors retrieval results for the GIST dataset. Overlaps between retrieval on original data and retrieval using PQ and SortPQ, respectively.

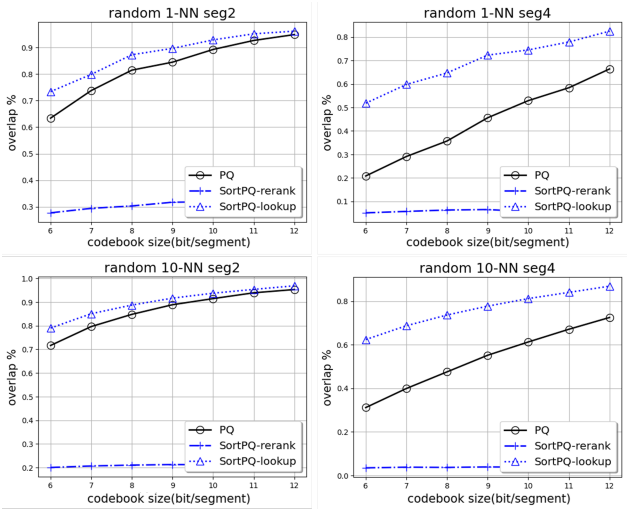

Figure 6. Nearest neighbors retrieval results for the random dataset. Overlaps between retrieval on original data and retrieval using PQ and SortPQ, respectively.

Fig.5 shows the nearest neighbor retrieval results for the GIST dataset. When segment size 2 was applied, both re-ranking search and lookup table search produced comparable results for top-1 neighbor search. However, lookup table search outperformed re-ranking search for top 10 neighbor search. Again, lookup table search showed clear advantage over re-ranking search when segment size 4 was applied.

Fig.6 shows the nearest neighbor retrieval results for the simulated random dataset. As expected, for evenly distributed, SortPQ with re-ranking search substantially underperformed PQ and SortPQ with lookup table search. For all three datasets, SortPQ with lookup table search consistently outperformed PQ search and SortPQ with re-ranking search.

4. Conclusions

We showed that the number of false positive neighbors produced by applying sorting transformation to vectors grows exponentially with respect to the segment size and number of segments for evenly distributed data. The re-ranking approach is not the most effective means to remove such false positive nearest neighbors. To address this problem, we adapted the lookup table approach that can completely avoid the false positive neighbor problem. Our experiments on both real data and simulated random data showed that the lookup table approach consistently improved over re-ranking based search. Our work further demonstrates the advantage of applying sorting transformation to vector data search.

Impact Statement

This paper presents work with the goal to advance the field of vector database search. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Arya, S. and Mount, D. M. Algorithms for fast vector quantization. In *[Proceedings] DCC93: Data Compression Conference*, pp. 381–390. IEEE, 1993.

Babenko, A. and Lempitsky, V. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 931–938, 2014.

Ge, T., He, K., Ke, Q., and Sun, J. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence*, 36(4):744–755, 2013.

Gray, R. Vector quantization. *IEEE Assp Magazine*, 1(2): 4–29, 1984.

Harvey, D. and Van Der Hoeven, J. Integer multiplication in time o(nlog\,n). *Annals of Mathematics*, 193(2):563–617, 2021.

Jegou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

Kalantidis, Y. and Avrithis, Y. Locally optimized product quantization for approximate nearest neighbor search. In

Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2321–2328, 2014.

Karri, C. and Jena, U. Fast vector quantization using a bat algorithm for image compression. *Engineering Science and Technology, an International Journal*, 19(2):769–781, 2016.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

Li, W. and Salari, E. A fast vector quantization encoding method for image compression. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):119–123, 1995.

Matsui, Y. Nano product quantization (nanopq). https://github.com/matsui528/nanopq, 2023.

Qian, S.-E. Fast vector quantization algorithms based on nearest partition set search. *IEEE transactions on image processing*, 15(8):2422–2430, 2006.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pp. 8748–8763. PMLR, 2021.

Wang, H. and Syeda-Mahmood, T. Vector quantization with sorting transformation. In *2024 IEEE International Conference on Big Data (BigData)*, pp. 2384–2389. IEEE, 2024.