Hashing-based Approximate Nearest-Neighbor Search for Large-scale Image Retrieval

Zhejun HE

zhejun.he@connect.polyu.hk

***Abstract* - With the extensive growth of datasets, content-based image retrieval (CBIR), which entails the calculation of the distance between image feature vectors, has been becoming an important and promising topic. Large amounts of vectorized data, extracted from images via machine learning algorithms, are usually of high dimension, and traditional linear scanning would not be efficient in terms of speed and resource utilization. Approximate Nearest-Neighbor Search (ANNs) is an effective solution for quickly retrieving the Top-k results that are closest to the query image. Therefore, we investigate two well-known hashing based ANNs methods in this study.**

Furthermore, we propose a faster hashing-based ANN algorithm, named FPQ. Our algorithm considers both the data distribution and the time-consuming nature in codebook construction, and it employs memory-efficient algorithms to achieve increased time efficiency in the training and indexing processes. The speed, accuracy, and memory consumption of FPQ are compared and evaluated with other algorithms. Our results indicate the superiority of FPQ in terms of speed while consuming minimal memory resources, and search accuracy is well retained. This research contributes to the development of efficient CBIR systems that can be deployed in real-world applications.

***Keywords*: Approximate Nearest-Neighbor Search, Hashing-Based Methods, Content-based Image Retrieval, High-Dimensional Data, Top-K Retrieval.**

I. PROBLEM DEFINITION

With the query image I_q, our objective is to efficiently retrieve top-k similar images close to I_q, from the image dataset $I = \{I_1, I_2, \dots, I_N\}$, where N is the size of the database usually of large-scale in practice. The image I_X can be converted into a vector in the following steps [1]:

(1) The image I_X is rasterized into an array of pixel elements, which can then be encoded into a matrix.

(2) Feature extraction, for example CNN, is adopted to extract numerical data from the matrix, which generally involves extracting low-level features such as color, texture, and shapes.

(3) The extracted features of an image are quantified into feature vector $X = [x_1, x_2, \dots, x_D]^T$.

This transformation could be applied to both query image and images in the dataset. Therefore, the image retrieval task in a large image collection can be converted into large-scale vector similarity search, with each feature vector linked to the corresponding image.

Then we consider a set of points D — each point represented by a vector — in d-dimensional space $\mathbb{R}^d$, and a query point q. We denote $\|p,\ q\|$ as the Euclidean distance between two points p and q in the space. We can obtain the top-k points through exhaustive search irrespective of the size of the set D and the dimension d.

Obtaining the exact nearest neighbor of q in the brute-force manner, however, in a high-dimensional Euclidean space of the system, which could contain billions of objects in practice, can be computationally expensive. This is known as the "Curse of dimensionality". For image retrieval in some large-scale datasets larger than $N = 10^6$, however, ANNs has been verified to achieve better trade-offs between accuracy and resources utilization than brute-force linear scan method [2].

In this study, we focus on optimizing hashing-based c-ANNs. c-ANNs is a popular version of ANNs problem. c-ANNs aims to obtain a point $o \in D$ that satisfies $\|o,\ q\| \leq c\|o^*,\ q\|$, where c is an approximation ratio that exceeds 1, and o^* is the exact nearest neighbor of q. This problem lays a foundation for top-k similar images retrieval.

II. EXISTING METHODS

In this session we review related work in the hashing based ANNS and their application for image retrieval. In general, the hashing-based ANNs methods project the point in high-dimensional space $\mathbb{R}^d$ to a low-dimensional embedded representation, which is a hash code technically. The hash code reduces the dimensionality of the data and enables efficient storage and distance calculation of the data points.

Numerous hashing-based ANN methods have been proposed in the previous decades, which can be divided into two sub-categories: locality sensitive hashing (LSH) and learning to Hash (L2H). [3] We examine these two methods in this session.

A. Locality-sensitive Hashing

Locality-sensitive hashing (LSH) is a hashing technique that relies on the concept that if two points in a space are in close proximity to each other, there is a high probability of them remaining close together even after being projected onto a

hyperplane. In essence, LSH works by bucketing similar points through hash functions. Then with the query points hashed, one can retrieve near neighbors as well as elements stored in buckets which contain that point. It enables much better efficiency than using the brute force approach which compares all vectors with each other.

The core of LSH is the scalar projection (dot product), given by $a^T x$, where x is a point in the space and a^T is a vector with components that are selected at random from a Gaussian distribution, for example $N(0, I)$. This scalar projection is then quantized with the hash function

$$h(x) = \left\lfloor \frac{a^T x + b}{w} \right\rfloor \quad (1)$$

where w is the width of each quantization bin, and b is random variable uniformly distributed in the interval $[0, w)$ that makes the quantization error easier to analyze.

Numerous methods have been developed based on LSH, which define hash in different fashions, for example MinHash [4]. The projection is usually obtained by concatenating multiple hash functions, and a quantization bucket is a set of points binned together based on their similarity. Even if the hashing details are different, the hash functions are designed with the goal that

1. For any points p and q in $\mathbb{R}^d$ that are close to each other ($\|p - q\| \leq R$), there is a high probability P_1 that they fall into the same bucket
$$P[h(p) = h(q)] \geq P_1$$
2. For any points p and q in $\mathbb{R}^d$ that are far apart ($\|p - q\| \geq cR, c > 1$), there is a low probability $P_2 \leq P_1$ that they fall into the same bucket
$$P[h(p) = h(q)] \leq P_2$$

The points in the dataset can be classified into various quantization buckets using hash functions. Given a query point q which will be then hashed, the algorithm iterates the data points that are hashed in the same bucket, and the desired point o is found when it satisfies $\|o, q\| \leq c\|o^*, q\|$ according to c-ANNs problem. Thus, this method allows for efficient retrieval of similar elements by reducing the comparison process, rather than performing a comparison of the full dataset.

It's noteworthy that LSH is data-independent, meaning data characteristics, such as data distribution, are not essential for designing hash functions. LSH fails to exploit the distribution of feature vectors. As a result, there is no guarantee that two adjacent will be mapped to adjacent hash spaces. Some scholars argue that this process of hashing unfortunately requires an enormous storage cost, if the size of the dataset becomes larger [5]. This original LSH index structure has excessively large indexes, which greatly impacted its efficiency and worked effectively only for datasets of relatively small or medium sizes [2]. However, some state-of-the-art LSH based methods like Collision Counting have alleviated this issue to some extent [6].

B. *Product Quantization*

Product Quantization (PQ) is an effective hashing technique in L2H for handling large-scale vector similarity search. In PQ algorithm, the product refers to the cartesian product, and the quantization refers to vector quantization. In PQ, the high-dimensional feature vector is compressed in such a manner that the PQ-encoded code can efficiently approximate the original vector, and the code can be stored in memory, thereby avoiding costly disk accesses. Without quantizing the query vector q like LSH, the distance between the q and the cluster center of each low-dimensional space can be calculated via "vector-to-centroid distances" [7].

Assume we have a feature vector $X = [x_1, x_2, \dots, x_D]^T$, where D is the dimension of the feature vector. In PQ, the vector is encoded into compact PQ-codes. Firstly, the vector X is split into multiple M sub-vectors, typically 8 or 16, each of which is significantly of lower dimension than the original vector, as demonstrated in Equation *(2)*.

$$\begin{aligned} X &= [x_1, x_2, \dots, x_D]^T \\ &= \left[\underbrace{x_1, x_2, \dots, x_{\frac{D}{M}}}_{x^{1T}}, \dots, \underbrace{x_{\frac{D-D}{M+1}}, \dots, x_D}_{x^{MT}} \right]^T \\ &= \left[x^{1T}, x^{2T}, \dots x^{MT} \right]^T \end{aligned} \quad (2)$$

where x^{th} is m^{th} sub-vector, and m $\in \{1, \dots, M\}$

Next step is vector quantization, and as seen in Figure 1, K-means algorithm is used to compress each sub-vector with quantizer q_c, which replaces each sub-region of a vector with the closest matching centroid. The centroids here refer to the most commonly occurring patterns in the dataset sub-vectors.

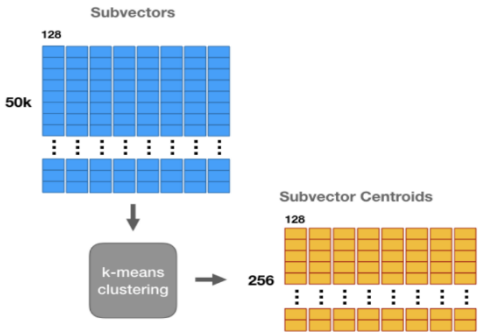


Figure 1: The sub-vectors are converted to the centroids [8].

To represent a vector of dimension D, we identify the closest centroid and its corresponding integer id for each sub-vector obtained in the first step. This generates a sequence of centroid ids, typically either 8 or 16, which significantly reduces space requirements. The mapping results of each sub-vector and centroid are stored in codebook C_i to enable

efficient retrieval. When reproduction value of the mapping is $I = I_1 \times \ldots \times I_m$, the codebook can be defined as

$$C = C_1 \times \ldots \times C_m \quad (3)$$

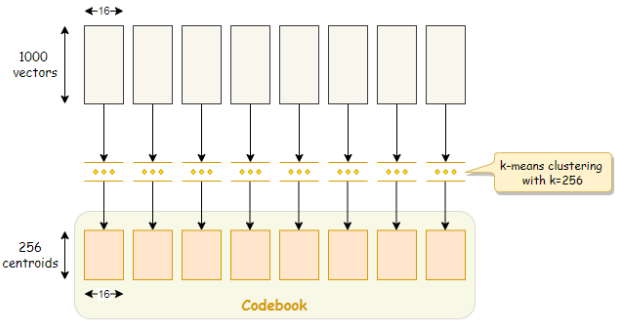
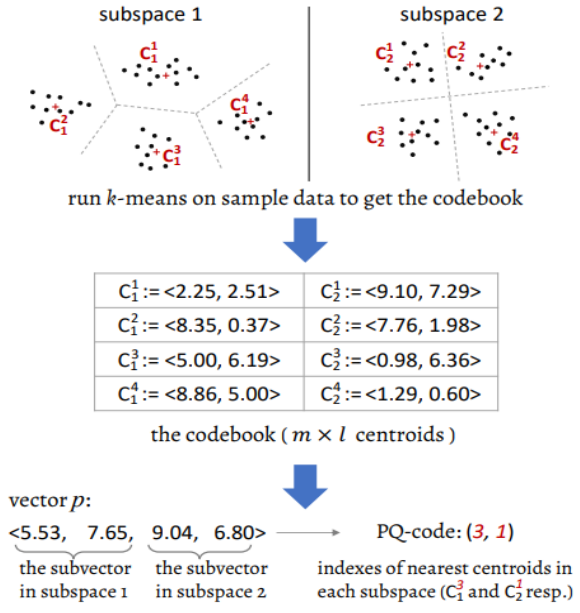


Figure 2: Each sub-vector converted into a centroid [7].

The figure 2 shows the process of obtaining the centroids via distance computation. Every centroid of this set is the concatenation of centroids of m subquantizers. However, In order to overcome the memory inefficiency associated with storing centroids, it is necessary to compress them into a compact code. To achieve this, we select the centroid closest to our target from a pool of centroids that belong to the same sub-space. Finally, we encode each segment of our original vectors through the use of centroid IDs, with each ID being encoded using 8 bits, known as PQ code.

We assume that the size of the original vector is 128×32 bits = 4096 bits, which is 512 bytes. We divide the vector into 8 sub-vector and each of them is converted into a PQ code as described earlier. As a result, this enables us to represent the 128-dimensional vector of floating-point numbers using the index of its closest center, which requires only 8 bytes of storage. Thus, this significantly improves memory usage. Figure 3 provides an example of this process.



C_1^1 := <2.25, 2.51>	C_2^1 := <9.10, 7.29>
C_1^2 := <8.35, 0.37>	C_2^2 := <7.76, 1.98>
C_1^3 := <5.00, 6.19>	C_2^3 := <0.98, 6.36>
C_1^4 := <8.86, 5.00>	C_2^4 := <1.29, 0.60>

Figure 3: The process of codebook construction and indexing [7].

Apart from traditional PQ, the inverted list is introduced to efficiently by [8] to store the mapping relationship, which is known as IVFPQ.

In the inverted index variant of Product Quantization (IVPFQ), coarse quantization is performed initially using vector quantization. Then, the dataset vectors are sequentially inserted into the inverted index table based on the quantization encoding. The inverted order is determined based on the distance between vectors and quantized feature values. This process improves the speed of search operations. However, we still observe that the standard PQ and IVPFQ algorithm uniformly groups vectors without considering their distribution. It should be acknowledged that some vectors may have a correlation between dimensions that impact their grouping. However, the computational cost of retrieval is still large when the size of dataset N is large, because of the low recall rate in both PQ and IVFPQ indexes [2].

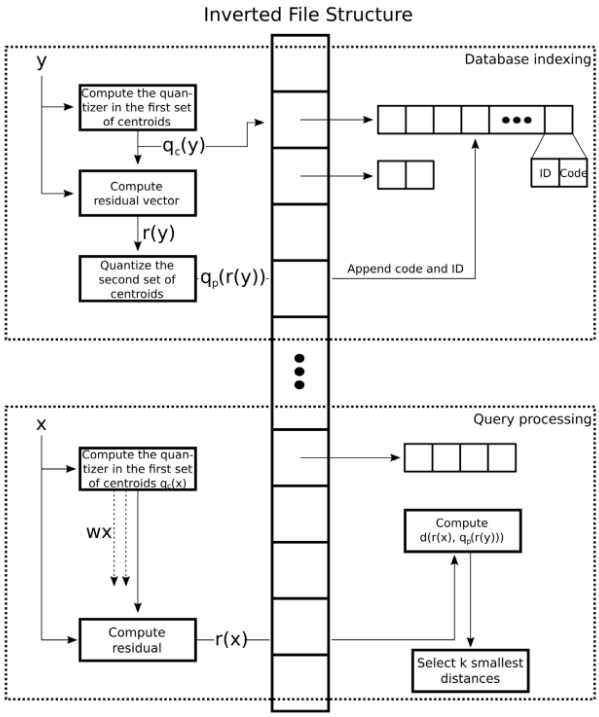


Figure 4: The process of inverted index construction [6].

I. MY PROPOSED METHODS

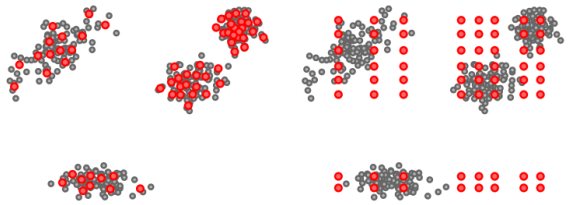
As is known, Product Quantization (PQ) decomposes a high-dimensional vector space into a Cartesian product of subspaces, which are quantized separately. However, to achieve optimal PQ performance, it is essential to determine the ideal space decomposition to efficiently represent the data. It is important to note that increasing the scale of training datasets leads to an increase in codebook size, under reasonable error control [10]. The increase in codebook size corresponds to the rise in data dimensionality and code size, making inverted index compression and search less efficient. In addition, PQ achieves faster distance comparison through dimensionality reduction and distance table lookup, but still

requires brute-force comparison, which slows down the compression process.

To tackle the computationally intensive comparison issues, we present FPQ, a new algorithm that builds upon the existing PQ algorithm by considering centroids distribution and utilizing parallel codebook generation to enhance the performance and efficiency of PQ. In the following section, we provide a detailed explanation of the FPQ algorithm.

A. Open Indexing Compression

In space $\mathbb{R}^d$, Cartesian k-means algorithm can compress data by specifying specifying k centroids. And then $\log_2 k$ bits can be representing any data point any data dimension d. The process of finding these centroids involves brute-force search, which can be computationally expensive at $O(dk)$, and producing low distortion often requires a very large k. Instead, PQ can restrict centroids to an axis-aligned, m-dimensional grid while still having k^m centroids that keep search time at $O(dk)$. Fig. 5 (b) shows that many of these centroids may not have any data points assigned to them. This arises when the distributions on m subspaces are not independent of each other.



(a) k-means (b) PQ

Figure 5: The red points denote centroids, which is train on a random set of points denoted by gray points [11].

In order to conduct fast searches, this algorithm should ultimately be viewed as a form of (lossy) data compression that prioritizes achieving minimal distortion. However, it should be noted that all bits allocated to data points should be used sparingly [11]. The original PQ algorithm assumes that the data points are of normal distribution, which is not always realistic or representative of real-world data. In consequence, some of the centroids learned during training may not receive enough data support, leading to suboptimal performance of the algorithm.

Similar to PQ, our proposed model will first split a vector into sub-vectors and encode each sub-vector using its own PQ codebook. We introduce the algorithm OpenIndexing, inspired by open addressing, through which we try addressing the distribution issue of centroid points. By setting the constant c ($c \geq 1$), we get the equation

$$Limit = \frac{c * DataLength * SubVectorNum}{k} \qquad (3)$$

If the number of points around the centroids exceeds this $Limit$, the next point to support this centroid will be distributed to the next nearest centroids. In this case, the centroid can be trained more evenly supported by more points. We admit a reasonable time and space it brings, but it could make centroids supported by more points. We also tried to integrate it into inverted file, but it failed after days of trials unfortunately.

Althrithom 1: OpenIndexing

Input: TrainData TD, NumSubVector NSV, SplitedData SD SubVectorCentroids SVC, Limit up, compressedData CD.

Output: Code c

```
1. k ←KMeans()
2. map ←hash(int, int)
3. For i in (1…TD.length)
4.       For j in (1…NSV)
5.              code←Find2MinCluster(SD[i][j],SVC[j])
6.              if(map.containKey(code[0]))
7.                     If(map[code[0]]>up)
8.                            c←code[1]
9.              update CD with c
10.      End For
11.End For
```

B. Parallel Codebook Construction

After implementing Product Quantization, we observed that the majority of compression time is spent on the codebook construction. K-means is used to compress each sub-vector with quantizer q_c, which replaces each sub-region of a vector with the closest matching centroid. Although this algorithm is commonly used for clustering tasks in Product Quantization, it is an iterative method that becomes computationally expensive with increasing data points.

Developing memory-efficient algorithms in codebook construction can be advantageous in these scenarios. By implementing brute-force comparison in parallel, the CPU processing power can be utilized efficiently. For example. multithreading implementation eliminates the CPU's need to wait for a specific thread to complete the execution, allowing it to switch to another one. Our proposed FPQ employs these parallelizing techniques to improve the speed during codebook construction through the simultaneous execution of multiple processes or threads. To achieve this, we suggest integrating PKMeans, a highly parallelized algorithm, into the current system. The version can be integrating MapReduce, as proposed by [12], or multithreading, or others. Although there is potential to refine the parallelized K-means algorithms further to increase the speed, this is beyond the scope of this study, and we do not further investigate here.

Althrithom 2: PKMeans

Input: K in K-Means,TrainData T, SplitedData S, SubVectors N,
subVectorCentroids sVC.

Output: TrainData T, SplitedData S

1. For i in (1..S.length)
2. subVector ←S[i]
3. do in parallel with MapReduce/Multithreading
4. For k in (1..K)
5. μ_k ←some random point
6. End For
7. repeat
8. For n in (1..N) do
9. $Z_n \leftarrow \underset{k}{\operatorname{argmin}} \|\mu_k - x_n\|$
10. End For
11. For k in (1..K)
12. $\mu_k \leftarrow MEAN(\{x_n : \mu_k = k\})$
13. End For
14. until converged
15. subVectorCentroids[i] ←Z
16. End For

II. Experimental Study

The primary objective of an image retrieval system is to find a number of images that are similar to the query image. This could involve several technical steps, including image pre-processing and feature extraction. However, this study aims to focus on large-scale vector similarity search, where each feature vector is associated with the corresponding image. Specifically, the goal of the experiment in this study is to evaluate the effectiveness of our approach.

A. Experimental setup

Dataset. In this study, we conduct experiments on the widely used MNIST[1] database of handwritten digits. This database contains 60,000 training images and 10,000 testing images, with each image represented as a 784-dimensional vector. MNIST is commonly used for training image processing systems, making it an ideal choice for our study. To access this dataset, we obtained the necessary training and testing data from Kaggle website[2].

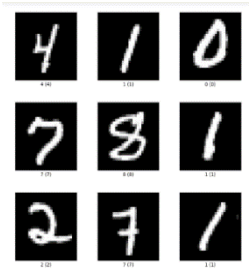


Figure 6: a sample of handwritten digits in MNIST database.

As for the parameter in the algorithm, we strictly follow the recommendation from [9] about the code length and k.

B. Evaluation Indicators

The first evaluation metric in this study is the mean categorization accuracy, or the proportion of test images that are correctly classified. The mean accuracy rate A denotes the ratio of the number of relevant images C among the retrieved images to the total number of query images N.

$$A = \frac{C}{N} \tag{4}$$

As for the memory usage in this study, we focus on heap memory since it is a more effective indicator of overall consumption. The method by which we analyze training and testing times can take place as a combined total or as separate individual evaluations. In addition to those, we take into account other important metrics, such as recall, precision, and fallout rate, which are commonly used to evaluate information retrieval systems. As there are no labels available in the testing dataset, we utilize the training data to predict and compare the results against the existing labels. For our purposes, we define 0 as P and any non-zero value as N. A correct prediction results in T, while an incorrect prediction results in F. For example, any number that is incorrectly classified as zero will be counted as FP. To express these metrics, we use Equations (5), (6), and (7).

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$Fallout = \frac{FP}{FP + FN} \tag{6}$$

$$Precision = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

C. Running environment and Settings.

The running environment details are shown below.

	Details
Programming Language	Java 17, Python 3.9
CPU	Intel (R) Core (TM) i7-7700 HQ @ 2.80GHz
RAM	1× 8 GB DDR4
ROM	1×Phison SM280128 GPTC15T-S114-110
Operating System	Windows 10 64-bit
Development environment	Eclipse (Java), Visual Studio Code (Python)

Table 1. The environment details of the platform in this study.

[1] http://yann.lecun.com/exdb/mnist/

[2] https://www.kaggle.com/competitions/digit-recognizer/data

Building upon [2], our study implements PQ using the Java programming language. And we expand upon said implementation, thus obtaining FPQ. Furthermore, we conduct comparisons with existing code based on scikit-learn, a powerful machine learning module written in python. For this part, we utilize the benchmark[3] from the existing code result.

D. Performance and Result Comparisons

We provide our code on GitHub, which can be verified. In the present study, time elapsed is measured in seconds, while heap memory consumption is measured in MB (megabytes). During all the experiments below, the testing data is set as 5,000, which can be adjusted if deemed necessary. The results running on MNIST are given in Fig 7, 8 and 9.

Our experimental findings indicate a significant performance improvement in terms of time elapsed, particularly in training time, which can be attributed to the utilization of the memory-efficient PKMeans algorithm.

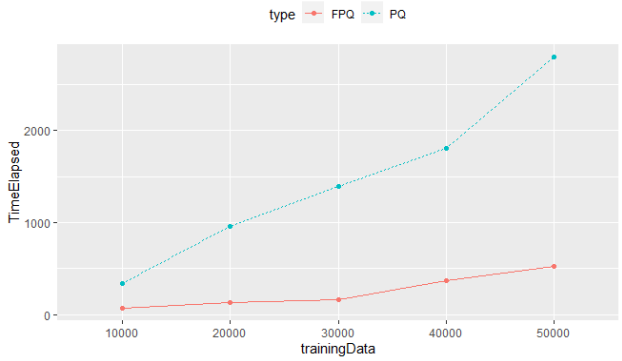


Figure 7: Performance comparison on elapsed time.

Figure 13 and 14 depict the comparison of accuracy and heap memory achieved from our proposed approach with PQ. Our proposed algorithm, OpenIndexing, introduces a small accuracy and memory overhead that warrants further investigation. However, the overhead is relatively minimal compared to the significant performance gain achieved by using FPQ in terms of time and memory. As a result, OpenIndexing allows for the training of significantly more data than PQ, leading to a substantial increase in accuracy rate.

Figure 8: Performance comparison on accuracy.

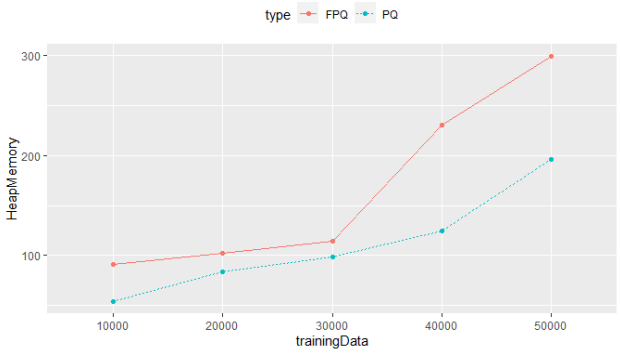


Figure 9: Performance comparison on Memory (Heap).

Our performance evaluation involved a comparison of PQ and FPQ with Scikit-learn implementation, as Table 1 shows. Although FPQ is outperformed by both PQ and Scikit-learn in terms of accuracy, our study revealed that FPQ delivers a significant performance advantage in terms of both elapsed time and memory utilization relative to PQ and Scikit-learn respectively. This considerable improvement can be achieved with only a minor reduction in accuracy. Therefore, the trade-off between performance and accuracy makes FPQ an attractive proposition for applications that demand high-speed processing of large datasets.

Model	PQ (Java)	FPQ (Java)	Sklearn (Python)
Time (s) (Train + Test)	2996.45	498.33	739.28
Memory Consumption	239.5 MB	329.1 MB	1.4 GB
Accuracy (%)	94.32	93.69	94.48

Table 2. The performance evaluation among PQ, FPQ and Scikit-learn in terms of elapsed time, memory consumption and accuracy, where Training Data = 50, 000 and testing data = 5, 000.

[3] https://github.com/alyssaong1/visual-search-demo/blob/master/mnist-faiss.ipynb

E. Discussion and room for improvement.

In this study, we proposed a new algorithm FPQ for Hashing-based ANNs. While our proposed algorithm showed advantages in terms of speed, we observed that it has limitations in accuracy and memory consumption. In this section, we will be discussing these findings and highlight some possible areas for improvement.

Our proposed algorithm prioritizes speed over accuracy, which could be more important in certain applications where real-time training processing is essential. However, in some other applications, accuracy may be more valuable than speed. Therefore, it is necessary to consider the specific use case when evaluating the performance of algorithms.

One of the reasons for the accuracy and memory loss in our proposed algorithm is because of OpenIndexing algorithm. We observed that this has a direct impact on the accuracy of the algorithm. Additionally, the high memory cost is caused by memory-efficient algorithms KPMeans.

Our proposed algorithm's accuracy was evaluated using the MNIST database and relevant metrics. We also examined the performance of Recall, Precision, and Fallout after conducting experiments. However, we noted that these metrics might not be appropriate measures since we predicted outcomes using the training data, and we do not provide further diagram in this study. Nevertheless, they may be useful in evaluating similar image retrieval engines. In addition, it is worth noting that the MNIST database is comparatively small in scale, so evaluating larger databases using these metrics may have certain limitations.

To improve the accuracy and memory efficiency of our proposed algorithm, here we suggest further investigating into the issues of the centroid distribution, which could potentially address the limitations we observed in our project. Also, we recommend exploring more memory-efficient algorithms potentially to be integrated into existing ANNs algorithm, which could help to optimize the algorithm's performance further.

In conclusion, our proposed algorithm has advantages in terms of speed but has limitations in accuracy and memory consumption. We discussed the trade-offs between speed and accuracy, analyzed the causes of the accuracy and memory loss, acknowledged the limitations of the database, evaluation metrics and tools, and suggested future directions for improvement. We hope that our findings will contribute to the ongoing research in the field of Hashing-based ANNs for Large-scale Image Retrieval.

REFERENCES

[1] K. Panchapakesan, “Image processing through vector quantization,” Thesis. *Multimedia Tools and Applications*. 2000.

[2] Y. Matsui, Y. Uchida, H. Jégou, and S. Satoh, “A Survey of Product Quantization,” *ITE TRANSACTIONS ON MEDIA TECHNOLOGY AND APPLICATIONS*, vol. 6, no. 1, pp. 2–10, 2018.

[3] W. Li *et al.*, “Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement,” *IEEE transactions on knowledge and data engineering*, vol. 32, no. 8, pp. 1475–1488, 2020.

[4] O. Jafari, K. M. Islam, and P. Nagarkar, “Drawbacks and Proposed Solutions for Real-time Processing on Existing State-of-the-art Locality Sensitive Hashing Techniques,” *arXiv.org*, 2019.

[5] B. Zheng, X. Zhao, L. Weng, N. Q. V. Hung, H. Liu, and C. S. Jensen, “PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search,” *Proceedings of the VLDB Endowment*, vol. 13, no. 5, pp. 643–655, 2020.

[6] S. Gasiorek, “Counting collisions in an N-billiard system using angles between collision subspaces,” *Symmetry, integrability and geometry, methods and applications*, vol. 16, 2020.

[7] S. O’Hara and B. A. Draper, “Are you using the right approximate nearest neighbor algorithm?,” in *2013 IEEE WORKSHOP ON APPLICATIONS OF COMPUTER VISION (WACV)*, 2013.

[8] P. Chang, “Product Quantization for Similarity Search” [Online] Available: https://towardsdatascience.com/product-quantization-for-similarity-search-2f1f67c5fddd

[9] H. Jégou, M. Douze, and C. Schmid, “Product Quantization for Nearest Neighbor Search,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[10] C.-Y. Chiu, J.-S. Chiu, S. Markchit, and S.-H. Chou, “Effective product quantization-based indexing for nearest neighbor search,” *Multimedia tools and applications*, vol. 78, no. 3, pp. 2877–2895, 2019.

[11] Y. Kalantidis & Y. Avrithis. “Locally Optimized Product Quantization for Approximate Nearest Neighbor Search,” IEEE Computer Vision Foundation. vol. 78, no. 3, pp. 27–55, 2014.

[12] S. Jin, Y. Cui, and C. Yu, “A New Parallelization Method for K-means,” 2016.