

Undergraduate Project Report: Vector Search Algorithms

Pranshu Gaur

April 11, 2023

Contents

1 Introduction	2
2 Developing an Understanding	2
3 Feature Extraction Techniques	2
4 Product Quantization	3
4.1 What is Quantization?	3
4.2 Terms and Terminologies	3
4.3 How to apply the PQ algorithm?	4
4.4 How well does PQ Perform?	5
5 Optimized Product Quantization	5
5.1 What is the need for Optimization?	5
5.2 How do we Optimize PQ?	5
6 Hierarchical Navigable Small Worlds	6
6.1 Idea behind HNSW	6
6.2 Terms and Terminologies	6
6.3 How does HNSW work?	7
6.4 How do the parameters affect HNSW?	8
6.5 What can we infer from the analysis?	8
7 Scalable Nearest Neighbors	9
7.1 Introduction to ScaNN	9
7.2 ScaNN Algorithm: Principle and Methodology:	9
7.3 How well did ScaNN perform?	9
8 Locality Sensitive Hashing	10
8.1 What is LSH?	10
8.2 How does LSH work?	10
8.3 Impact of LSH Parameters on Search Performance	11
8.4 Comparison of LSH and PQ/OPQ Performance:	11
9 Future Prospects	11

Abstract

Vector search algorithms are a class of algorithms that enable efficient search and retrieval of high-dimensional data, such as images, videos, and text. In recent years, there has been significant progress in developing vector search algorithms that achieve state-of-the-art performance on a wide range of applications, including image retrieval, face recognition, and text search.

This project report provides a comprehensive overview of vector search algorithms, including their principles, applications, and implementation details. It starts by introducing the basic concepts of vector quantization and nearest neighbor search, and then proceeds to describe the main algorithms and their variants.

The report also includes a detailed evaluation of the performance of these algorithms on several benchmark datasets, and discusses their strengths and weaknesses. Finally, the report concludes with a discussion of some open research challenges in the field of vector search, and potential directions for future work.

1 Introduction

Despite significant advancements in audio processing, the search for specific audio content remains a challenge. Audio data is often high-dimensional and requires efficient search techniques to retrieve relevant content. Vector search algorithms are designed to address this challenge by quantizing high-dimensional data into a compact representation, and then performing nearest neighbor search in the quantized space.

This report explores the use of vector search algorithms for audio content, specifically investigating their effectiveness in retrieving relevant audio clips from large datasets. We compare the performance of several vector search algorithms, including Product Quantization (PQ), Optimized Product Quantization (OPQ), Locality Sensitive Hashing (LSH), Scalable Nearest Neighbours (ScaNN), and Hierarchical Navigable Small Worlds (HNSW) and evaluate their ability to retrieve audio clips based on specific search criteria.

By examining the principles and limitations of vector search algorithms, we provide insights into how these algorithms can be used to enhance audio search capabilities. The results of our evaluation demonstrate the potential of vector search algorithms in improving the efficiency and effectiveness of audio search, and suggest opportunities for further research in this field.

2 Developing an Understanding

Audio search is the process of finding specific audio content within a large collection of audio files. Traditionally, audio search relied on metadata such as tags or file names to find specific files, but this method can be inaccurate or incomplete. With the rise of machine learning and artificial intelligence, audio search is now often performed using vector search algorithms that convert audio data into high-dimensional vectors and then search for similar vectors in the collection.

Algorithm General Audio Search Algorithm

- Load the Vector Database
 - Select the choice of Algorithm
 - Select the number of closest neighbours to return
 - Build the Searcher/Codebook accordingly
 - Ask for the Query Vectors
 - Implement the Search for each vector
 - Calculate the accuracy of the code by matching the cosine similarity between the query vector and neighbour vector
-

Vector search algorithms are designed to quickly search large databases for vectors that are similar to a query vector. These algorithms work by reducing the dimensionality of the vectors and quantizing them into discrete codes. One widely used approach is Product Quantization (PQ), which divides the vector into multiple subvectors and quantizes each subvector independently. Another popular algorithm is Locality Sensitive Hashing (LSH), which creates hash codes for each vector and then searches for vectors with similar hash codes.

3 Feature Extraction Techniques

Feature extraction is the process of converting raw audio data into high-dimensional feature vectors that can be used by vector search algorithms. There are several techniques for feature extraction from audio, but some of the most common ones are:

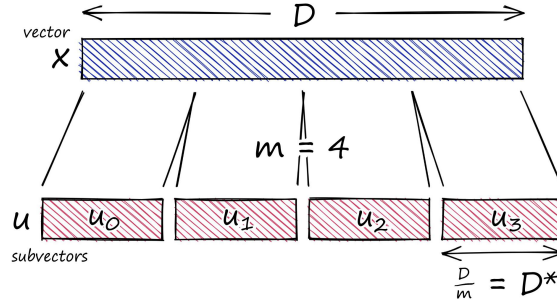
- **Mel-Frequency Cepstral Coefficients (MFCC):** MFCC is a widely used feature extraction technique that aims to mimic the human auditory system’s response to sound. MFCCs extract spectral information from an audio signal and represent it in the form of a feature vector.
- **Spectral Features:** These features include the short-time Fourier transform (STFT), spectrograms, and other spectral representations that capture the frequency and amplitude information of an audio signal.
- **Waveform Features:** Waveform features include zero-crossing rate, energy, and other properties that can be extracted directly from the audio waveform.
- **Deep Learning:** Recently, deep learning-based techniques such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) have been used for feature extraction from audio, achieving state-of-the-art results in many applications.

Once the audio has been transformed into a high-dimensional feature vector, it can be indexed and searched using vector search algorithms like Product Quantization, Optimized Product Quantization, Locality Sensitive Hashing, Scalable Nearest Neighbors, and Hierarchical Navigable Small Worlds.

4 Product Quantization

4.1 What is Quantization?

Quantization is a process of reducing the number of distinct values in a dataset. In the context of Product Quantization (PQ) for vector search algorithms, it involves partitioning a high-dimensional vector into multiple low-dimensional subvectors and then quantizing each subvector into a small number of discrete values. This allows for the use of compact codebooks to represent high-dimensional vectors, which can significantly reduce the memory requirements for vector search.



In PQ, the product of multiple codebooks is used to represent each high-dimensional vector. The process involves splitting a vector into multiple subvectors, each of which is quantized using a separate codebook. The resulting codes for each subvector are concatenated to form a compact representation of the original high-dimensional vector.

4.2 Terms and Terminologies

1. **Codebook:** A set of discrete values or codewords that are used to represent the original continuous data. In PQ, each subvector is quantized using a separate codebook.
2. **Subvector:** A low-dimensional vector obtained by partitioning a high-dimensional vector into multiple disjoint subvectors. In PQ, a high-dimensional vector is split into multiple subvectors, and each subvector is quantized separately.
3. **Centroid:** The average value of a set of data points in a high-dimensional space. In PQ, the codebook is represented by a set of centroids, and each subvector is quantized to the index of its nearest centroid.

4. **D**: The dimensionality of the original high-dimensional vectors.
5. **m**: The number of subvectors into which a high-dimensional vector is partitioned. ‘D’ must be divisible by ‘m’.
6. **k**: The number of codewords or centroids in each codebook. It is generally a power of 2.

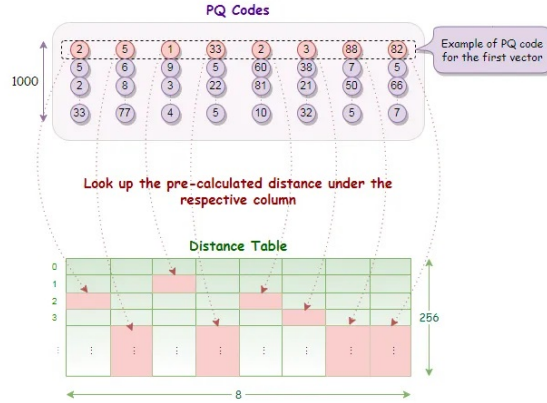
4.3 How to apply the PQ algorithm?

We first start by dividing our D dimensional vector into m subvectors. Let us name the chunks from 0 to $m - 1$. Let us take the first subvector of each vector and call it codebook C_0 . Now, we will use $k - means$ clustering to find k clusters for this set of subvectors. We will add these set of centroids (also called reproduction/reconstruction values) to the codebook C_0 and name each centroid as $C_{0,0}, C_{0,1}, \dots, C_{0,k-1}$. This process would be repeated for each subvector $i \in [m]$. The centroids are represented by $C_{i,j}$ where i is our subvector identifier, and j identifies the chosen cluster.

Next, we encode each of the vectors with a different code (cluster index, i.e. 0 to $k - 1$) by looking for the closest centroid in the codebook for each of the subvector, j . Here, we use L2-norm to calculate the distances.

$$\underset{\mathbf{k}}{\operatorname{argmin}} \sum_{i=1}^{D^*} [x_j^i - C_{j,k}^i]^2$$

Now, our D dimensional vector has been reduced to just m dimensions where each dimension represents the cluster number to which that subvector belongs. We repeat this for each vector to build a code library called PQ_Code.



The final step is to search a query vector using this method. We take a query vector and divide it into similar subvectors, and for each subvector, we pre-calculate the partial squared Euclidean distance with all the centroids of the same subvector from the codebook. We store these results into a matrix, M of size $k \times m$. Now, we can obtain the distance from each vector by summing up the partial distances of each row according to the PQ_Code. The distance of $(p + 1)^{th}$ vector from the query vector is given by,

$$distance[p] = \sum_{j=0}^{m-1} M[PQ_Code[p, j], j]$$

Algorithm 1 Product Quantization Algorithm

- 1: Divide each subvector into m chunks
 - 2: Find k clusters for each set of subvectors, store it as Codebook, $C_{j,k}$
 - 3: Calculate the closest cluster for each subvector of all the vectors, store it as PQ_Code
 - 4: Take the query vector and pre-compute partial distances from each cluster, store it as M
 - 5: Sum the individual distances for each vector according to the PQ_Code
 - 6: Sort the distances and obtain the vectors in order of closest match
-

4.4 How well does PQ Perform?

Product Quantization mainly focuses on reducing the memory used by quantizing the data. The reduction in search time is one of the advantages of this technique which happens because of the pre-computation of partial distances. But, the demerit of this method is that we need to pre-compute the partial distances for each of the query vector and the accuracy of the method is also mediocre.

Value of k	$k = 16$	$k = 32$	$k = 64$	$k = 128$	$k = 256$
Codebook Build Time (s)	26.99	32.62	44.26	58.70	93.31
PQ Code Build Time (s)	1.76	2.24	3.17	5.29	7.73
Total Build Time (s)	28.75	34.86	47.43	63.99	101.04
Total Search Time (s)	5.48	5.46	5.53	5.56	6.20

5 Optimized Product Quantization

5.1 What is the need for Optimization?

The need for optimization in Product Quantization (PQ) arises from the fact that the standard PQ algorithm has limitations in terms of its accuracy and search efficiency. Some of the key limitations include:

1. Suboptimal codebooks: The standard PQ algorithm uses K-means clustering to generate codebooks for each subvector. However, K-means can result in suboptimal codebooks that may not capture the true distribution of the data. This can lead to lower accuracy and higher search times.
2. Codebook size: PQ requires a large number of codewords in the codebook to accurately represent the subvectors. However, a large codebook size can lead to higher memory usage and slower search times.
3. Query time complexity: The standard PQ algorithm has a query time complexity that is linear in the number of subvectors, which can be computationally expensive for large datasets.
4. Search accuracy: The primary goal of vector search algorithms is to find the most similar vectors to a given query vector. However, if the quantization boundaries and codebooks used for subvector quantization are suboptimal, the search accuracy may be reduced. Optimization techniques such as OPQ can help to improve search accuracy by finding better quantization boundaries and codebooks.
5. Generalization: Vector search algorithms may perform well on certain types of datasets but may not generalize well to other datasets. Optimization techniques can help to improve the generalization by finding better solutions that work well across different datasets and vector types.

To address these limitations, various optimization techniques have been proposed for PQ, such as Optimized Product Quantization (OPQ). OPQ improves upon the standard PQ algorithm by optimizing the codebook generation process and reducing the codebook size. It also introduces a rotation matrix to align the subvectors, which can improve the accuracy of the algorithm. By optimizing PQ, it is possible to achieve better search efficiency and accuracy, making it a more effective vector search algorithm.

5.2 How do we Optimize PQ?

Optimized Product Quantization (OPQ) overcomes the challenges of traditional Product Quantization (PQ) by optimizing the codebooks used in the subvector quantization process. Here are some ways in which OPQ achieves this:

1. Rotation matrix: In OPQ, an additional rotation matrix is introduced that rotates the input vectors to a new coordinate system. Let X be the original input matrix and R be the rotation matrix. Then, the rotated input matrix X' can be obtained as $X' = XR$. The rotation matrix R is optimized to minimize the distortion between the input vectors and their corresponding

quantized representations. This is achieved by solving a matrix optimization problem that minimizes the sum of squared errors between X' and its quantized representation.

2. **Learning:** The rotation matrix in OPQ is learned using a gradient descent algorithm that minimizes the distortion between X' and its quantized representation. Let B be the codebook matrix and Q be the quantization function that maps X' to its quantized representation. Then, the distortion between X' and its quantized representation can be measured as the sum of squared errors between X' and $BQ(X')$. The gradient descent algorithm updates the rotation matrix R iteratively by computing the gradient of the distortion with respect to R and adjusting R in the direction of the negative gradient.
3. **Fine-tuning:** OPQ allows for fine-tuning of the codebooks by adjusting the quantization boundaries based on the distribution of the data. Let P be the probability distribution of the input vectors and B be the codebook matrix. Then, the optimal quantization boundaries can be obtained by solving a maximum likelihood estimation problem that maximizes the likelihood of the data given the codebook. This can be achieved using an iterative algorithm that adjusts the quantization boundaries based on the current estimates of P and B .
4. **Compression:** OPQ can be combined with compression techniques such as product hashing or vector quantization to reduce the memory requirements of the codebooks. Product hashing involves hashing the subvectors of the input vectors to a small number of hash tables, and then using the hash tables to lookup the corresponding codebook indices during quantization. Vector quantization involves clustering the subvectors of the input vectors into a small number of centroids, and then using the centroids as the codebook. Both techniques can significantly reduce the memory requirements of the codebooks without sacrificing search accuracy.

Overall, OPQ overcomes the challenges of traditional PQ by optimizing the codebooks used in the subvector quantization process. This can lead to significant improvements in search accuracy, query speed, storage efficiency, and generalization.

6 Hierarchical Navigable Small Worlds

6.1 Idea behind HNSW

It slots into the graph category of ANN algorithm. The basic idea behind HNSW is to create a graph-based structure for the data points, where each node in the graph represents a data point and the edges represent the relationships between the points. The structure is organized into multiple levels, with each level having a smaller number of nodes than the previous level. When a query is made, HNSW starts at the highest level of the graph and uses ANN to find the best match, it then proceeds to send the match to the next level until the last layer is reached.

6.2 Terms and Terminologies

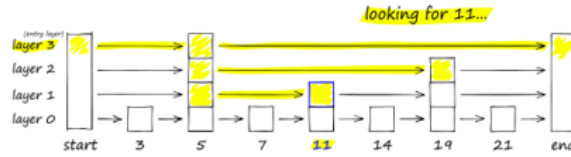
1. **Degree:** Number of links of a Vertex
2. **Level Multiplier m_L :** Normalization constant for probability function of vector insertion
3. **ef_Search:** The number of nearest neighbours to search while proceeding to the next layer
4. **ef_Construction:** The number of nearest neighbours to return while inserting the new element
5. **M:** The number of links to be returned at each layer while insertion
6. **M_max :** Maximum number of neighbours a vertex can have at a layer, usually $= M$
7. **M_maxO :** Maximum number of neighbours a vertex can have at layer 0, usually $= 2*M$
8. **Zoom-Out / Zoom-In:** Search Phase where we pass through low-degree/high-degree vertices

6.3 How does HNSW work?

There are two fundamental techniques that have contributed heavily to HNSW:

1. **Probability Skip Lists:** Probability skip lists are a probabilistic variation of skip lists that were introduced to reduce the search time complexity in hierarchical navigable small world (HNSW) graphs.

In probability skip lists, each node is assigned a probability value p , which is a number between 0 and 1. When we traverse the skip list, we use the probability value of each node to determine whether or not to make a jump to the next node. Specifically, at each node, we generate a random number r between 0 and 1. If $r \leq p$, we make a jump to the next node at the next level. Otherwise, we stay at the current node and continue to the next node at the current level. Otherwise, we stay at the current node and continue to the next node at the current level.



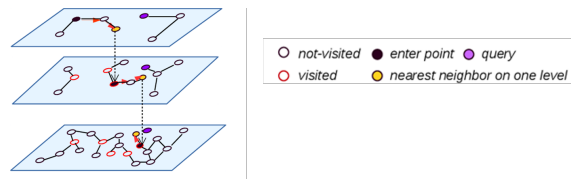
In HNSW graphs, we use probability skip lists to store the neighbors of each point in the graph. Each point is represented as a node in the skip list, and the neighbors of each point are stored as nodes that are linked to the corresponding node in the skip list. The probability values of the nodes in the skip list are chosen in such a way that the resulting graph has a small world property, i.e., each node has a small number of short-range edges and a few long-range edges that connect it to other parts of the graph.

During the search process, we start from a query point and traverse the skip list to find the nearest neighbors of the query point. The probability skip list allows us to quickly skip over parts of the graph that are far away from the query point and focus our search on the parts of the graph that are likely to contain the nearest neighbors. This makes the search process much faster than a brute-force search over all the points in the graph.

2. **Navigable Small World Graphs:** Navigable Small World (NSW) graphs are a type of graph structure that have been designed for efficient nearest neighbor search in high-dimensional spaces. NSW graphs combine the advantages of two other types of graphs: small-world graphs and navigable graphs.

Small-world graphs are characterized by a high clustering coefficient (i.e., neighbors of a node tend to be connected to each other as well) and a low characteristic path length (i.e., the average number of steps it takes to go from one node to another). These properties make small-world graphs efficient for local search, but they may not be optimal for global search.

Navigable graphs, on the other hand, are designed to optimize global search. They achieve this by embedding high-dimensional data into a lower-dimensional space, using techniques such as locality-sensitive hashing or principal component analysis. Once the data is embedded, a graph is constructed on top of the lower-dimensional representation, with the goal of preserving the pairwise distances between data points. This graph can then be searched efficiently using standard graph algorithms.

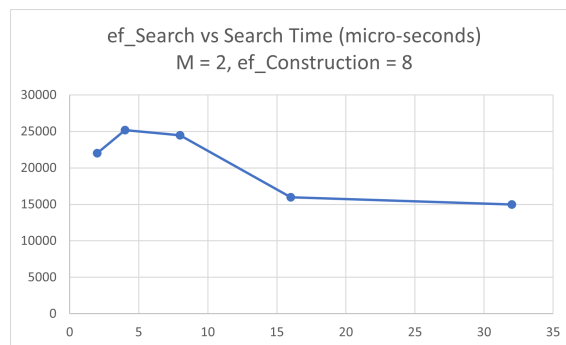
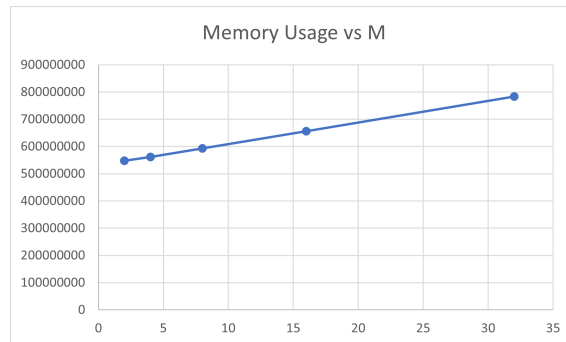
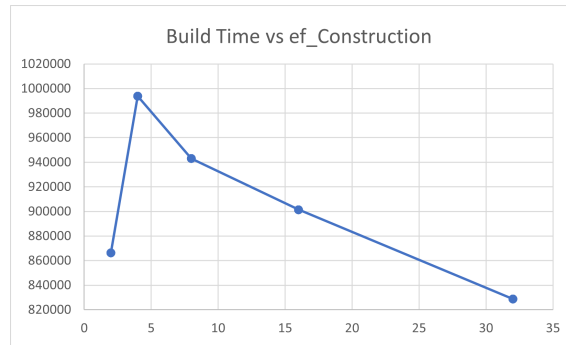


They are constructed by first creating a small-world graph and then augmenting it with long-range edges that are added based on the pairwise distances between data points. The resulting graph has a small characteristic path length, making it efficient for local search, but it also has long-range edges that help to maintain the global structure of the data.

The search algorithm used in NSW graphs is based on a variant of the breadth-first search algorithm, which is used to explore the graph in a way that minimizes the number of distance computations. The algorithm starts at the query point and traverses the graph in a series of hops, using a priority queue to keep track of the next node to visit. At each step, the algorithm selects the node with the smallest estimated distance to the query point, and then visits its neighbors to update their estimates. The algorithm terminates when the desired number of nearest neighbors has been found or when the search has reached a maximum depth.

6.4 How do the parameters affect HNSW?

Here are some of the results based on the implementation of HNSW.



6.5 What can we infer from the analysis?

1. Memory usage only depends on M and depends on it linearly.
2. Build_time depends on M and ef_Construction
3. Increasing M increases the recall as well as search_time, build_time and memory usage
4. No effect of ef_Search, ef_Construction on recall, only search time is affected

- Recall increased from 913 to 941 for when number of nodes to search were increased from 1 to 10 ($M=64$, $ef_Search = 2$, $ef_Construction = 32$)

7 Scalable Nearest Neighbors

7.1 Introduction to ScaNN

Scalable Nearest Neighbors (ScaNN) is an algorithm for fast and efficient approximate nearest neighbor search on large-scale datasets. It is designed to handle high-dimensional vectors and can be used in a variety of applications, such as recommendation systems and image search.

7.2 ScaNN Algorithm: Principle and Methodology:

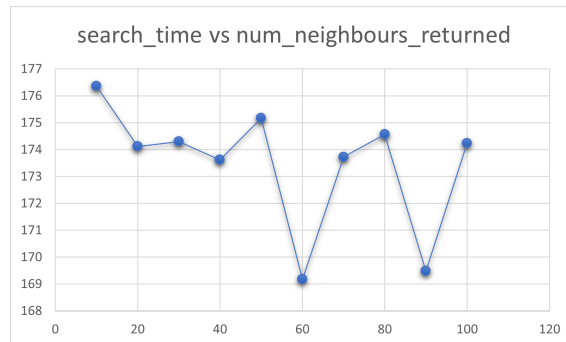
The ScaNN algorithm consists of three main steps: clustering, indexing, and querying.

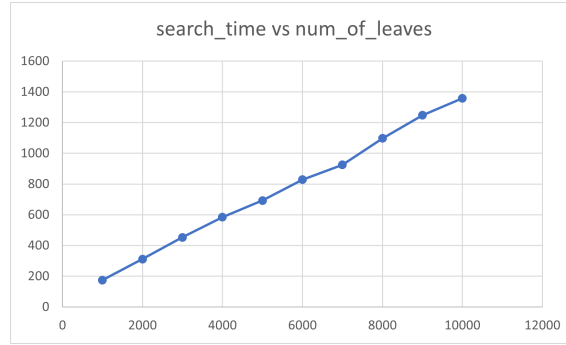
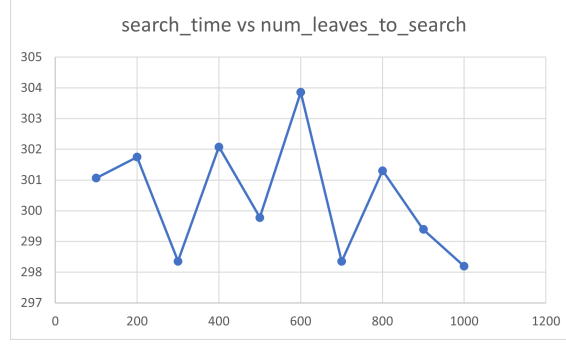
- Clustering:** In the first step, ScaNN partitions the dataset into a set of clusters, each containing a small subset of the total data points. The number of clusters is determined by a user-defined parameter, and the clustering algorithm used can vary (e.g., K-means, Hierarchical clustering, etc.).
- Indexing:** Once the clusters are created, ScaNN constructs an indexing structure that allows for efficient nearest neighbor searches. Specifically, it builds a hierarchical tree of clusters using the clustering labels, with each level of the tree representing a different scale of the dataset. At each level, the clustering labels are used to split the dataset into non-overlapping sub-regions, with each sub-region corresponding to a cluster. ScaNN constructs an inverted index for each sub-region that maps each vector to a unique identifier.
- Querying:** When a query vector is received, ScaNN traverses the hierarchical tree and identifies a small set of candidate clusters that are likely to contain nearest neighbors of the query. The query vector is then compared to the inverted indices of the candidate clusters, and a small set of nearest neighbors is identified. The number of neighbors returned can be adjusted by the user-defined parameters.

ScaNN is an approximate algorithm, which means that it can sacrifice some accuracy to achieve higher search speed. Specifically, ScaNN aims to return a set of points that are near-neighbors to the query, but not necessarily the exact nearest neighbors. The degree of approximation can be adjusted by changing the clustering parameters, indexing structure, or the number of neighbors returned.

Overall, ScaNN is designed to handle high-dimensional vectors and large-scale datasets, and has been shown to be significantly faster than other state-of-the-art algorithms while maintaining comparable accuracy.

7.3 How well did ScaNN perform?





The search_time is almost constant with change in number of neighbours to return or number of leaves to search.

It is linearly increasing with increase in number of leaves of each node.

ScaNN was the fastest and most accurate among the tried algorithms.

8 Locality Sensitive Hashing

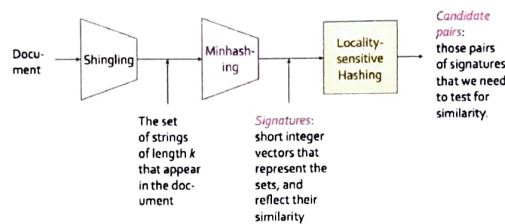
8.1 What is LSH?

Locality Sensitive Hashing (LSH) is a technique for solving the nearest neighbor search problem in high-dimensional data. It is a family of hash functions that maps high-dimensional vectors to binary codes such that the probability of collision between the codes is higher for vectors that are close to each other in the original space. The LSH technique has been widely used in information retrieval, recommendation systems, computer vision, and pattern recognition.

8.2 How does LSH work?

There are some basic principles on which LSH works:

1. Hashing Techniques for High-Dimensional Vector Search: Hashing techniques for high-dimensional vector search aim to convert high-dimensional vectors into low-dimensional binary codes, such that vectors that are close to each other in the original space have a higher probability of being mapped to the same code. These techniques enable fast and efficient search in large databases of high-dimensional vectors.
2. LSH Algorithm: Principle and Methodology: The LSH algorithm uses a family of hash functions to map high-dimensional vectors to binary codes. The hash functions are chosen in such a way that the probability of collision between the codes is higher for vectors that are close to each other in the original space. The LSH algorithm then uses these codes to search for the nearest neighbors of a given query vector in a database of high-dimensional vectors.



One application of LSH is for text data represented as "shingles," which are contiguous subsequences of words. The LSH algorithm can be used to find similar documents by projecting the shingle space into a lower-dimensional space, and using a hash function to map each shingle to a bucket. Documents that have many shingles that map to the same bucket are likely to be similar.

Another key component of LSH is min-hashing, which is used to estimate the similarity of sets. Given two sets A and B, the min-hash value of a set is defined as the minimum hash value of any element in the set. By comparing the min-hash values of two sets, we can estimate their Jaccard similarity, which is the size of the intersection of the sets divided by the size of their union.

In LSH, we use min-hashing to estimate the similarity of points in the projected space. Each hash function used by LSH is defined as a linear combination of the dimensions in the projected space, followed by taking the minimum value. By comparing the hash values of two points, we can estimate their similarity in the original high-dimensional space.

3. LSH Parameters and Configuration: The performance of the LSH algorithm depends on several parameters, such as the number of hash functions, the number of bits per code, and the threshold value for collision. The optimal values of these parameters depend on the specific dataset and the desired search performance. The LSH algorithm can be configured by tuning these parameters to achieve the desired search performance.

8.3 Impact of LSH Parameters on Search Performance

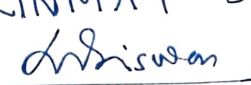
The performance of the LSH algorithm is sensitive to the choice of parameters, such as the number of hash functions and the number of bits per code. Increasing the number of hash functions or the number of bits per code can improve search accuracy, but also increases the computational cost and memory usage. The optimal values of these parameters depend on the specific dataset and the desired search performance.

8.4 Comparison of LSH and PQ/OPQ Performance:

LSH and Product Quantization (PQ) or Optimized Product Quantization (OPQ) are two popular techniques for high-dimensional vector search. LSH is much faster and more memory-efficient than PQ/OPQ, but it has lower search accuracy. The choice of technique depends on the specific dataset and the desired search performance.

9 Future Prospects

In this project I have worked mainly on the query vectors that have exact matches in the vector database, we can also try to implement these algorithms on perturbed queries. We can add noises from Uniform Distribution and then try to see how our algorithms are working. We can also take combinations of many vectors as the audio sound could be bigger. There are many scopes of experiments and I would like to keep on exploring these ideas.

! MRINMAY BISWAS

 11.04.23