Big Data Analysis & Application, CCS&T, Zhejiang University
Laboratory Project

VVS on Large-scale Vector Library

Authors : ChenkaiHu 胡宸恺 ZeyuXu 徐泽宇
KairongHan 韩恺荣 YixiaoLiu 刘奕骁 JianhaoWu 吴剑昊
Date : 2023-11-10

1. Brief Overview

In this project, we are to design an efficient and accurate query algorithm to quickly find the set of vectors most similar to a given query vector in a large-scale vector library.

2. Experimental Setup

2.1 Datasets

- **Dataset A**
  - Number of query vectors *N* = 500
  - Number of vectors in the large-scale vector database *G* = 500,000
  - Vector Dimension = 512
  - Number of most similar vectors to return *K* = 50
- **Dataset B**
  - Number of query vectors *N* = 5,000
  - Number of vectors in the large-scale vector database *G* = 5,000,000
  - Same vector dimension and number of most similar vectors to return

2.2 Hardware and Software Environment

- **Hardware**: Multi-core CPUs and GPUs
- **Software**: A programming environment that supports concurrent queries (specific programming languages and library versions)

2.3 Evaluation Metrics

- **Accuracy**: Assessed using *P@K*
- **Speed**: Calculated as the average query time t_q

3. Algorithm Description

3.1 Selected Algorithms

- **FAISS**
- **HNSW**
- **Other Alternative Algorithms We Tried**

3.2 FAISS

FAISS (Facebook AI Similarity Search) is a powerful similarity search library developed by Facebook (now Meta), designed to handle large-scale vector similarity search tasks, covering image, text, voice and other fields.

In FAISS, FLAT method is one of the basic similarity search methods. The core idea of FLAT (Flat Index) is to tile all vectors in an array and achieve similarity search by calculating the distance between vectors.

Main Steps

Load G and N query vectors from the.npy file. The vector is normalized because FAISS assumes that the data is already normalized.

Create an index using FAISS. We use IndexFlatL2 to create an index of the L2 space. Add the vector in G to this index.

For each query vector, the search method of FAISS is used to find the K vectors that are most similar. If allowed, perform concurrent queries on the GPU to speed up operations.

For accuracy, the standard "answer" in the.pkl file is compared with the algorithm output and calculated P@10. For speed, the time for each query is recorded and the average query time is calculated.

Code

Build Index:

```
def build_index(data_vectors):
    dim, measure = query_emb.shape[1], faiss.METRIC_L2
    param = 'Flat'
    index = faiss.index_factory(dim, param, measure)
    index.add(data_vectors)
    return index
```

Similarity Search:

```
def find_k_similar(query, index, k=10):
    # 查询
    _, faiss_indices = index.search(query, k)
    return faiss_indices
```

Analysis & Optimization

Here's an analysis of why the `Flat` method achieves such high accuracy(99.8% for A, 98.1% for B):

1. **No Approximation Error**: The `Flat` index essentially performs a full brute-force search, comparing against every vector in the database. This means there's no approximation error involved in the search process, ensuring the highest possible accuracy.
2. **No Need for Optimization or Parameter Tuning**: Unlike other index-based search methods, the `Flat` method does not involve any form of data compression or preprocessing, nor does it require tuning of parameters (like quantization details, tree depths, etc.). This means simplicity in implementation while maintaining accuracy.
3. **Direct and Comprehensive Comparisons**: As the `Flat` index involves direct comparison with every vector in the dataset, it ensures a comprehensive search across the entire vector space, thus finding the most accurate nearest neighbors.

To be noted:

For databases that are not too large, such as your experiment case, the 'Flat' method provides the highest accuracy in an acceptable amount of time.

Other attempts on more types of index for FAISS, like IVF/HNSW/.., were also taken. We will mention those later in this paper.

3.3 HNSW

In this approach, we have employed the Hierarchical Navigable Small World (HNSW) algorithm via the nmslib library to conduct similarity searches in a large-scale vector database.

> **The HNSW algorithm** creates a multi-layered graph where each layer is a navigable small world network, enabling efficient and fast approximate nearest neighbor searches by traversing these layers from top to bottom, starting with coarse-grained searches at higher layers and refining the search at lower layers.
>
> **NMSLIB** is an open source library for similarity search, designed to provide efficient approximate nearest neighbor search algorithms.

This method is expected to achieve perfect balance between speed and accuracy, making it an ideal choice for finding the most similar vectors in a high-dimensional space in such a large-scale vector database.

Main Implementation Steps

Building the Index:

- Use the `init` method of NMSLIB to initialize the HNSW index object and choose to use the L2 distance as the similarity metric.

- The index parameters are set with a focus on ensuring a balance between construction efficiency and search quality.

  Defines the parameters of the index, including the maximum number of connections per node (`M`), the search quality parameter at construction (`efConstruction`), and the post-processing parameter (`post`).
- Add the vector library `xb` to the index and call the `createIndex` method to build the index, optionally printing information about the construction progress.
- Finally, save the constructed index to a file for later reuse.

Performing the Search:

- Query parameters are set to optimize the search quality, including the search quality parameter `ef` and the query algorithm selection `searchMethod`.
- Use the `knnQueryBatch` method to perform a batch query on the constructed indexes to get the nearest neighbor indexes for each query vector.

Key Code Analysis

- **Index Creation**: The parameters `M`, `efConstruction`, and `post` are crucial for building an efficient index. `M` determines the maximum number of connections per node, and `efConstruction` affects the search quality.
- **Query Execution**: The search quality parameter `ef` during the query is set high (4000) to ensure accuracy, while the `searchMethod` is chosen for its speed.

Optimization & Analysis

Adjust the parameters of the HNSW algorithm to strike a balance between search efficiency (speed) and accuracy.

1. **Key Parameters**:

   Parameter Tuning: The parameters like `M`, `efConstruction`, and `ef` can be further tuned for different datasets to achieve an optimal balance between speed and accuracy.

   - `M`: the number of bi-directional links to be created for each element in the index. the larger M is, the more accurate it is, but it will increase memory usage and indexing time.
   - `efConstruction`: the size of the dynamic candidate list during index construction. The larger the value, the higher the quality of the constructed index, but it also increases the construction time and memory usage.
   - `ef`: the size of the dynamic candidate list used in the search phase. The larger the value, the more accurate the search, but the slower it is.
   - `post`: The number of additional post-processing steps after index construction. Allows further refinement of the index.

Start with the default or recommended settings. Given higher accuracy is needed, gradually increase `M`, `efConstruction` and `ef`. Observe the effect on accuracy and search time and adjustd accordingly.

Param	Value
M	6
efConstruction	100
post	2
ef	4000

Attention⚠️ In the process of fine-tuning, increasing parameters such as "M" and "efConstruction" can significantly increase memory usage.

This analysis outlines how the nmslib solution provides a high degree of accuracy while also ensuring efficient query processing. By fine-tuning the parameters and leveraging the strengths of HNSW, this approach stands out amon all others.

3.4 More Approaches

Beside the above two ways, we have tried more algorithms listed below.

1. **FAISS + IVF（96.2%）**

   IVF indexing improves search efficiency by partitioning the data into multiple clusters (each represented by a center vector), then determining the nearest cluster centers at query time, and then searching in the vectors corresponding to those centers.

   Compared with simple Flat index, IVF index requires extra memory to store cluster center information. In addition, with uneven data distribution, some clusters may be too dense while others are sparse, which may affect the accuracy and efficiency of the search.

2. **FAISS + HNSW64（85%）**

   FAISS combined with the HNSW algorithm uses a hierarchical graph structure for fast navigation and search, where fast region localization at the higher levels and precise search at the lower levels provide efficient and accurate nearest neighbor lookup.

3. **FAISS + PQ（95%）**

   FAISS combined with PQ enables efficient and storage-intensive vector search by decomposing each high-dimensional vector into multiple subvectors, and quantizing and indexing these subvectors independently.

   PQ indexing is able to provide faster searches, but this usually comes at the expense of some search accuracy.

4. ANNOY（**90.9%**）

Annoy implements nearest-neighbor search using a tree-based ensemble, where each tree is an incomplete partition of the dataset by means of a random projection. This approach creates an incomplete but efficient data structure in each tree for fast approximate search.

5. **Neural Network**

   While deep learning-based methods perform well in feature extraction and certain types of similarity search tasks, they may not be the best choice for large-scale vector-efficient similarity search tasks, given the consumption of computational and memory resources, scalability and flexibility limitations, and model generalization capabilities and interpretability issues.

   In these scenarios, traditional indexing methods (e.g., tree-based methods, hashing, or quantization methods) that are more lightweight and easier to scale and update are often more appropriate.

4. Experimental Results

4.1 FAISS

Results: FAISS+FLAT

Indicator	**Value**
Total Query Time	1.03 seconds
Average Time	2.06 ms
Accuracy	99.80%
Accuracy for B	**98.10%**

4.2 HNSW

Results: HNSW of nmslib

Indicator	**Value**
Total Query Time	2.6622 seconds
Average Time	5.3244 ms
Accuracy for A	99.80%
Accuracy for B	**97.08%**

4.3 Analysis and Discussion of Results

4.3.1 Effectiveness of Optimization Strategies

Index Optimization: Analyze the impact of using different types of indexes (e.g., Flat, IVF, HNSW) on search performance.

Parameter Tuning: Explore the impact of tuning parameters (e.g., number of clustering centers, number of trees, etc.) on algorithm performance.

Parallelization and Hardware Optimization: evaluate the role of hardware acceleration using multi-core CPUs, GPUs, etc. on improving search efficiency.

4.3.2 Possible Directions for Improvements

Algorithm Hybridization: Consider combining multiple algorithms, e.g., Annoy for fast initial screening followed by FAISS for exact search.

Dynamic Update Mechanisms: Explore the performance of algorithms when data is dynamically updated (e.g., new vectors are added in real time), and ways to improve this.

Automatic parameter tuning: investigate the use of machine learning or heuristic algorithms to automatically tune search parameters for different datasets and query loads.

Deep Learning Model Improvement: Explore ways to optimize the model structure and training process to improve efficiency and accuracy if deep learning-based approaches are used.

Interpretability and visualization: develop tools and methods to improve the interpretability of algorithms to help users better understand search results.

6. Conclusion

In this experiment, we implement several cosine similarity-based vector query algorithms, such as FAISS and HNSW, for fast and accurate querying by building index structures. The experimental results show that these algorithms perform well in large-scale vector libraries.

However, in practical applications, different indexing structures and similarity measures can be selected according to specific scenarios to further fine-tune the parameters to maximize the performance of the algorithms.