Efficient Hybrid Search in Vector Databases

Abstract

Hybrid search refers to a type of searching that combines both vector search and scalar search, i.e., a similarity search with scalar attribute filtering. Despite being under-researched, it in fact has many everyday-life applications. For example, it can be used to find similar images with a specific size, or related songs marked with certain tags.

In this project, we compare a few KNN searching algorithms to identify the most suitable ones for hybrid search. In addition, we propose a concurrent filtering algorithm that speeds up hybrid searches and improves its usability.

1. Introduction

This project has two goals: first, to examine existing KNN algorithms and identify appropriate ones for hybrid search, and second, to propose the concurrent filtering algorithm that can potentially improve the efficiency.

* 1. KNN Search

According to previous survey [1], mainstream KNN solutions can be classified into three types: hashing-based, partition-based, and graph-based. The main difference is the form of the index.

Hashing-based KNN

The index of such methods is a hash table that assigns points to different buckets based on their location. When a query point is given, only the most relevant buckets need to be searched. Examples include Locality sensitive hashing (LSH) and Learning to Hash method (L2H). These two types evolve into more variants, and their major difference is whether the hash function is data-independent or learnt from the data distribution.

Partition-based KNN

For this kind of KNN, the index is usually a tree that partitions the vector space recursively based on specific standards. To find the neighbors of a query point efficiently, it is necessary to determine which branches to search accordingly. Some examples are VP-Tree, Ball Tree, KD-Tree and Annoy.

Graph-based KNN

These algorithms use a graph to reconstruct the spatial relationship between data points in the original space. Since information loss inevitable during the reconstruction with a simple graph, such methods are usually not suitable for exact search. However, empirically the accuracy can still be near 100%. Many methods based on KNN Graph, HNSW, and SW belong to this category.

The results shown in [1] indicate that not all methods have the same efficiency. Graph-based methods tend to be the most efficient, while hashing-based and partition-based methods are shown to be 1 to 100 times slower on a few real datasets. However, partition-based, and hashing-based methods also have their advantages, including the potential for exact search and shorter index building time.

In this project, we compare several KNN methods using library implementations to verify the results of [1] and try to select appropriate ones for hybrid search. We obtained similar results to [1] in terms of efficiency. In addition, we believe all three types can be helpful when conducting hybrid search. For hashing based KNN, if the filtering criteria is integrated into the hash function, it is possible to simplify hybrid search into a simple KNN search. For the other KNN methods, some modification to the index may be needed.

* 1. Attribute Filtering

Traditionally, there have been two major approaches to attribute filtering: pre-query filtering, and post-query filtering. Although more sophisticated solutions exist as pointed out by Zilliz [2], they can be regarded as derivations of the two methods.

If we define hybrid search as finding items that simultaneously satisfy some vector similarity criteria (CV) and some scalar attribute criteria (CA), then pre-query filtering is to verify CA before CV and post-query filtering is to verify CV before CA.

For example, the Zilliz paper [2] lists five hybrid search methods, of which strategies A and B belong to pre-query filtering, strategy C belongs to post-query filtering, and strategies D and E are a combination of both.

1. Proposed Method

Figure 1 illustrates the workflow of both pre-query and post-query filtering. Furthermore, it shows **concurrent filtering**, which is a method we propose that effectively combines the advantages of the other two.

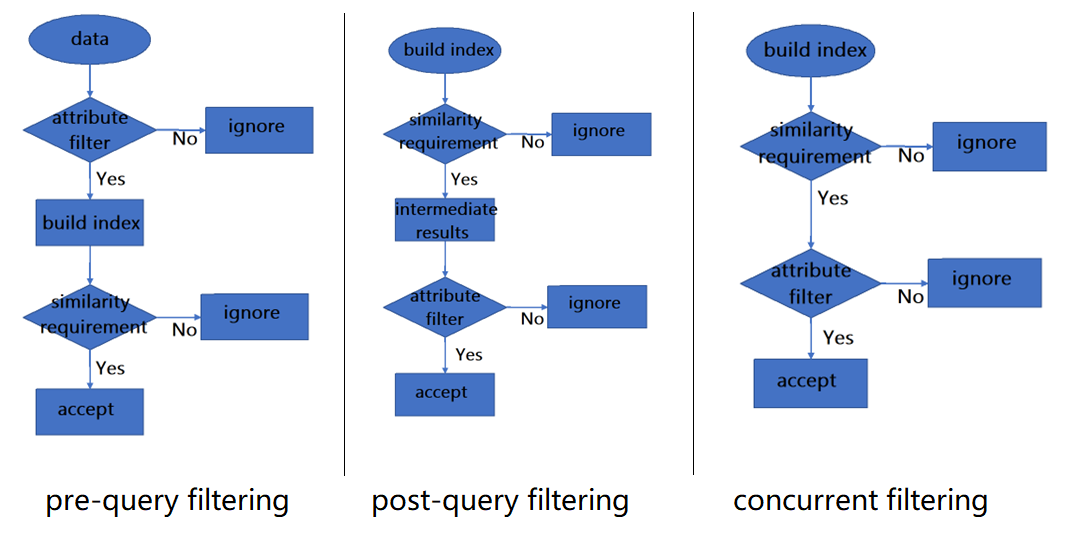


Figure 1: a comparison of attribute filtering algorithms

Both pre-query filtering and post-query filtering come with drawbacks. Pre-query filtering is more efficient and can return exactly as many results as requested, but its index is query-dependent, i.e., a new index needs to be built to meet new requirements whenever CA changes. Post-query filtering is more flexible in this sense, but it is less efficient, and the number of results returned is not precise because no one knows how many of similarity search outputs can pass CA filtering.

To address these problems, we propose concurrent filtering, which is a combination of the two existing methods. While a full index is built as in post-query filtering, CA filter is applied during (instead of after) vector search and information can be exchanged with the similarity search algorithm. In this way, it achieves post-query filtering’s maximum flexibility, reduces overhead caused by considering points that do not fulfill CA, and returns exactly as many results as requested through dynamic management of the result set. It effectively combining the advantages and removing the disadvantages of both pre-query and post-query filtering.

1. Experimental Results

We use nmslib (<https://github.com/nmslib/nmslib>) and annoy (<https://github.com/spotify/annoy>) to compare the efficiency of some KNN algorithms. Additionally, we conduct a series of experiments to compare concurrent filtering’s performance against pre-query filtering and post-query filtering. We use VP-Tree and NSW (a variant of SW) as similarity search algorithms.

We use Euclidean distance as metric and all other parameters as default. Three datasets are used for this comparison: GloVe 50d with 400,000 entries of 50-dimensional vectors (<https://nlp.stanford.edu/data/glove.6B.zip>), NUS-WIDE Normalized\_CH with 269,648 entries of 64-dimensional vectors (<https://lms.comp.nus.edu.sg/wp-content/uploads/2019/research/nuswide/NUS_WID_Low_Level_Features.rar>), and VirusShare (redundant features removed) with 107,856 entries of 265-dimensional vectors (<https://archive.ics.uci.edu/ml/machine-learning-databases/00413/dataset.zip>). For each of the datasets, the efficiency of KNN algorithms is compared by measuring the time needed for 100 queries where the query point is randomly selected from the first 100,000 entries and k, the number of nearest neighbors to be returned, is set to be 20, 1000, and 20000 respectively in three groups of experiments. For the comparison among pre-query, post-query and concurrent filtering, we use own implementations (<https://github.com/gzlzgzl/HybridSearch>) of VP-Tree and NSW on GloVe 50d with the filtering criteria “word length is even”.

* 1. Figures

2.1.1. KNN Efficiency Comparison

We obtain results that are comparable to [1]. Graph-based methods are the most efficient and are 10 to 100 times faster than partition-based methods. Hashing-based methods are not included because they do not fit into the concurrent filtering framework.

Figure 1: running time (ms) vs. method for k=2

Figure 2: running time (ms) vs. method for k=1000

Figure 3: running time (ms) vs. method for k=20000

2.1.3. Evaluation of Concurrent Filtering

Below is the time per single query for the three approaches on GloVe 50d. Note that this cannot solely rank the effectiveness of the algorithms. Here, we list two major reasons:

First, the index building time is not included and it is much longer than the time for a single query. The index of pre-query filtering only works for one filtering criteria, but the index for post-query and concurrent filtering work for all criteria. This can be observed from the flowchart in section 2.1.2 (only pre-query filtering builds the index after applying attribute filter).

Second, unlike the other two counterparts, post-query filtering cannot return exactly as many results as requested, because there is no way for the attribute filter to provide feedback to the vector search algorithm on how many points have been validated.

Figure 7: running time (ms) vs. different implementations of VP-Tree

Figure 8: running time (ms) vs. different implementations of NSW

* 1. Tables

Raw data obtained from our experiments are given below.

2.2.1. Comparison between KNN algorithms

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Time (ms) of 100 queries of ANN methods | | | | | | |
| k=20 |  |  |  |  |  |  |
| Trial 1 | HNSW | Annoy | SW | NAPP | VP-Tree | Brute force |
| GloVe | 1 | 2 | 3 | 630 | 909 | 2235 |
| NUS-WIDE | 4 | 3 | 6 | 761 | 204 | 1439 |
| VirusShare | 3 | 10 | 6 | 621 | 12 | 1386 |
| Trial 2 | HNSW | Annoy | SW | NAPP | VP-Tree | Brute force |
| GloVe | 5 | 2 | 10 | 1678 | 1730 | 2322 |
| NUS-WIDE | 4 | 2 | 6 | 749 | 194 | 1459 |
| VirusShare | 4 | 10 | 24 | 428 | 22 | 1385 |
|  |  |  |  |  |  |  |
| k=1000 |  |  |  |  |  |  |
| Trial 1 | HNSW | Annoy | SW | NAPP | VP-Tree | Brute force |
| GloVe | 29 | 345 | 29 | 2831 | 3151 | 3579 |
| NUS-WIDE | 16 | 265 | 16 | 1261 | 1031 | 2293 |
| VirusShare | 16 | 344 | 47 | 734 | 250 | 2214 |
| Trial 2 | HNSW | Annoy | SW | NAPP | VP-Tree | Brute force |
| GloVe | 31 | 360 | 31 | 2867 | 3258 | 3656 |
| NUS-WIDE | 16 | 266 | 16 | 1250 | 996 | 2368 |
| VirusShare | 16 | 359 | 47 | 766 | 281 | 2215 |
|  |  |  |  |  |  |  |
| k=20000 |  |  |  |  |  |  |
| Trial 1 | HNSW | Annoy | SW | NAPP | VP-Tree | Brute force |
| GloVe | 33 | 657 | 29 | 5339 | 5626 | 6050 |
| NUS-WIDE | 16 | 5083 | 16 | 2974 | 3380 | 4398 |
| VirusShare | 16 | 3897 | 63 | 1261 | 1938 | 3755 |
| Trial 2 | HNSW | Annoy | SW | NAPP | VP-Tree | Brute force |
| GloVe | 29 | 784 | 28 | 5359 | 5670 | 6145 |
| NUS-WIDE | 16 | 4900 | 16 | 2902 | 3213 | 4270 |
| VirusShare | 16 | 3913 | 78 | 1327 | 2043 | 3716 |

2.2.2. Comparison between attribute filtering algorithms

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hybrid Search Algorithms on GloVe 50d (time in ms) | | | | | | |
|  | VP-Tree | | | NSW | | |
| Trial 1 | Pre-query filtering | Post-query filtering | Concurrent filtering | Pre-query filtering | Post-query filtering | Concurrent filtering |
| k=20 | 86 | 193 | 196 | 9 | 9 | 4 |
| k=2000 | 112 | 263 | 217 | 334 | 953 | 279 |
| k=20000 | 221 | 487 | 395 | 1759 | 3093 | 1698 |
| k=100000 | 395 | 966 | 531 | 1821 | 4346 | 2357 |
| Trial 2 | Pre-query filtering | Post-query filtering | Concurrent filtering | Pre-query filtering | Post-query filtering | Concurrent filtering |
| k=20 | 85 | 190 | 190 | 8 | 8 | 2 |
| k=2000 | 124 | 270 | 228 | 328 | 476 | 248 |
| k=20000 | 218 | 502 | 397 | 1672 | 3743 | 1446 |
| k=100000 | 410 | 958 | 532 | 1926 | 4510 | 2157 |
| Trial 3 | Pre-query filtering | Post-query filtering | Concurrent filtering | Pre-query filtering | Post-query filtering | Concurrent filtering |
| k=20 | 82 | 189 | 193 | 8 | 9 | 6 |
| k=2000 | 113 | 276 | 210 | 210 | 1057 | 237 |
| k=20000 | 231 | 572 | 373 | 1537 | 3465 | 1614 |
| k=100000 | 416 | 1043 | 546 | 1926 | 4072 | 2232 |

2.3. Math and Equations

//to add (optional): time complexity of 3 versions of VP-Tree and NSW, and when should concurrent filtering be chosen over pre-query filtering

1. Citations

[1] 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, doi: 10.1109/TKDE.2019.2909204.

[2] J. Wang et al. “Milvus: A Purpose-Built Vector Data Management System,” SIGMOD/PODS '21: International Conference on Management of Data, 2021.

//to add: original publication for each KNN algorithm