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

Two objectives are set for this project: first, examine existing KNN algorithms and identify appropriate ones for hybrid search, and second, propose the concurrent filtering algorithm that can potentially improve performance.

1.1. 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 hand-crafted 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 guidelines. To find 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 is inevitable during reconstruction, such methods are usually not suitable for exact search. However, empirically, the accuracy can be very close to 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 select appropriate ones for hybrid search. We have obtained similar results to [1] in terms of efficiency. In addition, we believe all three categories of KNN 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 standard KNN search. For the other categories, some modification to the index may be needed.

1.2. 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 (C_V) and some scalar attribute criteria (C_A), then pre-query filtering is to verify C_A before C_V and post-query filtering is to verify C_V before C_A.

For example, the Zilliz paper [2] lists five hybrid search methods. While 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.

2. Proposed Method

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

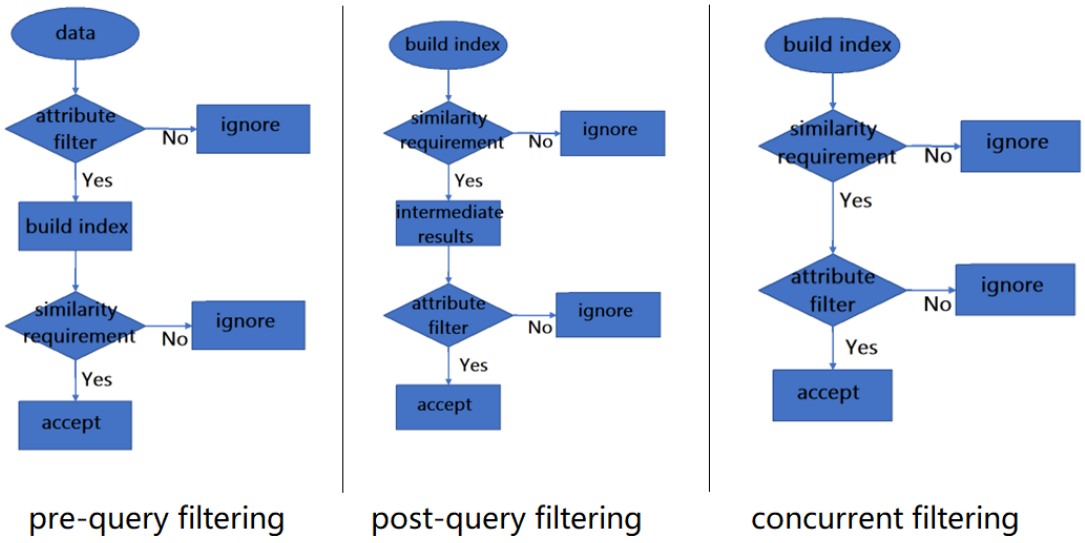

Figure 1: a comparison of attribute filtering algorithms

Pre-query Filtering

This strategy selects all points that pass attribute filtering and then conducts similarity within the result. For example, suppose we want to find houses of size 700 – 900 feet that are close to the airport. Following this approach, we will first select all houses that are 700 – 900 feet, and then determine which ones of these are close to the airport.

Post-query Filtering

This method first uses vector search to obtain similar results to the query point, and then conducts attribute filtering. For instance, we first search for all houses that are close to the airport, and then select those that are 700 – 900 feet as the result.

Concurrent Filtering

Instead of treating vector search and attribute filtering as two separate stages, we propose to apply two criteria simultaneously. This means both location and size are considered when we determine whether a house should be added to the list of interest.

Pre-query filtering and post-query filtering are obviously independent of the similarity search algorithm because attribute filtering is conducted outside of its execution interval. However, how concurrent filtering can be implemented solely depends on the vector search method. In general, concurrent filtering can be applied if the KNN searching algorithm uses a dynamic result list that records intermediate results. Algorithm 1 is an example of integrating concurrent filtering into VP-Tree. VP-Tree is a partition-based KNN algorithm that recursively selects a random vantage point (VP) to split a space into two subtrees, which contain points that are closer to and farther from the VP, respectively. The boundary can be considered as a hypersphere whose radius is the median of distances from other points to VP.

Algorithm 1: VP-Tree Concurrent Filtering

Constant parameters:
Point *query*, Integer *k*

Global variables:
Maxheap *result* /* *result* stores intermediate KNN results at a specific moment, ordered by distance to the *query* point, and it has a maximum capacity of *k*. Further insertion implies replacement of the root node. */

Input: Node *node* /* *node* is the current node, which contains a vantage *point* and a *radius* */

Output: *result*

1. /*Identify the threshold *t* that determines whether new entries are accepted into *result* */
2. **if** *result* is not full
3. $t \leftarrow \infty$
4. **Else**
5. $t \leftarrow$ max distance in *result*
6. /* *d* is the distance between the current point and the query point */
7. *d* ← EuclideanDistance (*node.point*, *query*)
8. /* *r* is the radius of the current node that separates the inner and outer subtree */
9. *r* ← *node.radius*
10. /* determine if the current point should be added to the intermediate result. */
11. **if** *d*<=*t* **and** *node.point* fulfills attribute filtering criteria

```
12.             add node.point to result
13.   /* search the inner subtree if it overlaps with the query area */
14.   if d<=r+t
15.             search(inner)
16.   /* search the outer subtree if it overlaps with the query area */
17.   if d>=r-t
18.             search(outer)
```

The difference compared to standard similarity search is the underlined part of line 11 above. By altering the condition for a point to join the intermediate result, the new algorithm becomes capable of considering similarity and attribute information simultaneously. In this concurrent filtering setting, points that cannot pass attribute filtering remain in the index, but cannot be accepted as candidates, and they only function as bridges between other points in the index.

Both pre-query filtering and post-query filtering suffer from significant drawbacks. Pre-query filtering is 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 the attribute filtering criteria change. Post-query filtering is more flexible in this sense, but it is inefficient, and the number of results returned is not precise because it is unpredictable how many points can pass attribute filtering after finishing the similarity search.

We combine the two methods as concurrent filtering to address these problems. While a full index is built as in post-query filtering, attribute filtering is applied during (instead of after) vector search and feedback can be delivered to the similarity search engine. In this way, it not only achieves post-query filtering's high flexibility, but also reduces overhead caused by accepting points that cannot pass attribute filtering. It also returns exactly as many results as requested through dynamic management of the result set. It effectively integrates the advantages and resolves the disadvantages of both pre-query and post-query filtering. The characteristics of the three methods are summarized in Table 1.

Method	flexible index (against different filtering criteria)	efficiency (per query)	number of results
pre-query filtering	no	very high	exact
concurrent filtering	yes	high	exact
post-query filtering	yes	low	unspecific

Table 1: qualitative comparison between attribute filtering approaches

3. 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".

3.1 Figures

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.

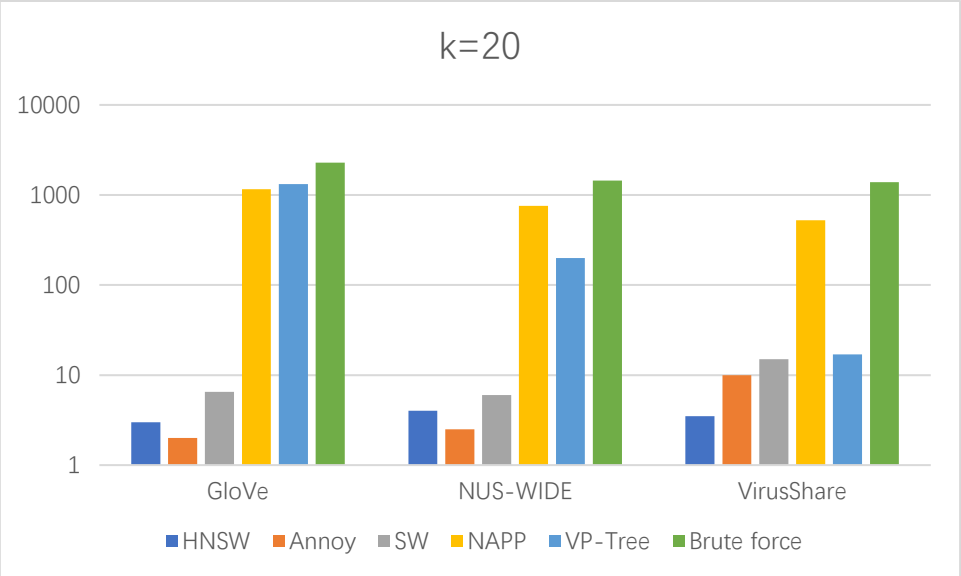

Figure 2-1: running time (ms) vs. method for k=2

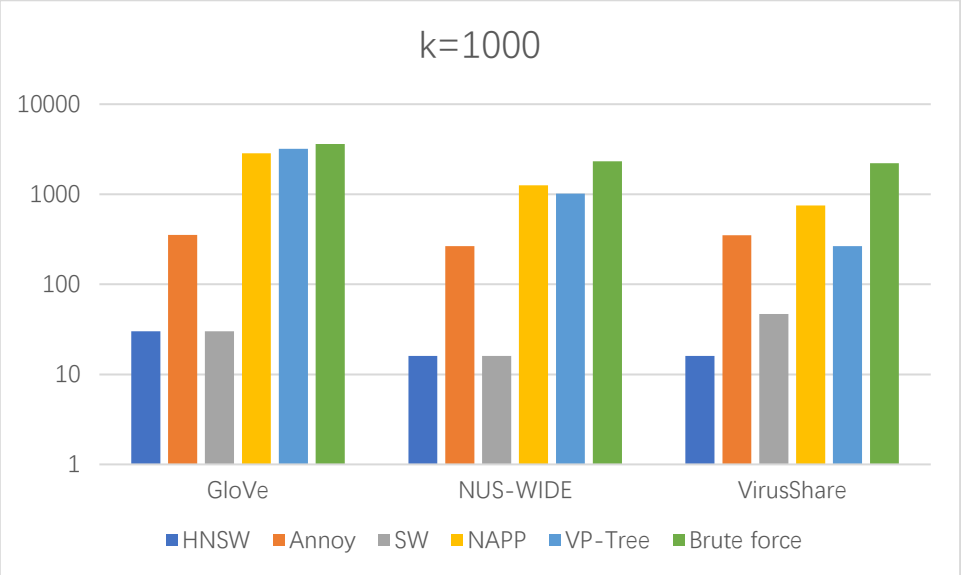

Figure 2-2: running time (ms) vs. method for k=1000

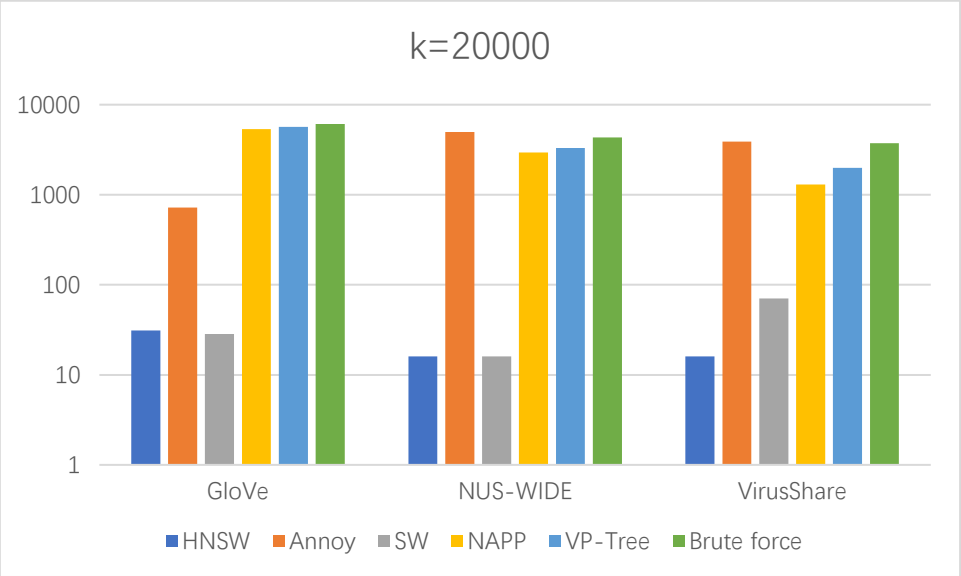

Figure 2-3: running time (ms) vs. method for k=20000

Evaluation of Concurrent Filtering

Figures 7 and 8 present the time per single query for the three approaches on the GloVe 50d dataset. Note that this cannot solely rank the effectiveness of the algorithms. Here are 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 Figure 1 (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.

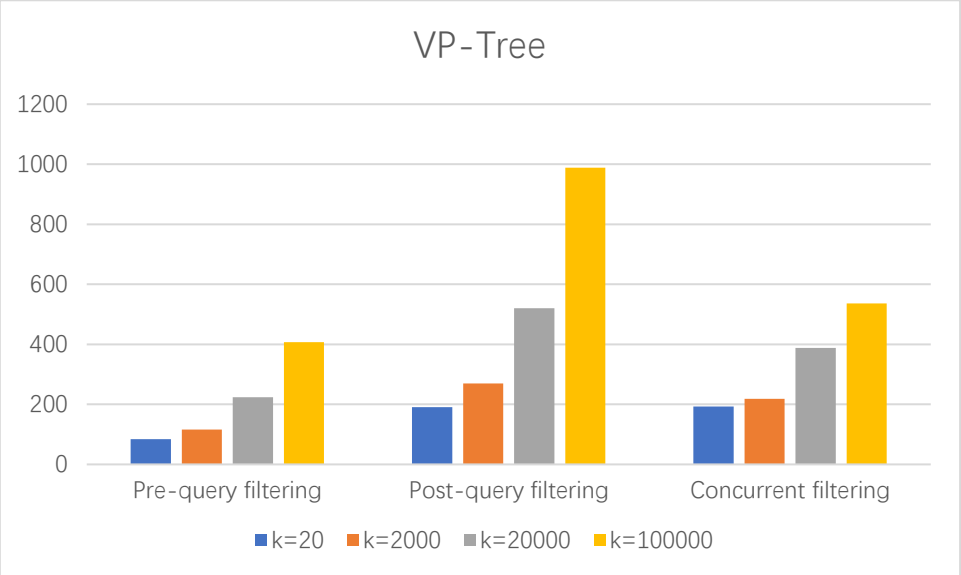

Figure 3-1: running time (ms) vs. different implementations of VP-Tree

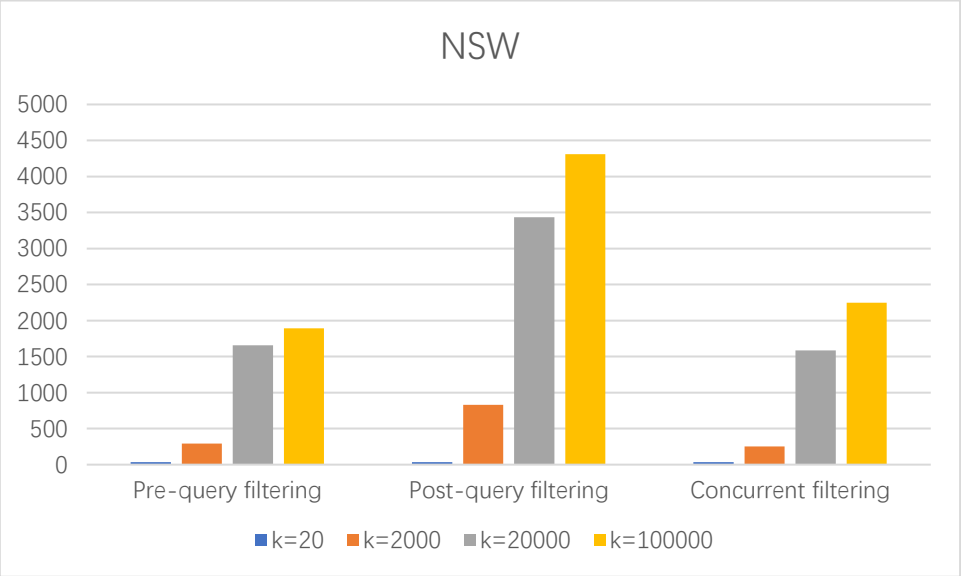

Figure 3-2: running time (ms) vs. different implementations of NSW

3.2 Tables

Raw data obtained from our experiments are given below.

3.2.1. 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

3.2.2. 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

3.3. 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

4. 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