Why NOT Operations are Difficult in Vector Search

Jaepil Jeong
Cognica, Inc.
Email: jaepil@cognica.io
Date: February 2, 2025

1. Vector Search and the Challenge of NOT Operations

With the recent surge in embedding (vector) based search technology, we can now find documents through *meaning* rather than keyword matching. For example, searches for "AI research" and "artificial intelligence related papers" can yield similar contextual results. However, one challenging aspect of vector search is that it's difficult to naturally support **NOT operations** (negative operations), which are commonly used in traditional keyword search.

Why is it difficult?

1. **Vector Search Results are 'Score' Centered**
   In traditional search, `NOT` conditions are simply implemented as "exclude documents containing specific words." In vector search, documents (or data) are represented by how "similar" they are through distance or cosine similarity. It's tricky to set criteria like "exclude anything **too** similar to this document," and it's not easy to efficiently search vector indexes based on such criteria.
2. **Mismatch Between ANN (Approximate Nearest Neighbor) Index and Negative Conditions**
   Vector search engines typically use ANN algorithms optimized for "quickly finding the most similar results (nearest neighbors)." Conversely, `NOT` operations need to "maximally exclude items similar to specific vectors," which can be the opposite of what ANN index structures are designed for.
3. **Inefficiency of Post-filtering**
   The most common workaround is to first retrieve the top results with high similarity, then remove (set difference) those that match the NOT condition (results that are **too** similar to the vector to be negated). However, this approach requires retrieving a sufficiently large number of top results, which can easily degrade query performance.

2. Theoretical Background

Looking briefly at vector search theory, it maps text or images to numerical vectors in a d-dimensional space through embedding, then finds similar objects using **distance** or **similarity** measures with the query vector.

2.1 Embedding and Similarity Metrics

- **Embedding**: A function $f(T) = \mathbf{v}$ that maps text T to a d-dimensional vector $\mathbf{v} \in \mathbb{R}^d$. This can be obtained from various models including neural networks.
- **Similarity Metrics**: Common metrics include cosine similarity and Euclidean distance, defined as follows:

- Cosine Similarity: $\text{sim}(\mathbf{v}, \mathbf{u}) = \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{v}\|\|\mathbf{u}\|}$
- Euclidean Distance: $d(\mathbf{v}, \mathbf{u}) = \|\mathbf{v} - \mathbf{u}\|_2 = \sqrt{\sum_{i=1}^{d}(v_i - u_i)^2}$

Finding `Top-K` results in vector search means finding the k vectors with the highest similarity to (or shortest distance from) the query vector Q.

2.2 Approximate Nearest Neighbor Search (ANN)

When dealing with large-scale data, calculating similarities between embedding vectors one by one takes $O(N)$ time, which is highly inefficient. To solve this, **Approximate Nearest Neighbor (ANN)** techniques are widely used. For example, various structures like HNSW, IVF, and PQ can significantly reduce search time.

The general nearest neighbor problem can be defined as follows:

- **Dataset**: $S \subset \mathbb{R}^d$, $|S| = N$
- **Query Vector**: $q \in \mathbb{R}^d$
- **Distance Function**: $d(x, q)$, e.g., Euclidean distance

The exact nearest neighbor $\text{NN}(q)$ is an $x \in S$ that satisfies:

$$\text{NN}(q) = \arg\min_{x \in S} d(x, q) \tag{1}$$

However, performing this process exactly every time for large-scale data is computationally expensive. Therefore, the core of the **ANN problem** is to reduce computation time by allowing approximation. Usually, for an approximation factor $c > 1$, finding an $x \in S$ that satisfies:

$$d(x, q) \leq c \cdot d(\text{NN}(q), q) \tag{2}$$

is called a c-approximate nearest neighbor (c-ANN).

In other words, when $c = 1$, it's exact (precise) search, and when $c > 1$, it allows some margin of error in exchange for dramatically faster search speed. ANN algorithms are typically designed to efficiently find points that satisfy such approximation conditions.

3. Efficient Intersection/Difference Operations

To overcome these limitations, Aeca Database implements an **index structure that can perform efficient set operations (intersection, difference)** internally, directly supporting `NOT` operations for vector search. The key is having a structure that can perform intersection and difference operations at the **database engine level**, rather than simply post-processing Top-K lists.

3.1 Hybrid Index Structure

- **Hybrid Indexing**: Aeca Database has a structure that **combines vector indexes with traditional indexes (B-Tree, Bitmap, etc.)** on top of a high-performance OLTP engine.
  - For example, basic text filters (tags, keywords, date ranges, etc.) are quickly filtered using B-Tree or Bitmap indexes.
  - The remaining candidate set is then queried using vector indexes to rank by similarity (or distance).

- **Set Operation Engine**: It's designed to quickly perform operations like intersection, union, and difference between these candidate sets. This allows natural combination of traditional index-based filters with vector search conditions.

3.2 Implementing NOT Using Intersection/Difference Operations

Let's mathematically express the desired conditions in vector search. If we want to "see only items among Top-K similar to query vector Q that are **not too close** (similarity < τ) to specific vector B":

$$C_Q = \{x \mid \text{similarity}(x, Q) > \alpha\} \tag{3}$$

$$C_B = \{x \mid \text{similarity}(x, B) > \beta\} \tag{4}$$

Here, C_Q is the set of items with similarity above α to query Q, and C_B is the set of items with similarity above β to vector B that we want to exclude. The NOT operation can be expressed as the set difference $C_Q \setminus C_B$.

Internally, Aeca Database:

1. **Uses ANN indexes** to quickly generate candidate sets C_Q and C_B.
    - For example, using HNSW-based indexes, it finds k similar items to Q for C_Q, and does the same process for C_B.
2. **Efficiently calculates** $C_Q \setminus C_B$ through **hybrid indexes** and the **set operation engine**.
    - Regular ANN libraries would need to get two candidate sets and implement set difference through separate post-processing routines.
    - Aeca Database optimizes this at the DB engine level, executing it within a single query plan.
3. The internal query planner optimizes operations by efficiently merging/splitting C_Q and C_B in various scenarios, such as when C_B is very large or C_Q is below certain thresholds.

This process is not just conceptual set operations but is practically performant thanks to Aeca Database's architecture that **combines OLTP with vector search**.

4. Query Example

Let $Q = [0.1, 0.2, \dots]$ and $B = [0.3, 0.4, \dots]$ be vectors from embedding models. The following expresses a query like "find documents with tag='LLM' that are similar to query vector Q but not similar to specific domain vector B":

```
{
  "$search": {
   "query": "tag:LLM AND vector:[0.1, 0.2, ...] AND NOT vector:[0.3, 0.4, ...]"
  }
}
```

Aeca Database follows this process:

1. Quickly filters document candidates using the traditional index for **tag='LLM'**.
2. Finds set C_Q of candidates with **vectors similar to** Q from the vector index.

3. Simultaneously finds set C_B of candidates with **vectors similar to** B from the vector index.
4. The final result is $C_Q \cap (\text{tag}{=}\text{LLM}) \setminus C_B$, combining intersection and difference operations.
5. Internally, while using ANN indexes to find C_Q and C_B, the set difference operation is handled by the DB engine's set operation module.

This shows how text filters (traditional indexes), vector filters (ANN indexes), and set operations (intersection/difference) organically combine to effectively handle NOT operations.

5. Additional Technical Details

1. **Index Implementation**:
   - Vector Index: Uses HNSW, IVF, or PQ-based structures to generate Top-K lists like C_Q, C_B.
   - Traditional Index: Uses B-Tree, LSM-Tree (sorted keys), Bitmap (binary attributes), etc. for fast filtering.
2. **Set Operation Optimization**:
   - The DB engine predicts (based on statistics) how large C_Q and C_B are, and how likely they are to be empty sets, to determine the optimal operation order.
   - Example: If $|C_B|$ is small, it might be advantageous to quickly find C_B first and then subtract it from C_Q (set difference).
   - Conversely, if C_B is very large, C_Q might be found first, and then C_B can be selectively calculated only for the intersection area.
3. **Complex Queries**:
   - Besides NOT operations, it supports logical operations like AND, OR, allowing processing of complex queries combining metadata + vector search + negative conditions in a single step.

6. Conclusion

- NOT operations in vector search fundamentally need to maintain *low* similarity, which tends to conflict with ANN structures designed to find "nearest neighbors."
- Existing approaches (post-processing, query vector adjustment, etc.) are not intuitive and face performance degradation or inaccuracy issues.
- **Aeca Database** directly supports NOT operations in vector search through its **hybrid index structure** and **native set operation (intersection, difference) support**.
  - Quickly calculates $C_Q \setminus C_B$ by distinguishing between C_Q and C_B.
  - Efficiently processes multiple filters (AND, OR, NOT, etc.) within a single query plan.
  - Internal query planner considers set sizes and statistical information to optimize operation order.

This approach expands the utility of vector search and enables more sophisticated meaning-based search. It can be particularly useful in **hybrid search** environments combining keywords and vector operations. As vector search engines evolve to handle various complex queries, approaches like Aeca Database's are expected to become increasingly important.

References

- Facebook AI Research, Faiss
- Spotify, Annoy
- Yury Malkov et al., HNSW
- DB research materials related to Hybrid Indexing and Query Planner