벡터 검색에서 NOT 연산이 어려운 이유

Jaepil Jeong
Cognica, Inc.
이메일: jaepil@cognica.io
날짜: 2025년 2월 2일

1. 벡터 검색과 NOT 연산의 어려움

최근 임베딩(벡터) 기반 검색 기술이 급부상하면서, 키워드 매칭이 아닌 '의미'를 통해 문서를 찾을 수 있게 되었습니다. 예를 들어 "AI 연구"와 "인공지능 관련 논문"이 서로 비슷한 맥락의 검색 결과를 제공할 수 있는 식입니다. 그런데 벡터 검색에서 한 가지 까다로운 점은, 전통적인 키워드 검색에서는 흔히 쓰이는 **NOT 연산**(부정 연산)을 자연스럽게 지원하기 어렵다는 점입니다.

왜 어려울까요?

1. **벡터 검색 결과는 '점수(Score)' 중심입니다**
   전통적인 검색에서는 `NOT` 조건을 "특정 단어를 포함한 문서를 제외"와 같은 형태로 간단히 구현합니다. 벡터 검색에서는 문서(또는 데이터)가 거리나 코사인 유사도를 통해 얼마나 "유사한지"로 표현됩니다. "이 문서와 **너무** 유사한 것은 제외"라는 기준을 정하기가 까다롭고, 그 기준에 따라 벡터 인덱스를 효율적으로 검색하기가 쉽지 않습니다.
2. **ANN(Approximate Nearest Neighbor) 인덱스와 부정 조건의 불일치**
   벡터 검색 엔진은 일반적으로 "가장 유사한 결과(최근접 이웃)를 빠르게 찾는" 데 최적화된 ANN 알고리즘을 사용합니다. 반면에 `NOT` 연산은 "특정 벡터와 유사한 항목을 최대한 배제"해야 하는데, 이는 ANN 인덱스 구조의 설계 목적과 정반대일 수 있습니다.
3. **후처리(Post-filtering)의 비효율성**
   가장 흔한 우회 방법은 유사도가 높은 상위 결과를 먼저 가져온 다음, NOT 조건에 해당하는 것(부정할 벡터와 **너무** 유사한 결과)을 차집합으로 제거하는 방식입니다. 하지만 이 방식은 충분히 많은 상위 결과를 가져와야 하며, 이 과정에서 쿼리 성능이 쉽게 저하될 수 있습니다.

2. 이론적 배경

벡터 검색 이론을 간단히 살펴보면, 텍스트나 이미지를 임베딩을 통해 d차원 공간의 수치 벡터로 매핑한 뒤, 쿼리 벡터와의 **거리(distance)** 또는 **유사도(similarity)**를 사용하여 비슷한 객체를 찾습니다.

2.1 임베딩과 유사도 지표

- **임베딩**: 텍스트 T를 d차원 벡터 $\mathbf{v} \in \mathbb{R}^d$로 매핑하는 함수 $f(T) = \mathbf{v}$입니다. 이는 신경망 등 다양한 모델로부터 얻을 수 있습니다.
- **유사도 지표**: 일반적으로 사용되는 지표에는 코사인 유사도와 유클리디안 거리가 있으며, 다음과 같이 정의됩니다:
  - 코사인 유사도: $\text{sim}(\mathbf{v}, \mathbf{u}) = \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{v}\|\|\mathbf{u}\|}$
  - 유클리디안 거리: $d(\mathbf{v}, \mathbf{u}) = \|\mathbf{v} - \mathbf{u}\|_2 = \sqrt{\sum_{i=1}^{d}(v_i - u_i)^2}$

벡터 검색에서 `Top-K` 결과를 찾는다는 것은 쿼리 벡터 Q와 가장 유사도가 높은(또는 거리가 가장 가까운) k개의 벡터를 찾는 것을 의미합니다.

2.2 근사 최근접 이웃 검색 (ANN)

대규모 데이터를 다룰 때 임베딩 벡터 간의 유사도를 하나씩 계산하는 것은 $O(N)$ 시간이 걸려 매우 비효율적입니다. 이를 해결하기 위해 **근사 최근접 이웃(Approximate Nearest Neighbor, ANN)** 기법이 널리 사용됩니다. 예를 들어, HNSW, IVF, PQ 등 다양한 구조를 통해 검색 시간을 크게 단축할 수 있습니다.

일반적인 최근접 이웃 문제는 다음과 같이 정의됩니다:

- **데이터셋**: $S \subset \mathbb{R}^d$, $|S| = N$
- **쿼리 벡터**: $q \in \mathbb{R}^d$
- **거리 함수**: $d(x, q)$, 예: 유클리디안 거리

정확한 최근접 이웃 $\mathrm{NN}(q)$는 다음을 만족하는 $x \in S$를 의미합니다:

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

하지만 대규모 데이터에서 이 과정을 매번 정확히 수행하는 것은 계산 비용이 매우 큽니다. 따라서 **ANN 문제**의 핵심은 근사를 허용하여 계산 시간을 단축하는 것입니다. 일반적으로 근사 인자 $c > 1$에 대해, 다음을 만족하는 $x \in S$를 찾는 것을:

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

c-근사 최근접 이웃(c-ANN)이라고 합니다.

다시 말해, $c = 1$일 때는 정확한(정밀) 검색이고, $c > 1$일 때는 어느 정도의 오차 범위를 허용하는 대신 검색 속도가 획기적으로 빨라질 수 있습니다. ANN 알고리즘은 일반적으로 이러한 근사 조건을 만족하는 점을 효율적으로 찾도록 설계됩니다.

3. 효율적인 교집합/차집합 연산

이러한 한계를 극복하기 위해, Aeca Database는 내부적으로 **효율적인 집합 연산(교집합, 차집합)을 수행할 수 있는 인덱스 구조**를 구현하여 벡터 검색에 대한 `NOT` 연산을 직접 지원합니다. 핵심은 단순히 Top-K 리스트를 후처리하는 것이 아니라, **데이터베이스 엔진 레벨**에서 교집합과 차집합 연산을 수행할 수 있는 구조를 가지는 것입니다.

3.1 하이브리드 인덱스 구조

- **하이브리드 인덱싱**: Aeca Database는 고성능 OLTP 엔진 위에 **벡터 인덱스와 전통적 인덱스(B-Tree, Bitmap 등)를 결합**한 구조를 가집니다.
  - 예를 들어, 기본적인 텍스트 필터(태그, 키워드, 날짜 범위 등)는 B-Tree나 Bitmap 인덱스를 사용하여 빠르게 필터링합니다.
  - 남은 후보 집합은 벡터 인덱스를 사용하여 유사도(또는 거리) 기준으로 순위를 매깁니다.
- **집합 연산 엔진**: 이러한 후보 집합들 간의 교집합, 합집합, 차집합 같은 연산을 빠르게 수행하도록 설계되었습니다. 이를 통해 전통적 인덱스 기반 필터와 벡터 검색 조건을 자연스럽게 결합할 수 있습니다.

3.2 교집합/차집합 연산을 이용한 NOT 구현

벡터 검색에서 원하는 조건을 수학적으로 표현해 봅시다. "쿼리 벡터 Q와 유사한 상위 K개 항목 중에서 특정 벡터 B와 **너무 가깝지 않은**(유사도 < τ) 것만 보고 싶다"라고 하면:

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

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

여기서 C_Q는 쿼리 Q에 대해 유사도가 α 이상인 집합이고, C_B는 배제하고 싶은 벡터 B와 유사도가 β 이상인 집합입니다. NOT 연산은 차집합 $C_Q \setminus C_B$로 표현할 수 있습니다.

Aeca Database는 내부적으로:

1. **ANN 인덱스**를 사용하여 후보 집합 C_Q와 C_B를 빠르게 생성합니다.
    - 예를 들어, HNSW 기반 인덱스를 사용하여 Q와 유사한 항목 k개를 찾아 C_Q를 구하고, C_B에 대해서도 같은 과정을 수행합니다.
2. **하이브리드 인덱스**와 **집합 연산 엔진**을 통해 $C_Q \setminus C_B$를 효율적으로 계산합니다.
    - 일반적인 ANN 라이브러리는 두 후보 집합을 얻은 후 별도의 후처리 루틴으로 차집합을 구현해야 합니다.
    - Aeca Database는 이를 DB 엔진 레벨에서 최적화하여 단일 쿼리 계획 내에서 실행할 수 있습니다.
3. 내부 쿼리 플래너는 C_B가 매우 크거나 C_Q가 특정 임계값 이하인 경우 등 다양한 시나리오에서 C_Q와 C_B를 효율적으로 병합/분할하여 연산을 최적화합니다.

이 과정은 단순한 개념적 집합 연산이 아니라, Aeca Database가 **OLTP와 벡터 검색을 결합**한 아키텍처 덕분에 성능적으로도 실용적입니다.

4. 쿼리 예시

임베딩 모델에서 나온 벡터 $Q = [0.1, 0.2, \dots]$와 $B = [0.3, 0.4, \dots]$가 있다고 합시다. 다음은 "tag='LLM'이면서 쿼리 벡터 Q와 유사하지만 특정 도메인 벡터 B와는 유사하지 않은 문서를 찾아라"라는 쿼리를 표현한 것입니다:

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

Aeca Database는 다음 과정을 따릅니다:

1. 전통적 인덱스를 사용하여 **tag='LLM'** 조건에 맞는 문서 후보를 빠르게 필터링합니다.
2. 벡터 인덱스에서 Q**와 유사한 벡터**를 가진 후보 집합 C_Q를 찾습니다.
3. 동시에 벡터 인덱스에서 B**와 유사한 벡터**를 가진 후보 집합 C_B를 찾습니다.
4. 최종 결과는 교집합과 차집합 연산을 결합한 $C_Q \cap (\text{tag}{=}\text{LLM}) \setminus C_B$입니다.
5. 내부적으로 C_Q와 C_B를 찾기 위해 ANN 인덱스를 사용하는 동안, 차집합 연산은 DB 엔진의 집합 연산 모듈이 처리합니다.

이는 텍스트 필터(전통적 인덱스), 벡터 필터(ANN 인덱스), 그리고 집합 연산(교집합/차집합)이 유기적으로 결합되어 NOT 연산을 효과적으로 처리하는 방법을 보여줍니다.

5. 추가 기술 상세

1. **인덱스 구현**:
    - 벡터 인덱스: C_Q, C_B 같은 Top-K 리스트를 생성하기 위해 HNSW, IVF, 또는 PQ 기반 구조를 사용합니다.
    - 전통적 인덱스: 빠른 필터링을 위해 B-Tree, LSM-Tree(정렬된 키), Bitmap(이진 속성) 등을 사용합니다.
2. **집합 연산 최적화**:
    - DB 엔진은 C_Q와 C_B가 얼마나 큰지, 그리고 공집합일 가능성이 얼마나 되는지를 통계를 기반으로 예측하여 최적의 연산 순서를 결정합니다.

    - 예: $|C_B|$가 작다면, 먼저 C_B를 빠르게 찾은 다음 C_Q에서 빼는(차집합) 방식이 유리할 수 있습니다.
    - 반대로, C_B가 매우 크다면 C_Q를 먼저 찾고, 교집합 영역에 대해서만 선택적으로 C_B를 계산할 수 있습니다.
3. **복잡한 쿼리**:
    - NOT 연산 외에도 AND, OR 같은 논리 연산을 지원하여 메타데이터 + 벡터 검색 + 부정 조건이 결합된 복잡한 쿼리를 단일 실행 계획으로 처리할 수 있습니다.

6. 결론

- 벡터 검색에서 NOT 연산은 기본적으로 *낮은* 유사도를 유지해야 하는데, 이는 "최근접 이웃"을 찾도록 설계된 ANN 구조와 충돌하는 경향이 있습니다.
- 기존 접근 방식(후처리, 쿼리 벡터 조정 등)은 직관적이지 않고 성능 저하나 부정확성 문제에 직면합니다.
- **Aeca Database**는 **하이브리드 인덱스 구조**와 **네이티브 집합 연산(교집합, 차집합) 지원**을 통해 벡터 검색에서 NOT 연산을 직접 지원합니다.
    - C_Q와 C_B를 구분하여 $C_Q \setminus C_B$를 빠르게 계산합니다.
    - 여러 필터(AND, OR, NOT 등)를 단일 쿼리 실행 계획 내에서 효율적으로 처리합니다.
    - 내부 쿼리 플래너가 집합 크기와 통계 정보를 고려하여 최적의 연산 순서를 결정합니다.

이러한 접근 방식은 벡터 검색의 활용성을 확장하고 더 정교한 의미 기반 검색을 가능하게 합니다. 특히 키워드와 벡터 연산을 결합한 **하이브리드 검색** 환경에서 유용하게 활용될 수 있습니다. 벡터 검색 엔진이 다양한 복잡한 쿼리를 처리하는 방향으로 발전함에 따라 Aeca Database와 같은 접근 방식이 점점 더 중요해질 것으로 예상됩니다.

참고 자료

- Facebook AI Research, Faiss
- Spotify, Annoy
- Yury Malkov et al., HNSW
- 하이브리드 인덱싱 및 쿼리 플래너 관련 DB 연구 자료