벡터 검색에서 NOT 연산이 어려운 이유

1. 벡터 검색과 NOT 연산의 어려움

최근 임베딩(벡터) 기반 검색 기술이 급부상하면서, 키워드 매칭이 아니라 '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 식이죠. 그런데 벡터 검색에서 한 가지 까다로운 점은, 전통적인 키워드 검색에서는 흔히 쓰이는 **NOT 연산**(부정 연산)을 **자연스럽게** 지원하기 어렵다는 점입니다.

왜 어려울까?

1. **벡터 검색의 결과는 '순위(Score)' 중심**
   전통 검색에서는 `NOT` 조건을 간단히 "특정 단어를 포함한 문서를 제외" 같은 형태로 구현합니다. 벡터 검색에서는 문서(혹은 데이터)가 "어느 정도로" 유사한가를 거리나 코사인 유사도로 표현합니다. "이 문서와 **너무** 유사한 것은 제외"라는 기준을 잡기 까다롭고, 그 기준에 따라 벡터 인덱스를 효율적으로 탐색하기가 쉽지 않습니다.
2. **ANN(Approximate Nearest Neighbor) 인덱스와 부정 조건의 미스매치**
   벡터 검색 엔진은 대개 "가장 유사한 결과(최근접 이웃)를 빨리 찾는" 데 최적화된 ANN 알고리즘을 사용합니다. 반대로 `NOT` 연산은 "특정 벡터와 유사한 것은 최대한 배제"를 해야 하는데, ANN 인덱스 구조와는 정반대의 요구사항일 수 있습니다.
3. **후처리(Post-filtering)의 비효율성**
   가장 흔히 쓰이는 우회 방법은, 유사도 높은 상위 결과를 뽑아놓고 그중 NOT 조건에 해당하는 것(부정할 벡터와 '너무' 유사한 결과)을 다시 제거(차집합)하는 방식입니다. 단, 이렇게 하려면 상위 결과를 충분히 많이 뽑아야 하며, 그 과정에서 쿼리 성능이 저하되기 쉽습니다.

---

2. 이론적 배경

벡터 검색의 이론을 간단히 살펴보면, 텍스트나 이미지를 수치 벡터로 임베딩(Embedding)하여 d-차원 공간에 매핑한 뒤, 쿼리 벡터와의 **거리(distance)** 혹은 **유사도(similarity)**를 통해 비슷한 객체를 찾습니다.

2.1 임베딩(Embedding)과 유사도 지표

- **임베딩**: 텍스트 T를 d-차원 벡터 $\mathbf{v} \in \mathbb{R}^d$로 매핑하는 함수 $f(T) = \mathbf{v}$. 이는 신경망 등 다양한 모델로부터 얻을 수 있습니다.
- **유사도 지표**: 흔히 사용되는 지표로는 코사인 유사도와 유클리디안 거리가 있으며, 이 둘은 다음과 같이 정의됩니다.
  - 코사인 유사도: $\mathrm{sim}(\mathbf{v}, \mathbf{u}) = \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{v}\| \|\mathbf{u}\|}$
  - 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 Q와의 유사도가 가장 높은(또는 거리가 가장 가까운) k개의 벡터를 찾는 과정을 의미합니다.

2.2 근사 최근접 이웃 탐색 (ANN)

데이터가 대량일 때, 임베딩 벡터 간 유사도를 하나하나 계산하는 것은 $O(N)$ 시간이 걸려 매우 비효율적입니다. 이를 해결하기 위해 **Approximate Nearest Neighbor (ANN)** 기법이 널리 사용됩니다. 예를 들어, HNSW나 IVF, PQ 등 다양한 구조를 통해 검색 속도를 크게 단축할 수 있습니다.

일반적인 최근접 이웃(Nearest Neighbor) 문제를 정의하면 다음과 같습니다:

- **데이터 세트**: $S \subset \mathbb{R}^d$, $|S| = N$.
- **쿼리 벡터**: $q \in \mathbb{R}^d$.
- **거리 함수**: $d(x, q)$, 예를 들어 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 대용량 데이터에서 이 과정을 매번 정확히 수행하는 것은 계산 비용이 너무 큽니다. 따라서 근사(Approximation)를 허용하여 계산 시간을 단축하는 것이 **ANN 문제**의 핵심입니다. 보통 근사도가 $c > 1$인 경우,

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

를 만족하는 $x \in S$를 찾으면, 이를 c-근사 최근접 이웃(c-ANN)라고 부릅니다.

즉, $c = 1$이면 정확한(정밀) 검색, $c > 1$이면 어느 정도 오차 범위를 허용하는 대신 검색 속도가 비약적으로 빨라질 수 있습니다. ANN 알고리즘은 보통 이런 근사 조건을 만족하는 점을 효율적으로 찾아내도록 설계됩니다.

---

3. Aeca Database의 해결 방법: 효율적인 교집합/차집합 연산

Aeca Database는 이러한 한계를 극복하고자, 내부적으로 **'효율적인 집합 연산(교집합, 차집합)을 할 수 있는 인덱스 구조'**를 구현하여, 벡터 검색에 대한 `NOT` 연산을 직접 지원합니다. 단순히 Top-k 리스트를 구한 뒤 후처리하는 것이 아니라, **데이터베이스 엔진 레벨**에서 교집합과 차집합을 수행할 수 있는 구조가 핵심입니다.

3.1 하이브리드 인덱스 구조

- **하이브리드 인덱싱**: Aeca Database는 고성능 OLTP 엔진 위에 **벡터 인덱스와 전통 인덱스(B-Tree, Bitmap 등)를 결합**한 구조를 가집니다.
  - 예컨대, 기본 텍스트 필터(태그, 키워드, 날짜 범위 등)는 B-Tree나 Bitmap 인덱스로 빠르게 추립니다.
  - 그 후 남은 후보 집합을 벡터 인덱스로 조회하여, 유사도(혹은 거리) 기준으로 순위를 매깁니다.
- **집합 연산 엔진**: 이렇게 추린 후보 집합끼리 교집합, 합집합, 차집합 같은 연산을 빠르게 수행할 수 있도록 설계되어 있습니다. 따라서 전통 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 항목들(Top-k) 중, 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 집합, C_B는 배제하고 싶은 벡터 B와 유사도가 β 이상인 집합입니다. NOT 연산은 곧 차집합 $C_Q \setminus C_B$ 로 표현할 수 있습니다.

Aeca Database는 내부적으로

1. **ANN 인덱스**를 이용해 C_Q와 C_B 각 후보 집합을 빠르게 생성합니다.

- 예: HNSW 기반 인덱스로, C_Q를 찾기 위해 Q와 유사한 항목을 k개 뽑고, C_B에 대해서도 같은 과정을 진행합니다.

2. **하이브리드 인덱스** 및 **집합 연산 엔진**을 통해 $C_Q \setminus C_B$를 효율적으로 계산합니다.
   - 일반 ANN 라이브러리로는 후보 집합을 2번 구한 뒤, 별도의 후처리 루틴으로 차집합을 구현해야 합니다.
   - Aeca Database는 DB 엔진 레벨에서 이를 최적화하여, 동일한 질의 계획(Plan) 안에서 수행할 수 있게 합니다.
3. 상황에 따라, C_B가 매우 큰 경우나 C_Q가 특정 임계값 이하인 경우 등 다양한 시나리오에서, 내부 쿼리 플래너가 C_Q와 C_B를 효율적으로 병합/분할하여 연산을 최적화합니다.

이 과정은 단순히 개념적인 집합 연산에 그치지 않고, Aeca Database가 **OLTP와 벡터 검색을 결합**한 아키텍처 덕분에 성능적으로도 실용적입니다.

---

4. 질의 예시

다음은 "tag='LLM'이면서, 쿼리 벡터 Q와 유사한 문서를 찾되, 특정 분야 벡터 B와는 유사하지 않았으면 좋겠다"라는 질의를 표현한 것입니다.

```
{}
```

Aeca Database는 다음과 같은 과정을 거칩니다.

1. **tag='LLM'** 필터로 전통 인덱스에서 문서 후보를 빠르게 추립니다.
2. 해당 후보들에 대해 Q**와의 벡터 유사도**가 0.8 이상인 집합 C_Q을 구합니다.
3. 동시에 B**와의 벡터 유사도**가 0.9 이상인 집합 C_B도 벡터 인덱스에서 추립니다.
4. 최종 결과는 $C_Q \cap (\text{tag}=\text{'LLM'}) \setminus C_B$ 로, 교집합과 차집합을 결합하여 원하는 결과를 얻습니다.
5. 내부적으로, C_Q와 C_B를 각각 구할 때 ANN 인덱스를 활용하되, 차집합 연산은 DB 엔진의 집합 연산 모듈이 담당합니다.

이처럼 텍스트 필터(전통 인덱스)와 벡터 필터(ANN 인덱스), 그리고 집합 연산(교집합/차집합)이 유기적으로 결합되어 NOT 연산을 효과적으로 처리할 수 있습니다.

---

5. 추가 기술적 상세

1. **인덱스 구현**:
   - 벡터 인덱스: HNSW, IVF, 또는 PQ 기반 구조를 사용해 C_Q, C_B 같은 Top-k 리스트를 생성.
   - 전통 인덱스: B-Tree(정렬 키), Bitmap(이진 속성 등) 등을 사용해 빠른 필터링.
2. **집합 연산 최적화**:
   - DB 엔진은 C_Q와 C_B가 각각 얼마만큼 큰지, 그리고 공집합일 가능성이 얼마나 되는지를 미리 추정(통계 기반)하여 최적의 연산 순서를 결정합니다.
   - 예: $|C_B|$가 작다면, 먼저 C_B를 빠르게 구하고 나서 C_Q에서 빼는(차집합) 방식이 이점이 있을 수 있습니다.
   - 반대로, C_B가 매우 클 경우에는 C_Q가 먼저 구해진 뒤, 교집합 영역만큼만 C_B를 선별적으로 계산할 수도 있습니다.
3. **복합 쿼리**:
   - NOT 연산은 물론, AND, OR 등의 논리 연산도 동시에 지원하므로, 메타데이터 + 벡터 검색 + 부정 조건이 어우러진 복합 쿼리를 한 번에 처리할 수 있습니다.

6. 결론

- 벡터 검색에서 NOT 연산은 기본적으로 거리나 유사도를 '낮게' 유지해야 하는 조건이기 때문에, "가장 가까운 이웃"을 찾는 ANN 구조와는 어긋나기 쉽습니다.
- 기존 방식(후처리, 쿼리 벡터 조정 등)은 직관적이지 않고, 성능 저하나 부정확성 문제가 생깁니다.
- **Aeca Database**는 **하이브리드 인덱스 구조**와 **네이티브 집합 연산(교집합, 차집합) 지원**을 통해, 벡터 검색에서의 NOT 연산을 직접 지원합니다.
  - C_Q와 C_B를 구분하여, $C_Q \setminus C_B$를 빠르게 계산.
  - 다중 필터(AND, OR, NOT 등)를 단일 쿼리 계획 내에서 효율적으로 처리.
  - 내부 쿼리 플래너가 집합의 크기, 통계 정보를 고려해 최적의 순서로 연산.

이러한 접근 방식은 벡터 검색의 활용성을 확장하며, 보다 정교한 의미 기반 검색을 가능하게 합니다. 또한, 키워드와 벡터 연산이 결합된 **하이브리드 검색** 환경에서 유용하게 활용될 수 있습니다. 앞으로 벡터 검색 엔진들이 다양하고 복합적인 질의를 처리할 수 있는 방향으로 진화함에 따라, Aeca Database와 같은 접근법이 점차 중요해질 것으로 예상됩니다.

참고 자료

- Facebook AI Research, Faiss
- Spotify, Annoy
- Yury Malkov et al., HNSW
- Hybrid Indexing 및 Query Planner 관련 DB 연구 자료