신형진

Github: https://github.com/hjinshin
Email: gudwls818@gmail.com

Backend Developer

Stack: Java, Spring Boot, MySQL, Redis

Core Value

[1] 사용자 경험과 안정성을 고려한 시스템 설계를 지향합니다.
단순히 기능을 구현하는 데 그치지 않고, 실제 사용자가 느끼는 경험과 서비스의 안정성을 최우선으로 생각합니다.
안정적이면서도 효율적인 구조를 고민하며, 기술적 개선을 통해 더 나은 사용자 경험을 만들어가기 위해 노력합니다.

[2] 상황에 맞는 합리적인 의사결정을 하기 위해 노력합니다.
팀과 서비스의 목표에 부합하는 합리적인 의사결정을 내리기 위해 노력합니다.
이를 위해 테스트, 가설 설정, DB 실행 계획 등 객관적인 데이터를 기반으로 상황에 맞는 최선의 선택을 고민합니다.

[3] 개인의 성장이 조직의 성장에 기여한다고 생각합니다.
개인의 성장이 곧 조직의 성장으로 이어진다고 믿습니다.
그래서 스스로 역량을 개발하고, 이를 팀과 공유하며 함께 성장하는 환경을 만들어가는 것을 중요하게 생각합니다.
꾸준한 학습과 지식 공유를 통해 조직 전체의 역량을 높이고, 팀 성과에 실질적으로 기여고자 합니다.

Experience

Whokie (Github) / Backend Developer

칭찬으로 나를 알아가는 소셜 미디어 서비스 / 2024.09 - 2025.04
Java, Spring Boot, MySQL, Redis

- 조회수 집계 로직의 응답 속도를 높이고, 데이터베이스 부하를 줄이기 위해 Redis 기반의 Write-Back 캐싱 구조 도입 (p.3)
- 동시 요청을 낙관적 락으로 제어하는 과정에서 발생한 데드락을 MySQL 상태정보 로그를 통해 분석하고, 외래키 제약 조건을 제거하여 해결 (p. 4-5)
- 정렬 연산을 생략하고, Index Range Scan으로 데이터를 조회하기 위해 복합 인덱스 추가 (p.6)
- JWT 인증·인가 처리와 이를 위한 커스텀 리졸버·인터셉터 구현
- nGrinder 기반 부하 테스트를 통해 프로필 조회·메시지 목록 API 병목 지점을 찾아 응답 시간을 약 60% 개선
- 카카오 테크 캠퍼스 신규 서비스 개발 프로젝트 최우수상(2위) 수상

BetterGPT(Github/Demo) / Desktop Application Developer

GPT API와 음성인식을 활용한 데스크탑 기반 GUI 애플리케이션/ 2023.03 - 2023.06
Python, LangChain, PyQt6, VectorDB(Chroma DB)

- 로컬에 파일을 벡터화해 VectorDB에 저장하고, 의미 기반 유사도 검색을 통한 효율적인 사용자 맞춤형 검색 파이프라인 구축 (p.7-8)
- LangChain을 활용해 API 호출 흐름을 추상화함으로써 특정 API에 종속되지 않는 애플리케이션 구조 설계
- PyQt6 기반 오버레이 GUI를 구현해 멀티태스킹 환경에서도 비침투적인 UX 제공
- Google Speech-to-Text API와 ChatGPT API를 결합해 실시간 음성 질의응답 기능 개발

크누버스(Github/Demo) / Backend Developer

경북대학교 강의 시간표 및 강의 계획서 서비스 / 2024.03 - 2024.06
Java, Spring Boot, MySQL

- 사용자 시간표 저장 및 조회 API 구현
- 강의 목록 필터링 및 검색 API 구현
- 강의 데이터 자동 수집 및 저장 파이프라인 구성

Education

경북대학교 / 컴퓨터학부 졸업

2019.03 - 2025.02

카카오 테크 캠퍼스 2기 / 수료

2024.04 - 2024.11

- Java/Spring 기반 웹 서비스 개발(Backend)

Certification

- 정보처리기사

Awards

- 제 5회 POSTECH OIBC Challenge - 태양광 발전량 예측 경진대회(2023.12) 대상
- 카카오 테크 캠퍼스 - 신규 서비스 개발 프로젝트(2024.11) 최우수상

Redis 기반의 Write-Back 캐싱 구조 도입

[Situation] (상황)

사용자의 프로필 페이지를 조회할 때마다 해당 사용자가 신규 방문자인지 확인하고, 신규 방문자일 경우 조회수를 증가시키는 로직이 존재했습니다.
그러나 조회수 데이터와 일일 방문자 정보를 모두 관계형 데이터베이스에 저장하고 있어, 단순 조회 요청에도 매번 DB 접근이 발생했습니다.

문제점

- 데이터베이스 접근은 메모리 기반 연산에 비해 상대적으로 **느림**
- **Connection Pool이 한정**되어 있어 요청이 몰릴 경우 대기 시간 증가
- 잦은 접근은 **데이터베이스 부하**를 가중시켜 전체 서비스 성능 저하로 이어질 위험이 있음

아직 사용자 수가 크게 늘지 않았음에도 이러한 구조적 한계가 드러났고, 향후 확장을 고려한 개선이 필요했습니다.

[Task] (업무)

1. 데이터베이스 접근 횟수를 줄여 **성능 저하와 DB 부하를 완화할 것**
2. 조회수 데이터와 일일 방문자 정보의 **영속성을 보장할 것**

[Action] (행동)

이 문제를 해결하기 위해 팀 내에서 다양한 방안을 논의했습니다. 저는 실시간성과 동시성 제어 측면에서 Redis 활용이 적합하다고 보았지만, 일부 팀원은 데이터 영속성을 보장할 수 없다는 이유로 반대했습니다.

서비스 특성과 실제 트래픽 상황을 근거로 분석하며 논의를 이어갔고, 최종적으로 Redis에서 조회수를 실시간으로 처리해 성능을 확보하고, 스케줄러를 통해 주기적으로 DB에 반영하는 **Write-Back 캐싱 전략**을 도입하기로 합의했습니다.

이때 스케줄러의 동기화 주기는 1시간으로 설정했습니다.
-> Flick라는 마케팅 플랫폼 조사에 따르면 1인당 월평균 프로필 조회수는 약 440회, 즉 시간당 0.6회 수준이므로 1시간에 1번만 반영해도 충분하다고 판단했습니다.

[Result] (결과)

VUsers 198로 5분간 부하 테스트를 진행한 결과, 평균 응답 대기 시간이 약 60% 감소하며 성능이 크게 개선되었습니다.

또한 조회수 데이터와 일일 방문자 정보를 1시간 단위로 데이터베이스에 반영함으로써 성능 뿐만 아니라, 데이터의 영속성을 일정 수준 보장할 수 있었습니다.

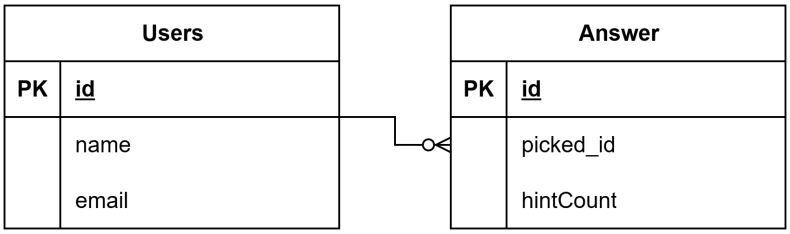
이로써 성능과 안정성의 균형을 맞추면서, 향후 사용자 증가에도 대응할 수 있는 확장성 있는 구조를 마련할 수 있었습니다.

데드락을 해결하기 위한 외래키 제약 조건 제거

[Situation] (상황)

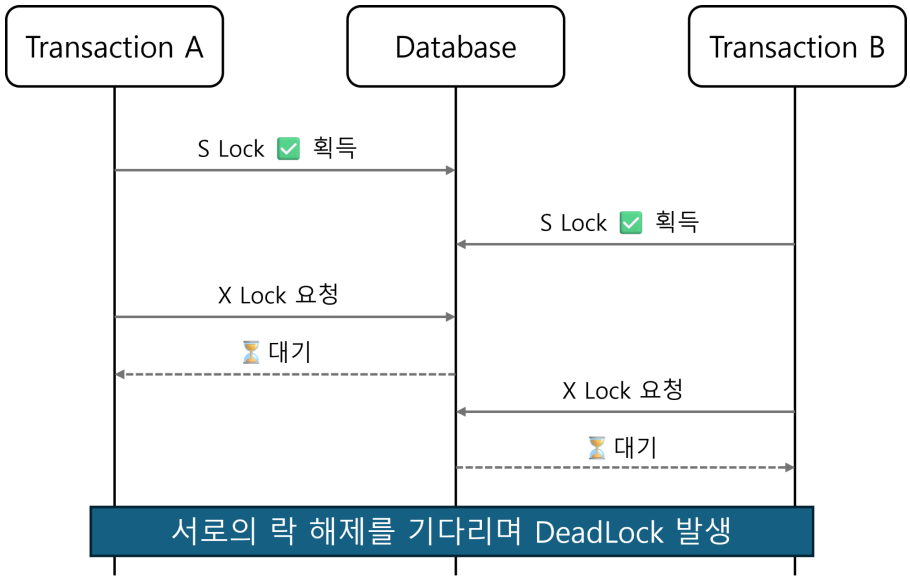
서비스 운영 중, 사용자가 동시에 여러 칭찬 메시지를 받을 때 메시지 수신이 간헐적으로 실패하는 문제가 발생했습니다.

(User - Answer 관계)



MySQL InnoDB 로그를 분석한 결과, User 엔티티에 대해 S락을 선점한 상태에서 X락을 요청하면서 **데드락**이 발생한 것을 확인했습니다. 다만, 코드상에서 포인트를 증가시킬 때 X락을 요청한다는 사실은 명확했지만, S락이 걸린 원인은 쉽게 파악하기 어려웠습니다.

(트랜잭션 시퀀스 플로우)



[Task] (업무)

데드락의 정확한 원인을 규명하고, 이를 해결하여 **서비스 안정성을 확보**하는 것이 목표였습니다.
동시에 여러 메시지를 받더라도 **포인트가 정상적으로 증가하도록 동시성을 제어**하는 것도 중요한 과제였습니다.

[Action] (행동)

MySQL 공식 문서를 검토한 결과, **외래키 제약 조건 검사 과정에서 부모 또는 자식 레코드에 S락이 걸릴 수 있음**을 확인했습니다.
즉, Answer(칭찬 메시지) 엔티티를 저장할 때 User와의 외래키 무결성 검사가 수행되면서 S락이 걸렸고, 이로 인해 데드락이 발생했던 것입니다.

문제 해결을 위해 트랜잭션에 비관적 락을 적용하는 방법도 고려했습니다.

- 장점: User 엔티티에 대한 **접근을 직렬화하여 데드락 방지 가능**
- 단점: **트랜잭션 직렬화**로 인해 **처리량 저하 및 지연 시간 증가**

성능 저하가 크다는 단점 때문에 비관적 락은 배제했습니다. 대신, **낙관적 락으로 동시성을 제어**하면서 **외래키 제약을 제거**하는 방식을 선택했습니다.
User–Answer 관계에서 외래키 값이 변경되지 않는 구조였기 때문에, **외래키 제약 없이도 데이터 무결성을 보장할 수 있다고 판단**했습니다.

[Result] (결과)

낙관적 락과 재시도(Retryable) 로직을 통해 동시에 여러 메시지를 수신하더라도 포인트가 정상적으로 증가하는 것을 확인했습니다.

또한 User–Answer 간 외래키 제약을 제거한 뒤에는, Answer 엔티티를 Insert할 때 User에 S락이 걸리지 않아 데드락이 재발하지 않음을 확인했습니다.

정렬 연산을 생략하기 위한 복합 인덱스 추가

[Situation] (상황)

부하 테스트를 진행하던 중, 사용자별 칭찬 목록을 페이징(pageable)으로 조회하는 과정에서 심각한 성능 저하가 발생했습니다.

```
EXPLAIN ANALYZE
SELECT * FROM answer
WHERE picked_id = 123
AND created_at BETWEEN '2024-03-01 00:00:00' AND '2025-03-19 23:59:59'
ORDER BY created_at DESC
LIMIT 10 offset 30;

-> Limit/Offset: 10/30 row(s)  (cost=1.05e+6 rows=10) (actual time=6882..6882)
   -> Sort: answer.created_at DESC, limit input to 40 row(s) per chunk
      -> Filter: ((answer.picked_id = 123) and (answer.created_at between 'start_date'
        and 'end_date' ...))
         -> Table scan on answer
```

DB 실행 계획을 확인한 결과, Answer 테이블 전체를 스캔한 뒤 created_at에 대해 정렬하는 과정이 병목으로 작용하고 있었습니다. 그 결과, 약 1,000만 건의 데이터를 기준으로 실행 시간이 약 7초가 소요되는 것을 확인했습니다.

[Task] (업무)

대규모 데이터 환경에서도 사용자별 칭찬 목록을 안정적으로 조회할 수 있도록 하는 것이 중요한 과제였습니다. 이를 위해 **불필요한 정렬 연산을 제거**하고, **쿼리를 최적화**해 **서비스의 응답 속도를 개선**할 필요가 있었습니다.

[Action] (행동)

DB 실행 계획에서 성능 저하의 원인이 정렬에 있음을 확인한 후, 이를 인덱스로 대체하기 위해 복합 인덱스를 설정했습니다.

- 동등 조건(picked_id)을 선두 컬럼으로, 정렬 컬럼(created_at)을 후행 컬럼으로 두어 **(picked_id, created_at)** 인덱스를 생성

이를 통해 동일한 사용자 그룹 내에서 생성일시를 기준으로 정렬이 자동으로 보장되도록 했습니다.

[Result] (결과)

```
-> Limit/Offset: 10/30 row(s) (cost=13.8 rows=0) (actual time=0.143..0.143)
   -> Index range scan on answer using idx_answer_picked_created
     (cost=13.8 rows=12) (actual time=0.04..0.141 rows=12 loops=1)
```

인덱스 적용 후 실행 계획에서 정렬 연산이 사라지고 Index Range Scan으로 최적화된 것을 확인했습니다.

그 결과, 동일한 조건에서 쿼리 실행 시간이 약 0.1ms로 단축되어, 대규모 데이터 환경에서도 빠르게 조회할 수 있게 되었습니다.

로컬 파일 VectorDB 저장 및 의미 기반 검색 파이프라인 구축

[Situation] (상황)

기존 ChatGPT는 웹 기반 서비스라서 사용자가 자신의 로컬 파일에 접근할 수 없다는 한계가 있었습니다. 이로 인해 개인 문서에서 원하는 정보를 직접 검색하거나 답변에 활용하기 어려웠습니다.

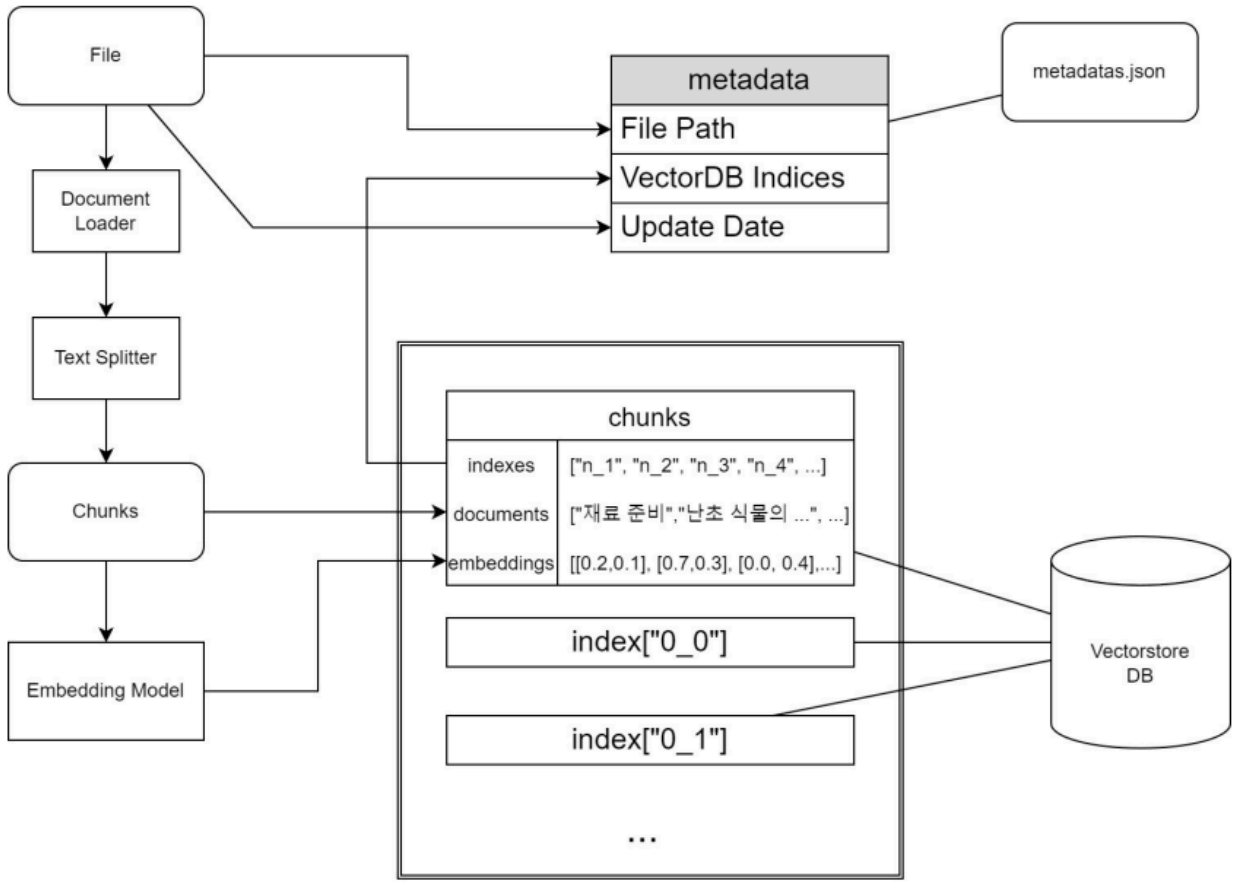
설령 로컬 파일에 접근이 가능하더라도 모든 파일 내용을 그대로 API에 전송하면 호출 비용이 과도하게 증가하는 문제가 있었습니다.

[Task] (업무)

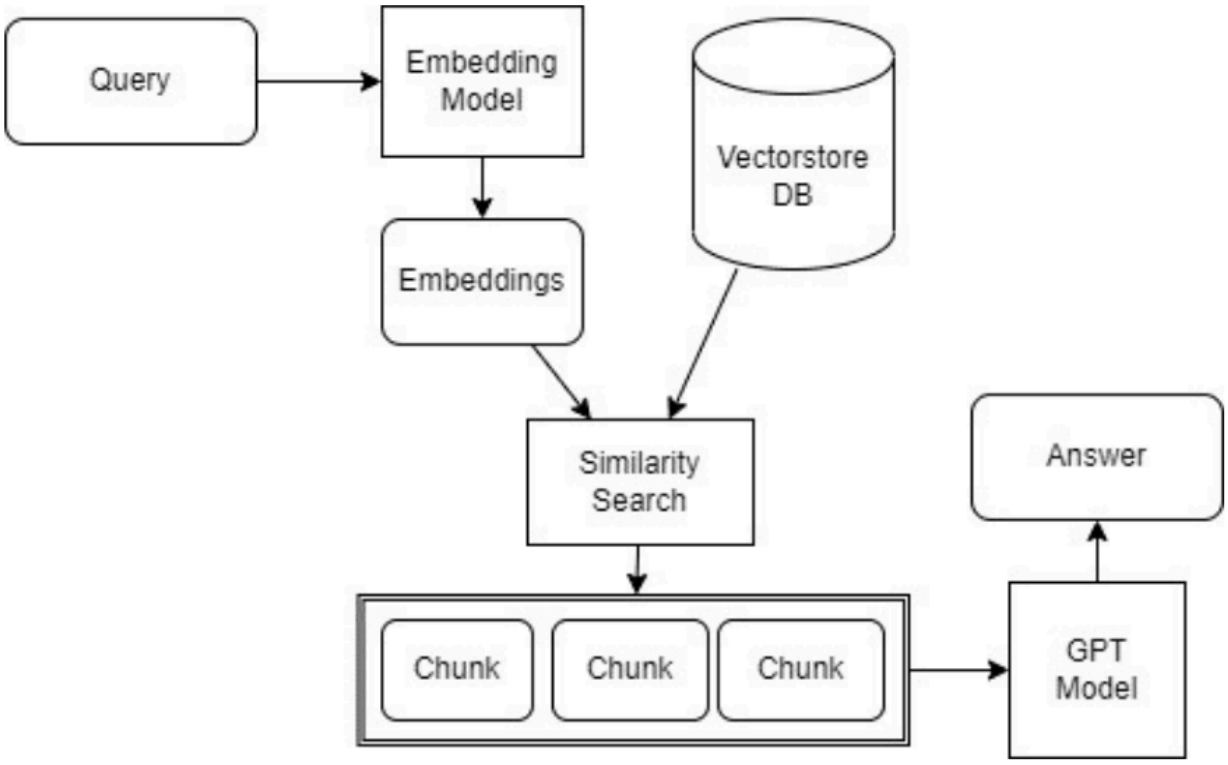
사용자가 자신의 로컬 문서까지 검색에 활용할 수 있도록 하고, 동시에 효율적인 방식으로 원하는 정보를 찾을 수 있도록 하는 파이프라인을 구축해야 했습니다.

[Action] (행동)

우선 로컬에 저장된 PDF, Word 파일에서 텍스트를 추출한 뒤, 청크 단위로 분할하여 임베딩을하고 이를 VectorDB에 저장했습니다.



이후 사용자가 쿼리를 입력하면 해당 쿼리를 벡터화하여 VectorDB와 비교하고, 유사도 검색을 통해 가장 관련성이 높은 청크만 추출하도록 했습니다.



이렇게 선별된 청크를 최초 사용자 쿼리와 함께 GPT API에 전달해, **로컬 문서의 문맥이 반영된 답변을 생성**할 수 있도록 파이프라인을 설계했습니다.

[Result] (결과)

이 파이프라인을 통해 모든 파일을 통째로 전달하지 않고, 유사 청크만 선별하여 GPT API로 전송함으로써 **API 호출 비용을 크게 절감**할 수 있었습니다.

또한 단순 키워드 매칭이 아닌 의미 기반 유사도 검색 덕분에 **정밀한 검색 결과를 제공**할 수 있었습니다.

결과적으로 기존 ChatGPT의 한계였던 **로컬 파일 접근 문제를 보완**하고, 비용 효율성과 정확성을 모두 갖춘 **사용자 맞춤형 검색 환경을 제공**할 수 있게 되었습니다.