N+1 Problem

태그

📌 목차

1. N+1 Problem
    1-1. 원인
2. 해결법
    2-1. Fetch Join + Lazy Loading
        1) Fetch Join
        2.1) Collection 연관관계 Fetch Join 시 주의사항
        2.2) Distinct절 사용해야 함
        2.3) Collection Fetch Join은 하나까지만 가능
        2.4) Paging 금지
    2-2. default_batch_fetch_size, @BatchSize
    2-3. 일반 Join 후 Projection하여 특정 칼럼만 DTO로 조회
    2-4. @EntityGraph
3. 변경 감지 쿼리에서 발생하는 문제
4. 결론

1. N+1 Problem

- **ORM**에서 발생하는 성능 이슈의 가장 흔한 원인
- 연관 관계에서 발생하는 이슈로, 연관 관계가 설정된 엔티티를 한 번 조회할 때 조회된 데이터 개수(N)만큼 연관 관계의 조회 쿼리가 추가로 발생하는 문제

📌 **ORM (Object Relational Mapping)**

- 객체와 관계형 DB의 데이터를 자동으로 매핑해주는 것
- 객체 지향 프로그래밍은 **클래스**를, 관계형 DB는 **테이블**을 사용
- 객체 모델과 관계형 모델 간에 불일치 존재
- ORM을 통해 객체 간의 관계를 바탕으로 SQL을 자동 생성하여 불일치 해결
- Persistence API라고도 할 수 있음

📌 **JPA (Java Persistence API)**

- 자바 진영의 ORM 기술 표준
- 인터페이스의 모음
- JPA 인터페이스를 구현한 대표적인 오픈소스가 Hibernate
- 장점
  - 객체 중심 개발
  - 생산성 (간단한 CRUD)
  - 유지보수
  - Object와 RDB간 패러다임 불일치 해결

1-1. 원인

- 관계형 DB와 객체지향 언어간의 패러다임 차이로 인해 발생
- 객체는 연관관계를 통해 Reference를 가지고 있으면 언제든지 메모리 내에서 Random Access를 통해 연관 관계에 접근할 수 있지만, **RDB의 경우 SELECT 쿼리를 통해서만 조회할 수 있기 때문**
- 예시:

```
@Entity
public class Article extends BaseEntity{
    @OneToMany(mappedBy = "article", fetch = FetchType.LAZ
```

```
    private List<Opinion> opinions = new ArrayList<>();
}
```

- Article 1회 조회시 opinions 리스트 자리에 프록시 객체가 생성되고, 이를 코드 내에서 조회하는 순간 N개의 SELECT 쿼리 발생
- 즉, 각각의 Opinion마다 SELECT 쿼리 발생

> 📌 **프록시 객체 (Proxy Entity)**
>
> - 실제 엔티티를 필요할 때만 꺼내 쓸 수 있도록 하는 가짜 객체
> - 실제 클래스를 상속받아 만들어지며, 겉모양은 같아 Proxy 객체를 그대로 써도 무방
> - 실제로 필요한 순간까지 쿼리를 실행하지 않음 (내부 target에 값이 없음)

2. 해결법

2-1. Fetch Join + Lazy Loading

- Root Entity에 대해서 조회할 때 Join 쿼리를 발생시켜 Lazy Loading으로 설정되어 있는 연관관계를 한번에 조회할 수 있는 기능
- 예시:

```
@Query("select Distinct a from Article a join fetch a.opin
List<Article> findAllArticleFetchJoinOpinion();
```

1) Fetch Join

- ORM에서의 사용을 전제로 DB Schema를 Entity로 자동 변환 해주고, 영속성 컨텍스트에 영속화함
  - 때문에 연관 관계가 영속성 컨텍스트 1차 캐시에 저장되어 다시 Entity 그래프를 탐색하더라도 조회 쿼리가 수행되지 않음
  - 반면 일반 Join 쿼리는 데이터를 단순 조회하므로 영속성 컨텍스트나 Entity와 무관

- 따라서 가능하면 Fetch Join을 활용해야 ORM을 활용하여 RDB와의 패러다임 차이를 줄일 수 있음

> 📌 **JPA 영속성 컨텍스트**
>
> - Entity를 영구 저장하는 환경
> - Application과 DB 사이에서 객체를 보관하는 가상의 DB 역할
> - EntityManager를 통해 Entity를 저장/조회 시 EntityManager는 이를 영속성 컨텍스트에 보관 및 관리함
> - `em.persist(member);`
>   - 영속성 컨텍스트에 저장하는 메소드
> - 특징
>   - EntityManger를 생성할 때 하나 생성
>   - EntityManager를 통해 영속성 컨텍스트에 접근 및 관리 가능

2.1) Collection 연관관계 Fetch Join 시 주의사항

- Fetch Join을 Collection에 대해서 할 경우 SQL Native Join 쿼리 발생
- 이 경우 1:N 관계이므로 1쪽의 데이터는 중복된 상태로 조회
  - 다음 레코드 형태로 데이터를 받아오고, 이를 객체로 매핑함

레코드	Article	Opinion
1	Article1	Opinion1
2	Article1	Opinion2
3	Article2	Opinion3
4	Article2	Opinion4

2.2) Distinct절 사용해야 함

- 위 표에서 처럼, 1:N 관계에서는 중복 데이터가 존재
- 이를 방지하기 위해 Distinct절 활용 (**JPQL**상의 Distinct로, SQL에서의 Distinct와 차이 존재)

- **SQL Distinct**: DB에서 수행되며, Join되어 발생한 각 row를 비교해서 다른 경우만 남김
- **JPQL Distinct**: 조회 대상 Entity(Select 쿼리 바로 다음에 오는 Entity 객체)에 대해서 Distinct 수행

2.3) Collection Fetch Join은 하나까지만 가능

- 여러 Collection에 대해 Fetch Join 시 잘못된 결과 발생 가능
- 반드시 하나 까지만 Fetch Join 해야 함

2.4) Paging 금지

- Out of Memory 발생 가능
- 테이블의 모든 데이터를 메모리로 로딩해야 하기 때문

2-2. default_batch_fetch_size, @BatchSize

- Lazy Loading 시 프록시 객체를 조회할 때 `where in`절로 묶어 한번에 조회할 수 있게 해주는 옵션
- yml에 전역 옵션으로 적용하거나 `@BatchSize`를 통해 연관관계 BatchSize를 다르게 적용할 수 있음
- BatchSize는 100~1000정도로 적용하고, DBMS에 따라 `where in`절은 1000까지 제한하는 경우가 있으므로 1000 이상은 잘 설정하지 않음
- WAS는 BatchSize가 커도 어차피 데이터를 메모리에 로딩해야 해서 괜찮지만, DB에는 부담이 갈 수 있어 적절히 조절해야 함

2-3. 일반 Join 후 Projection하여 특정 칼럼만 DTO로 조회

```
SELECT new 패키지 경로.ArticleDto(원하는 필드)
FROM Article ar
    JOIN ar.opinions op ON op.article_id = ar.id
```

- 장점
  - Entity Column이 많을 때 Projection하여 특정 칼럼만 조회 가능
  - 커버링 인덱스 활용 가능성 증가
- 단점

    - 영속성 컨텍스트와 무관하게 동작
    - Repository가 DTO에 의존하므로 API 변경에 따른 DAO 수정 가능성 존재
- 이 방식을 사용하는 쿼리는 DAO를 분리하는 것이 좋음

2-4. @EntityGraph

- Lazy Loading을 Eager Loading으로 부분적으로 전환하는 기능
- 장점
    - 여러 1:N 연관관계를 한번에 Join할 수 있음
    - Fetch Join의 경우 1개의 Collection까지만 같이 Join하여 조회 가능

> ⚠️ **Eager Loading은 X!**
>
> - Entity 객체를 한번에 조회하도록 하는 기능
> - 특정 경우에는 N+1 문제를 부분적으로 해결해 줄 수 있지만, 비권장
> - 어떤 Entity 연관관계 범위까지 Join 쿼리로 조회해올 지 예상하기 힘들어 지기 때문에 필요 없는 데이터까지 로딩하여 비효율적일 수 있음
> - 따라서 주인이 아닌 OneToOne 관계를 제외하고는 모두 **Lazy Loading**으로 설정해야 함

3. 변경 감지 쿼리에서 발생하는 문제

> ☝🏻 N+1 문제는 아님!

- 변경 감지를 통해 여러 Entity를 수정하는 경우 쿼리가 Entity 개수만큼 발생하는 문제가 있음
- 이 경우에는 Update 쿼리를 직접 작성해야 함
- 단, 이는 영속성 컨텍스트와 무관하므로 EntityManager의 flush()나 clear()를 호출하는 것이 좋음

4. 결론

- OneToOne 관계는 최대한 Fetch Join 활용
- Collection 관계는 default_batch_fetch_size 활용
- 많은 칼럼 중 특정 칼럼만 조회해야 할 경우나 커버링 인덱스를 활용하고 싶은 경우에 데이터 전송량을 줄이고 싶으면 일반 Join 하고 Projection하여 DTO로 바로 변환
    - DAO 분리 권장