N+1 Problem

1. N+1 문제란?

N+1 문제는 연관관계가 설정된 엔티티 사이에서 한 엔티티를 조회하였을 때, 조회된 엔티티의 개수(N개)만큼 연관된 엔티티를 조회하기 위해추가적인 쿼리가 발생하는 문제를 의미합니다.

즉, N+1에서 1은 한 엔티티를 조회하기 위한 쿼리의 개수이며, N은 조회된 엔티티의 개수만큼 연관된 데이터를 조회하기 위한 추가적인 쿼리의 개수를 의미한다.

2. 발생하는 상황

하나의 게시글에는 여러 개의 댓글이 달릴 수 있으며, 이를 코드로 나타내면 다음과 같다.
< Posts.java >

```
@Entity
public class Post {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String title;
        private String content;

        @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemo
val = true)
        private List<Comment> comments = new ArrayList<>();

        // 생성자, Getter 생략
        public Comment writeComment(final String content){
                Comment comment = new Comment(conten, this);
                this.comments.add(comment);
                return comment;
        }
}
```

< Comment.java >

```
@Entity
public class Comment {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Lond id;
        private String content;

        @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "post_id")
        private Post post;

        // 생성자, Getter 생략
}
```

< N+1 문제가 발생하는 테스트 코드 >

```
@SpringBootTest
@Transactional
class PostRepositoryTest {

        @Autowired
        private EntityManager em;

        @Autowired
        private PostRepository postRepository;

        @Test
        @DisplayName("N+1 발생 테스트")
        void test {
                saveSampleData();
                em.flush();
                em.clear();
                System.out.println("-------------영속성 컨텍스트 비우기--------
---\n");


                System.out.println("-------------POST 전체 조회 요청 --------
------");
                List<Post> posts = postRepository.findAll);
                System.out.println("-------------POST 전체 조회 완료 [1번의 쿼
리 발생 ---");


                Systsem.out.println("------------POST 제목, 내용 조회 요청 ---
```

```
-----");
                posts.forEach(it -> System.out.println("POST 제목: [%s], POS
T 내용: [%s]".formatted(it.title, it.content));
                System.out.println("-------------POST 제목, 내용 조회 완료. [추
가적인 쿼리 발생하지 않음]-----");


                System.out.println("-------------POST에 달린 comment 내용 조회
요청 [조회된 POST의 개수 N만큼 추가적인 쿼리 발생]-----");
                posts.forEach(post ->
                        post.comments().forEach(comment -> {
                                System.out.println("POST 제목: [%s], COMMENT
내용: [%s]".formatted(comment.post.title, comment.content));
                        });
                });
                System.out.println("------------POST에 달린 comment 내용 조회
완료-------");
        }



        private void saveSampleData {  // 10개의 post에 각 3개의 댓글 저장
                final String postTitleFormat = "[%d] post-title";
                final String postContetnFormat = "[%d] post-content";
                final String commentContentFormat = "[%d] comment-content";

                IntStream.rangeClosed(1, 10).forEach(i -> {
                        Post post = new Post(format(postTitleFormat, i), fo
rmat(postContentFormat, i));

                        IntStream.range(1, 3).forEach(j -> {
                                post.writeComment(format(commentContentForm
at, j));
                        });

                        postRepository.save(post);
                });
        }
}
```

→ 위 코드에서 POST에 달린 Comment 내용 조회 시, N+1 개의 쿼리가 실행됨

3. 지연 로딩과 즉시 로딩

`@ManyToOne`, `@OneToOne` 은 기본적으로 **즉시 로딩**으로 설정되어 있고, `@ManyToMany`, `@OneToMany` 는 기본적으로 **지연 로딩**으로 설정되어 있다.

> 💡 N+1문제는 지연로딩(FetchType.LAZY)과 즉시로딩(FetchType.EAGER) 모든 경우에 발생할 수 있다.
>
> 위의 예제도 즉시 로딩으로 설정한다면, 쿼리가 발생하는 상황만 달라질 뿐, N+1 번의 쿼리가 발생하는 것은 동일하다.

→ 즉시로딩으로 Post 클래스 변경 시,
< Posts.java >

```
@Entity
public class Post {
        // 필드 생략

        @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemo
val = true)
        private List<Comment> comments = new ArrayList<>();

        // 생성자, Getter 생략, 메서드 생략

}
```

→ POST 전체 조회 요청 시에 즉시 로딩으로 인해 조회된 POST의 개수 N 만큼 추가적인 쿼리 발생, POST에 달린 comment 내용 조회 요청 시에도 N + 1개의 쿼리 발생

→ 즉, 동일한 코드에서 쿼리가 발생하는 시점만 달라질 뿐, 전체적인 쿼리의 수는 동일하다.

4. 해결 방법

→ 해결 시 모든 경우에 일단 **지연 로딩**을 사용한다.

4-1. @OneToOne 관계에서 발생하는 N+1

💡 **fetch join, @EntityGraph**로 해결 가능하다.

※ 주의) @OneToOne 연관관계에서는 연관관계의 주인이 아닌 엔티티를 조회하는 경우, 지연 로딩으로 설정하였더라도, 연관된 엔티티를 즉시 로딩으로 조회한다.

```
Member 객체와 Locker 클래스 필드를 OneToOne 관계로 연결하려 할 때,
JPA에서는 Member 객체의 Locker 필드의 프록시를 만들기 위해서는, DB에 그 값이 있는지 없는지 확인해야 합니다. (있으면 프록시를 넣고, 없으면 null을 넣습니다)

그러나 위의 관계에서는 LOCKER 테이블에 FK가 있으므로, 값을 확인하기 위해서는 LOCKER 테이블을 조회해야 하는것입니다.
 즉 어차피 쿼리가 발생하기 때문에, 굳이 프록시로 만들 이유가 없는 것입니다. 따라서, 지연 로딩으로 설정하더라도 즉시 로딩됩니다.
```

- 먼저 문제 상황 (N+1) 을 살펴보자

< Member.java >

```
@Entity
public class Member {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String name;

        @OneToOne(mappedBy = "member", fetch = FetchType.LAZY)
        private Locker locker;
}
```

< Locker.java >

```
@Entity
public class Locker {
        @Id
        @GenerateValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private int number;
```

```
        @OneToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "member_id")
        private Member member;

        public void register(Member member){
                this.member = member;
                member.setLocker(this);
        }
}
```

< 문제 상황 >

```
@SpringBootTest
@Transactional
class LockerRepositoryTest {

        @Autowired
        private EntityManager em;

        @Autowired
        private MemberRepository memberRepository;

        @Autowired
        private LockerRepository lockerRepoistory;

         @BeforeEach
    public void init() {
        IntStream.rangeClosed(1, 3).forEach(i -> {
            Member member = new Member("[%d] MEMBER".formatted(i));
            Locker locker = new Locker(i);
            locker.register(member);
            memberRepository.save(member);
            lockerRepository.save(locker);
        });

        em.flush();
        em.clear();
    }

        @Test
        @DisplayName("OneToOne 주인으로 조회하기")
        void test1 {
                System.out.prinln("LOCKER 조회");
                List<Locker> lockers = lockerRepository.findAll(); // 쿼리 1
번 - 3개의 Locker 조회
                System.out.println("Locker 조회 완료\n");
```

```
                lockers.forEach(it -> {
                        // 연관관계의 주인이 아닌 엔티티는 무조건 즉시 로딩되기 때문에
쿼리가 추가로 발생한다, 여기서 주인은 Locker
                        // (1[Member 조회] + 1[Locker 조회]) * 3[Locker 개수]
= 총 6번의 쿼리
                        System.out.println("[%d]번 LOCKER 사용하는 회원 이름:
[%s] \n".formatted(it.name, it.member.name));
                });
        }

        @Test
        @DisplayName("OneToOne 주인 아닌 엔티티로 조회하기 - 지연 로딩으로 설정하더라
도 즉시 로딩된다");
        void test2 {
                System.out.println("MEMBER 조회");
                // 1[회원 전체 조회] + 3[조회된 회원의 수만큼 즉시 로딩으로 인한 쿼리
발생] = 총 4번의 쿼리
                List<Member> members = memberRepository.findAll();
                System.out.println("MEMBER 조회 완료\n\n");

                members.forEach(it -> {
                        System.out.println("[%d]번 LOCKER 사용하는 회원 이름:
[%s]\n".formatted(it.locker.number, it.name));
                });

        }
}
```

<문제 해결>

1. **fetch join 사용**

: 다음과 같이 @Query를 통해 fetch join을 사용하여 해결해보자

```
public interface MemberRepository extends JpaRepository<Member, Long>{
        @Override
        @Query("select m from Member m join fetch m.locker")
        List<Member> findAll;
}
```

```
public interface LockerRepository extends JpaRepository<Locker, Long> {
        @Override
        @Query("select l from Locker l join fetch l.member")
        List<Locker> findAll;
}
```

→ 한번의 쿼리로 문제 해결!!

2. **@EntityGraph 사용**

: @EntityGraph는 fetch join을 편하게 사용하도록 도와주는 기능이다.

```
public interface MemberRepository extends JpaRepository<Member, Long>{
        @Override
        //@Query("select m from Member m join fetch m.locker")
        @EntityGraph(attributePaths = {"member"})
        List<Member> findAll;
}
```

```
public interface LockerRepository extends JpaRepository<Locker, Long> {
        @Override
        // @Query("select l from Locker l join fetch l.member")
        @EntityGraph(attributePaths = {"locker"})
        List<Locker> findAll;
}
```

4-2. @ManyToOne 관계로 연관된 엔티티가 조회되는 경우

💡 fetch join, @EntityGraph로 해결 가능하다.
쿼리가 한번 더 발생하지만 @BatchSize로도 해결 가능하다.

1. **fetch join 사용**

```
public interface CommentRepository extends<Comment, Long> {
        @Overrider
        @Query("select c from Comment c join fetch c.post")
        List<Comment> findaAll();
}
```

2. **@EntityGraph 사용**

```
public interface CommentRepository extends JpaRepository<Comment, Long> {
        // @Query("selec c from Comment c join fetch c.post")
        @Overrider
        @EntityGraph(attributePaths = {"post"})
```

```
        List<Comment> findAll();
}
```

3. **@BatchSize**

   : 이는 쿼리 1번으로 해결되지 않고, 2번으로 나누어 해결한다.

```
@Entity
@BatchSize(size=100)
public class Post {
        // 생략
}
```

위와 같이 설정하거나, 혹은 application.yml 파일에서 다음 속성을 설정하여 적용할 수 있다.

```
spring.jpa.properties.hibernate.default_batch_fetch_size = 100
```

위와 같이 comment의 id를 IN절에 넣어 한번에 가져오는 것을 알 수 있다.

이때 IN절에 한번에 들어가는 크기를 BatchSize의 옵션으로 설정할 수 있다.

4-3. @OneToMany 관계로 연관된 엔티티가 조회되는 경우

> 💡 fetch join, @EntityGraph로 해결 가능하다.
> (그러나 페이징은 진행할 수 없으며, 둘 이상의 컬렉션을 페치 조인하는 데이터가 부정합하게 조회되기 때문에 이를 사용하지 않는 것이 좋다.)
>
> OneToMany 관계에서는 @BatchSize 혹은 @Fetch(FetchMode.SUBSELECT)로 해결한다.

1. **fetch join 사용 - 권장하지 않음**

```
public interface PostRepository extends JpaRepository<Post, Long> {
        @Override
        @Query("select p from Post p join fetch p.comments")
        List<Post> findAll();
}
```

→ 이 경우 일대다 조인을 했기 때문에 결과가 늘어나서 중복된 결과가 나타날 수 있었지만, 하이버네이트 6이후부터는 자동으로 distinct를 해주기 때문에 문제가 발생하지 않는다.

2. **@EntityGraph 사용 - 권장하지 않음**

```
public interface PostRepository extends JpaRepository<Post, Long> {

    //@Query("select p from Post p join fetch p.comments")
    @Override
    @EntityGraph(attributePaths = {"comments"})
    List<Post> findAll();
}
```

→ 위와 동일한 쿼리 발생

3. **@BatchSize - 권장**

```
spring.jpa.properties.hibernate.default_batch_fetch_size = 100
```

또는

```
@Entity
public class Post {
        @BatchSize(size=100)
        @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemo
val = true)
        private List<Comment> comments = new ArrayList<>();
}
```

4. **@ Fetch (FetchMode.SUBSELECT)**

```
@Entity
public class Post {

    @Fetch(FetchMode.SUBSELECT)
    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval
= true)
    private List<Comment> comments = new ArrayList<>();
}
```

5. N + 1 기본 해결 방법 정리

💡 @XToOne의 경우 : fetch join을 통해 해결 (혹은 @EntityGraph)
@XToMany의 경우 : BatchSize를 통해 해결 (혹은 @Fetch(FetchMode.SUBSELECT))