**Hibernate N+1 problem – 2022**

Hibernate N+1 problem occurs when you use FetchType.LAZY for your entity associations. If you perform a query to select n-entities and if you try to call any access method of your entity's lazy association, Hibernate will perform n-additional queries to load lazily fetched objects.

For example, we have the following Author entity with one-to-many books collection:

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

private String fullName;

@OneToMany(fetch = FetchType.LAZY)

private Set<Book> books;

}

Let’s try to load all authors and print each author’s name with his books collection size:

entityManager.createQuery("select a from Author a", Author.class)

.getResultList()

.forEach(a -> System.out.printf("%s had written %d books\n",

a.getFullName(), a.getBooks().size()));

The first query Hibernate will generate is to select all authors:

SELECT author0\_.id AS id1\_0\_,

author0\_.fullName AS fullname2\_0\_

FROM authors author0\_;

After that, when we call size() method on the books collection, this association needs to be initialized, so Hibernate will perform an additional query:

SELECT books0\_.author\_id AS author\_i4\_1\_0\_,

books0\_.id AS id1\_1\_0\_,

books0\_.id AS id1\_1\_1\_,

books0\_.author\_id AS author\_i4\_1\_1\_,

books0\_.title AS title2\_1\_1\_,

books0\_.year AS year3\_1\_1\_

FROM books books0\_

WHERE books0\_.author\_id=?;

This query will be called n-times for each author when we print the amount of books in addition to the first query. Thus the total number of queries will be equal to N+1.

**Hibernate provides a couple of ways to eliminate this issue**

1. **The first solution is to use join fetch**

**entityManager.createQuery**("select a from Author a left join fetch a.books", Author.class);

SELECT author0\_.id AS id1\_0\_0\_,

books1\_.id AS id1\_1\_1\_,

author0\_.fullName AS fullname2\_0\_0\_,

books1\_.author\_id AS author\_i4\_1\_1\_,

books1\_.title AS title2\_1\_1\_,

books1\_.year AS year3\_1\_1\_,

books1\_.author\_id AS author\_i4\_1\_0\_\_,

books1\_.id AS id1\_1\_0\_\_

FROM authors author0\_

LEFT OUTER JOIN books books1\_ ON author0\_.id=books1\_.author\_id;

This query works fine, but it has one issue: it doesn’t allow us to use pagination because the limit will not be applied to the authors. If you specify query.setMaxResults(n), Hibernate will fetch all existing rows and do the pagination in the memory, significantly increasing memory consumption.

1. **Another way is to use @BatchSize on the lazy association**

public class Author {

…

@OneToMany(fetch = FetchType.LAZY, mappedBy = "author")

@BatchSize(size = 10)

private Set<Book> books;

}

Hibernate will create the first query to retrieve all authors:

SELECT author0\_.id AS id1\_0\_,

author0\_.fullName AS fullname2\_0\_

FROM authors author0\_;

In this case, we can easily perform the pagination on the authors. Then, when we call size() method on the books collection, Hibernate will perform this query:

/\* load one-to-many Author.books \*/

SELECT books0\_.author\_id AS author\_i4\_1\_1\_,

books0\_.id AS id1\_1\_1\_,

books0\_.id AS id1\_1\_0\_,

books0\_.author\_id AS author\_i4\_1\_0\_,

books0\_.title AS title2\_1\_0\_,

books0\_.year AS year3\_1\_0\_

FROM books books0\_

WHERE books0\_.author\_id in (?, ?, ?, ?, ?, ?, ?, ?, ?, ? /\*batch size\*/);

This query will be called N/M times, where N is the amount of authors and M is the specified batch size. Totally we will call N/M+1 queries.

1. **The third way is to use a sub query returning a list of author identifiers**

Hibernate provides this opportunity by setting @Fetch(FetchMode.SUBSELECT) on the lazy association:

public class Author {

…

@OneToMany(fetch = FetchType.LAZY, mappedBy = "author")

@Fetch(FetchMode.SUBSELECT)

private Set<Book> books;

}

The first query will load all authors:

SELECT author0\_.id AS id1\_0\_,

author0\_.fullName AS fullname2\_0\_

FROM authors author0\_;

The second query will fetch books by using authors sub query:

SELECT books0\_.author\_id AS author\_i4\_1\_1\_,

books0\_.id AS id1\_1\_1\_,

books0\_.id AS id1\_1\_0\_,

books0\_.author\_id AS author\_i4\_1\_0\_,

books0\_.title AS title2\_1\_0\_,

books0\_.year AS year3\_1\_0\_

FROM books books0\_

WHERE books0\_.author\_id in

(SELECT author0\_.id

FROM authors author0\_);

If you look closely into the IN condition, you’ll see that the code inside the sub query almost repeats the first query. It can slow the performance if we have to execute a very complex query twice. To speed up this case we can filter and page authors retrieving their ids by the 1st query. Then we can pass these identifiers directly to the 2nd query’s sub query:

List<Integer> authorIds = em.createQuery("select a.id from Author a", Integer.class)

.setFirstResult(5)

.setMaxResults(10)

.getResultList();

List<Author> resultList =

entityManager.createQuery("select a from Author a"

+ " left join fetch a.books"

+ " where a.id in :authorIds", Author.class)

.setParameter("authorIds", authorIds)

.getResultList();

[**https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping**](https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping)

Let's say you have a collection of Car objects (database rows), and each Car has a collection of Wheel objects (also rows). In other words, Car → Wheel is a 1-to-many relationship.

Now, let's say you need to iterate through all the cars, and for each one, print out a list of the wheels. The naive O/R implementation would do the following:

SELECT \* FROM Cars;

And then **for each Car:**

SELECT \* FROM Wheel WHERE CarId = ?

In other words, you have one select for the Cars, and then N additional selects, where N is the total number of cars.

Alternatively, one could get all wheels and perform the lookups in memory:

SELECT \* FROM Wheel;

This reduces the number of round-trips to the database from N+1 to 2. Most ORM tools give you several ways to prevent N+1 selects.

[**https://www.javacodemonk.com/n-1-problem-in-hibernate-spring-data-jpa-894097b9**](https://www.javacodemonk.com/n-1-problem-in-hibernate-spring-data-jpa-894097b9)

### Spring Data JPA Approach

If we are using Spring Data JPA, then we have two options to achieve this - using EntityGraph or using select query with fetch join.

public interface UserRepository extends CrudRepository<User, Long> {

List<User> findAllBy();

@Query("SELECT p FROM User p LEFT JOIN FETCH p.roles")

List<User> findWithoutNPlusOne();

@EntityGraph(attributePaths = {"roles"})

List<User> findAll();

}

### Hibernate Approach

If its pure Hibernate, then the following solutions will work.

Using HQL Query

"from User u join fetch u.roles roles roles"

Using Hibernate Criteria API

Criteria criteria = session.createCriteria(User.class);

criteria.setFetchMode("roles", FetchMode.EAGER);