**N+1 Problem in Spring JPA and Solution-2024**

In Spring Data JPA, the “N+1 problem” refers to a performance issue that can occur when retrieving entities and their associated relationships from a database. Let’s understand this problem with an example. Suppose you have two entities, Author and Book, with a one-to-many relationship where an author can have multiple books. Now, let’s say you want to retrieve all books and their corresponding authors. You might write the following query using Spring Data JPA:

@Repository

public interface BookRepository extends JpaRepository<Book, Long> {

@Query("SELECT b FROM Book b")

List<Book> findAllBooks();

}

If you use this method to fetch all books, Spring Data JPA will generate a single query to retrieve the books. However, when you access the author property of each book, it will trigger a separate query to fetch the associated author. So if you have N books, this will result in N+1 queries, one query to fetch the books and an additional query for each book to fetch its author.

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Book name | Book Price | Author ID |
| 202 | Great Expectation | 2300 | 202 |
| 203 | Oliver Twist | 4700 | 202 |
| 252 | Romeo and Juliet | 8000 | 252 |
| 253 | Hamlet | 9000 | 252 |
| 302 | Paradise Lost | 3000 | 302 |
| 303 | Paradise Regained | 5000 | 302 |
| 352 | The Cat and Shakespeare | 5000 | 352 |
| 353 | Kanthapura | 3000 | 352 |

Table: Author Table:Book

|  |  |
| --- | --- |
| ID | Author Name |
| 202 | Charles Dickens |
| 252 | William Shakespeare |
| 302 | John Milton |
| 352 | Raja Rao |

In plain SQL, if we want to retrieve all the data by joining, we normally write like this.

SELECT a.id as author\_id, a.name as author\_name, b.id as book\_id, b.name as book\_name, b.price as boo\_price

FROM author a **LEFT OUTER JOIN** book b ON a.id = b.author\_id;

We get the below output which is correct way of doing it.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Author id | Author name | Book id | Book Name | Book price |  |
| 202 | Charles Dickens | 202 | Great Expectation | 2300 |  |
| 202 | Charles Dickens | 203 | Oliver Twist | 4700 |  |
| 252 | William Shakespeare | 252 | Romeo and Juliet | 8000 |  |
| 252 | William Shakespeare | 253 | Hamlet | 9000 |  |
| 302 | John Milton | 302 | Paradise Lost | 3000 |  |
| 302 | John Milton | 303 | Paradise Regained | 5000 |  |
| 352 | Raja Rao | 352 | The Cat and Shakespeare | 5000 |  |
| 352 | Raja Rao | 353 | Kanthapura | 3000 |  |

To mitigate the N+1 problem, Spring Data JPA provides several solutions:

1. **Use the JOIN FETCH** keyword in query like **@Query("SELECT b FROM Book b JOIN FETCH b.author")**
2. **Using @NamedEntityGraph**, **@NamedEntityGraph(name = "Book.author", attributeNodes = @NamedAttributeNode("author") )** in class and Mention the name in Repository **@EntityGraph("Book.author")**
3. Using @BatchSize()
4. Using Hibernate Specific **@Fetch(FetchMode.SUBSELECT)**
5. Using @ManyToOne(fetch = FetchType.***EAGER***) 🡪 Not Recommended always

**Solution-1**: lazy fetching behavior and use the JOIN FETCH keyword in your query to fetch the associated entities eagerly.

**@Query("SELECT b FROM Book b JOIN FETCH b.author")**

List<Book> findAllBooks();

Details:

Define in **Author Class** as below.

@OneToMany(fetch = FetchType.***LAZY***, mappedBy = "author", cascade = CascadeType.***ALL***)

**private** Set<Book> books = **new** HashSet<>();

Define in **Book Class** as below.

@ManyToOne

**private** Author author;

In Repository Layer, define as below.

@Repository

**public** **interface** BookCrudRepository **extends** CrudRepository<Book, Long> {

@Query("select b from Book b join fetch b.author")

**public** List<Book> getAllBooks();

}

Testing method

@Autowired

**private** BookCrudRepository bookCrudRepo;

bookCrudRepo.getAllBooks().forEach( book-> {

System.***out***.println("Book: "+book);

Author author = book.getAuthor();

System.***out***.println("Author: "+author);

});

Generated Query from Output as given below.

Select b1\_0.id, a1\_0.id, a1\_0.name, b1\_0.name, b1\_0.price

from

book b1\_0

join author a1\_0 on a1\_0.id=b1\_0.author\_id

**Solution-2**: using @NamedEntityGraph

Define a normal **Author class** as below.

@Entity(name = "Author")

@Table(name = "author")

@Getter @Setter @ToString(exclude = "books") //Otherwise, it will throw Stackoverflow error

**public** **class** Author {

@OneToMany(fetch = FetchType.***LAZY***, mappedBy = "author", cascade = CascadeType.***ALL***)

**private** Set<Book> books = **new** HashSet<>();

}

Define @NamedEntityGraph in Book Class as below.

@Entity(name = "Book") @Table(name = "book")

@Getter @Setter @ToString

@NamedEntityGraph(name = "Book.author", attributeNodes = @NamedAttributeNode("author") )

**public** **class** Book {

@ManyToOne

**private** Author author;

}

In Repository Layer, define as below.

@Repository

**public** **interface** BookCrudRepository **extends** CrudRepository<Book, Long> {

**@EntityGraph("Book.author")**

**@Query("select b from Book b")**

**public** List<Book> getAllBooksUsingNamedEntityGraph();

}

Generated Query given below.

select

b1\_0.id,

a1\_0.id,

a1\_0.name,

b1\_0.name,

b1\_0.price

from

book b1\_0

left join

author a1\_0

on a1\_0.id=b1\_0.author\_id

You can also use only **@EntityGraph** in the repository layer without declaring @NamedEntityGraph. The code is given below.

@Repository

**public** **interface** BookCrudRepository **extends** CrudRepository<Book, Long> {

// Only using @EntityGraph

**@EntityGraph(attributePaths = {"author"})**

@Query("select b from Book b")

**public** List<Book> getAllBooksOnlyUsingEntityGraph();

}

**Solution-3**: Solution using @BatchSize()

BatchSize basically groups your N part of the N+1 so that SQL becomes a IN query, reducing the number of total queries. The @BatchSize makes sense on both **One-To-Many** and **Many-to-One** as well

@BatchSize makes more sense for **to-many** associations (e.g. @OneToMany), not for to-one relations.

With batch fetching, you have (M/N + 1) database roundtrips, where M is the number of children entities in your uninitialized to-many association and N is the batch size.

In Author Class, define as below.

@Entity(name = "Author")

@Table(name = "author")

@Getter

@Setter

@ToString(exclude = "books") //Otherwise, it will throw Stackoverflow error

**public** **class** Author {

@BatchSize(size = 3)

@OneToMany(fetch = FetchType.***LAZY***, mappedBy = "author", cascade = CascadeType.***ALL***)

**private** Set<Book> books = **new** HashSet<>();

}

In Book Class, define as below.

@Entity(name = "Book")

@Table(name = "book")

@Getter

@Setter

@ToString

**public** **class** Book {

@ManyToOne(fetch = FetchType.***LAZY***)

**private** Author author;

}

No change in Repository Layer

Testing code

bookCrudRepo.findAll().forEach(book -> {

System.***out***.println("Book: " + book);

Author author = book.getAuthor();

System.***out***.println("Author: " + author);

});

Generated output

Hibernate:

select

b1\_0.id,

b1\_0.author\_id,

b1\_0.name,

b1\_0.price

from

book b1\_0

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

where

a1\_0.id=?

Book: Book(id=202, name=Great Expectation, price=2300, author=Author(id=202, name=Charles Dickens))

Author: Author(id=202, name=Charles Dickens)

Book: Book(id=203, name=Oliver Twist, price=4700, author=Author(id=202, name=Charles Dickens))

Author: Author(id=202, name=Charles Dickens)

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

where

a1\_0.id=?

Book: Book(id=252, name=Romeo and Juliet, price=8000, author=Author(id=252, name=William Shakespeare))

Author: Author(id=252, name=William Shakespeare)

Book: Book(id=253, name=Hamlet, price=9000, author=Author(id=252, name=William Shakespeare))

Author: Author(id=252, name=William Shakespeare)

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

where

a1\_0.id=?

Book: Book(id=302, name=Paradise Lost, price=3000, author=Author(id=302, name=John Milton))

Author: Author(id=302, name=John Milton)

Book: Book(id=303, name=Paradise Regained, price=5000, author=Author(id=302, name=John Milton))

Author: Author(id=302, name=John Milton)

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

where

a1\_0.id=?

Book: Book(id=352, name=The Cat and Shakespeare, price=5000, author=Author(id=352, name=Raja Rao))

Author: Author(id=352, name=Raja Rao)

Book: Book(id=353, name=Kanthapura, price=3000, author=Author(id=352, name=Raja Rao))

Author: Author(id=352, name=Raja Rao)

**Solution-4**: Using Hibernate Specific **@Fetch(FetchMode.SUBSELECT)**

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

In Author Class, define as below.

@Entity(name = "Author")

@Table(name = "author")

@Getter

@Setter

@ToString(exclude = "books") //Otherwise, it will throw Stackoverflow error

**public** **class** Author {

@OneToMany(fetch = FetchType.***EAGER***, mappedBy = "author", cascade = CascadeType.***ALL***)

@Fetch(FetchMode.***SUBSELECT***) // Hibernate Specific

**private** Set<Book> books = **new** HashSet<>();

}

In Book Class, define as below.

@Entity(name = "Book")

@Table(name = "book")

@Getter

@Setter

@ToString

**public** **class** Book {

@ManyToOne

**private** Author author;

}

Define Repository Class like this.

@Repository

**public** **interface** AuthorRepository **extends** CrudRepository<Author, Long> {

}

Testing Code

authorRepo.findAll().forEach( author -> {

System.***out***.println("Author: " + author);

author.getBooks().forEach( book -> {

System.***out***.println("Book: " + book);

});

});

Generated Query

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

Hibernate:

select

b1\_0.author\_id,

b1\_0.id,

b1\_0.name,

b1\_0.price

from

book b1\_0

where

b1\_0.author\_id in (select

a1\_0.id

from

author a1\_0)

**Solution-5**: Using @ManyToOne(fetch = FetchType.***EAGER***)

It is still not a recommended solution, it is prone to N+1 in case of large dataset.

Note: Important to Know

In Author Class, define as below.

@Entity(name = "Author")

@Table(name = "author")

@Getter

@Setter

@ToString(exclude = "books") //Otherwise, it will throw Stackoverflow error

**public** **class** Author {

@OneToMany(mappedBy = "author", cascade = CascadeType.***ALL***)

**private** Set<Book> books = **new** HashSet<>();

}

In Book Class, define as below.

@Entity(name = "Book")

@Table(name = "book")

@Getter

@Setter

@ToString

**public** **class** Book {

@ManyToOne(fetch = FetchType.***EAGER***)

**private** Author author;

}

Testing Code below.

authorRepo.findAll().forEach( author -> {

System.***out***.println("--------------- START -------------");

System.***out***.println("Author: " + author);

author.getBooks().forEach( book -> {

System.***out***.println("Book: " + book);

});

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ END \_\_\_\_\_\_\_\_\_\_\_\_\_\_");

});

Generated Code given below.

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

--------------- START -------------

Author: Author(id=202, name=Charles Dickens)

Hibernate:

select

b1\_0.author\_id,

b1\_0.id,

b1\_0.name,

b1\_0.price

from

book b1\_0

where

b1\_0.author\_id=?

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

where

a1\_0.id=?

Book: Book(id=202, name=Great Expectation, price=2300, author=Author(id=202, name=Charles Dickens))

Book: Book(id=203, name=Oliver Twist, price=4700, author=Author(id=202, name=Charles Dickens))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ END \_\_\_\_\_\_\_\_\_\_\_\_\_\_

--------------- START -------------

Author: Author(id=252, name=William Shakespeare)

Hibernate:

select

b1\_0.author\_id,

b1\_0.id,

b1\_0.name,

b1\_0.price

from

book b1\_0

where

b1\_0.author\_id=?

Hibernate:

select

a1\_0.id,

a1\_0.name

from

author a1\_0

where

a1\_0.id=?

Book: Book(id=252, name=Romeo and Juliet, price=8000, author=Author(id=252, name=William Shakespeare))

Book: Book(id=253, name=Hamlet, price=9000, author=Author(id=252, name=William Shakespeare))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ END \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Complete Repository class is given below.

@Repository

**public** **interface** BookCrudRepository **extends** CrudRepository<Book, Long> {

// Following solves N+1 issue, but FetType.LAZY should be defined

**@Query("select b from Book b join fetch b.author")**

**public** List<Book> getAllBooks();

@EntityGraph("Book.author")

@Query("select b from Book b")

**public** List<Book> getAllBooksUsingNamedEntityGraph();

// Only using @EntityGraph

**@EntityGraph(attributePaths = {"author"})**

@Query("select b from Book b")

**public** List<Book> getAllBooksOnlyUsingEntityGraph();

}

By default, JPA employs lazy loading for relationships to load only the necessary data. However, lazy loading can lead to the “**Could not initialize proxy – no Session**” error. To prevent this error, we can use FetchType.EAGER to load the entire relationship in a single query, eliminating the need for additional database requests.

or you can use the below properties in application.properties, sometimes, it is called as Antipattern.

**spring.jpa.properties.hibernate.enable\_lazy\_load\_no\_trans=true**

or

**@Query("SELECT c FROM COMPANY c JOIN FETCH c.employees")**

**public List<Company> getAll();**

**Note: FETCH STRATEGY**

**Fetch Strategy**

**==============**

**@OneToMany 🡺 LAZY**

**@ManyToMany 🡺 LAZY**

**@ManyToOne 🡺 EAGER**

**@OneToOne 🡺 EAGER**