**N+1 Problem in Spring JPA and Solution-2025**

The **N+1 problem** is a performance issue that arises when fetching related entities inefficiently. It occurs when:

1. A single query (1) is executed to fetch a list of parent entities.
2. For each parent entity (N), an additional query is executed to fetch its related child entities.

Imagine you have two entities with a relationship, for example, Author and Book, where an Author can have many Books (one-to-many relationship).

1. **The "1" Query:** When you fetch a list of parent entities (e.g., all Authors), JPA executes one SQL query to retrieve these Author records.
2. **The "N" Queries:** If you then iterate through this list of Authors and, for each Author, try to access their associated Books, JPA might execute a separate SQL query for *each* Author to fetch their Books. If you have 'N' authors, this results in 'N' additional queries.

**Total Queries = 1 (for Authors) + N (for each Author's Books) = N+1 queries.**

**Solutions to the N+1 Problem**

* **JOIN FETCH 🡺 @Query("SELECT b FROM Book b JOIN FETCH b.author")**
* **@EntityGraph, @EntityGraph("books") or @NamedEntityGraph(name = "Book.author", attributeNodes = @NamedAttributeNode("author") )**
* **@BatchSize**
* **DTO Projections:** Instead of fetching full entities, you can fetch only the necessary data into Data Transfer Objects (DTOs) using constructor expressions in JPQL or Spring Data JPA's interface-based projections.
* Using Hibernate Specific **@Fetch(FetchMode.SUBSELECT)**

**Consider the following Example of Author and Book**

@Entity(name = "Author") @Table(name = "author")

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

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

@Id @GeneratedValue

**private** Long id;

@Column(name = "name")

**private** String name;

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

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

**public** Author() {

**super**();

}

**public** Author(String name) {

**super**();

**this**.name = name;

}

}

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

@Getter @Setter @ToString

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

@Id @GeneratedValue

**private** Long id;

@Column(name = "name")

**private** String name;

@Column(name = "price")

**private** **int** price;

**public** Book() {

**super**();

}

**public** Book(String name, **int** price) {

**super**();

**this**.name = name;

**this**.price = price;

}

}

Repository class is given below

@Repository

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

}

Database Tables Created below.

**create table author\_books (**

**author\_id bigint not null,**

**books\_id bigint not null,**

**primary key (author\_id, books\_id)**

**)**

**create table book (**

**id bigint not null,**

**name varchar(255),**

**price integer,**

**primary key (id)**

**)**

**create table author (**

**id bigint not null,**

**name varchar(255),**

**primary key (id)**

**)**

Table: **Author\_Books**

author\_id | books\_id

-----------------+---------------

1 | 1

1 | 2

2 | 3

2 | 4

Table: **Book**

id | name | price

---+---------------------------------+------

1 | Great Expectation | 2300

2 | Oliver Twist | 4700

3 | Romeo and Juliet | 8000

4 | Hamlet | 9000

Table: **Author**

id | name

---+--------------------

1 | Charles Dickens

2 | William Shakespeare

As part of testing, we use the below method.

**public** **void** getAllAuthorDetails() {

Iterable<Author> iterable = authorRepo.findAll();

List<Author> authorList = StreamSupport

.*stream*(iterable.spliterator(), **false**)

.collect(Collectors.*toList*());

System.***out***.println("Author List Size: "+authorList.size());

sleep(5);

System.***out***.println("------------- AFTER 5 SECONDS ---------------");

authorList.forEach( author -> {

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

Set<Book> books = author.getBooks();

books.forEach(book -> System.***out***.println("Book Details: "+book));

});

}

**Query generated by Hibernate in simplified form**

**Query-1:** SELECT a.id, a.name FROM author a; 🡸 Only one query is fired to get the authors details.

To get the book details for each particular author, the following queries are fired.

Joining two tables **author\_books & book**

**Query-1:** SELECT ab.author\_id, b.id, b.name, b.price FROM **author\_books ab** JOIN **book b** ON b.id = ab.books\_id

WHERE ab.author\_id = ?;

**Query-2:** SELECT ab.author\_id, b.id, b.name, b.price FROM **author\_books ab** JOIN **book b** ON b.id = ab.books\_id

WHERE ab.author\_id = ?

So finally, 3 queries are fired to get the Author and their book details. In this example there are 2 authors and each author has 2 books.

**Solution-1**: Using **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();

Repository class looks like this

@Repository

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

**@Query("select a from Author a join fetch a.books")**

**public** Iterable<Author> findAll();

}

Now only one query will be generated to get the Author and their corresponding Book details.

SELECT a.id, a.name, b.id, b.name, b.price, ab.author\_id FROM **author a**

JOIN **author\_books ab** ON a.id = ab.author\_id

JOIN **book b** ON b.id = ab.books\_id;

In a more simplified way,

SELECT a.\*, b.\*, ab.author\_id FROM **author a**

JOIN **author\_books ab** ON a.id = ab.author\_id

JOIN **book b** ON b.id = ab.books\_id;

**Solution-2**: using **@EntityGraph** or **@NamedEntityGraph**

Repository class is given below

@Repository

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

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

**@Query("select a from Author a")**

**public** List<Author> findAllAuthors();

}

Testing code is given below

**public** **void** getAllAuthorDetailsUsingEntityGraph() {

List<Author> authorList = authorRepo.findAllAuthors();

System.***out***.println("Author List Size: "+authorList.size());

sleep(5);

System.***out***.println("------------- AFTER 5 SECONDS ---------------");

authorList.forEach( author -> {

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

Set<Book> books = author.getBooks();

books.forEach(book -> System.***out***.println("Book Details: "+book));

});

}

**Generated Query given below.**

**SELECT** a.id, a.name, b.id, b.name, b.pr**ice FROM author a**

**LEFT JOIN author\_books ab ON a.id = ab.author\_id**

**LEFT JOIN book b ON b.id = ab.books\_id;**

**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 = 2)

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

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

}

No change in Book Class.

Repository class is like this.

@Repository

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

}

Testing code

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

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

Author author = book.getAuthor();

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

});

**Generated Query given below.**

**Query-1: Select a.id, a.name from author a**

**Query-2: SELECT ab.author\_id, b.id, b.name, b.price FROM author\_books ab JOIN book b ON b.id = ab.books\_id**

**WHERE ab.author\_id = ?;**

**Query-3: SELECT ab.author\_id, b.id, b.name, b.price FROM author\_books ab JOIN book b ON b.id = ab.books\_id**

**WHERE ab.author\_id = ?;**

**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***, cascade = CascadeType.***ALL***)

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

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

}

No change in Book Class.

Repository class is 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 given below.**

**Query-1: Select a.id, a.name from author a**

**Query-2: SELECT ab.author\_id, b.id, b.name, b.price FROM author\_books ab JOIN book b ON b.id = ab.books\_id**

**WHERE ab.author\_id = ?;**

**Query-3: SELECT ab.author\_id, b.id, b.name, b.price FROM author\_books ab JOIN book b ON b.id = ab.books\_id**

**WHERE ab.author\_id = ?;**

**Notes:**

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**

**Explain left outer join in sql with good example**

A LEFT OUTER JOIN (often shortened to LEFT JOIN) in SQL returns all rows from the left table (the first table mentioned in the FROM clause) and the matching rows from the right table. If there's no match for a row in the left table, the columns from the right table will have NULL values.

Here's a breakdown:

"Left" Table: The table specified before the LEFT JOIN keyword.

"Right" Table: The table specified after the LEFT JOIN keyword.

Matching Rows: Rows where the join condition (specified in the ON clause) is true for both tables.

Non-Matching Rows from Left Table: All rows from the left table are included, regardless of whether a match exists in the right table. For these rows, the columns from the right table will display NULL.

Non-Matching Rows from Right Table: If a row in the right table doesn't have a match in the left table, it will not be included in the result set.

**When to use LEFT JOIN**

You'd typically use a LEFT JOIN when you want to retrieve all information from one primary table and then optionally add related information from another table. Common scenarios include:

Examples

Listing all customers and their orders (even if some customers haven't placed any orders).

Showing all products and their associated categories (even if some products haven't been assigned a category yet).

Retrieving all employees and their assigned departments (even if some employees are not yet assigned to a department).

Table: **Employees**

EmployeeID EmployeeName DepartmentID

1 Alice 101

2 Bob 102

3 Charlie NULL

4 David 101

Table: **Departments**

DepartmentID DepartmentName

101 Sales

102 Marketing

103 HR

Now, let's perform a LEFT JOIN to see all employees and their respective department names. We want to see all employees, even if they don't have an assigned department.

**SQL Query**

**SELECT e.EmployeeID, e.Name, d.DepartmentName FROM Employees e**

**LEFT OUTER JOIN Departments d ON e.DepartmentID = d.DepartmentID;**

**INNER JOIN vs LEFT OUTER JOIN**

| **Feature** | **INNER JOIN** | **LEFT OUTER JOIN** |
| --- | --- | --- |
| **Matching Rows** | Returns only rows where there is a match in both tables | Returns all rows from the left table, even if there is no match in the right table |
| **Unmatched Rows** | Excluded (discarded) | Included with NULL values for the right table's columns |
| **Result Set Size** | Smaller, only matched records | Larger, includes both matched and unmatched records from the left table |

**Example Comparison**

**INNER JOIN**

SELECT e.EmployeeID, e.Name, d.DepartmentName FROM Employees e INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

✅ **Only employees with matching departments are shown.**  
❌ **Charlie and David (who don’t have matching departments) are excluded.**

**LEFT OUTER JOIN**

SELECT e.EmployeeID, e.Name, d.DepartmentName FROM Employees e LEFT OUTER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

✅ **All employees are included, even if they don’t have a matching department.**  
❌ **Charlie and David appear, but their DepartmentName will be NULL.**

**Key Takeaway**

* Use **INNER JOIN** when you only need matching records.
* Use **LEFT OUTER JOIN** when you need all records from the left table, even if there’s no match.

**What is self Join**

A **self join** is a join where a table is joined with itself. It’s useful when you need to compare rows within the same table.

Example: Employee Hierarchy

Consider an Employees table that stores employees and their managers.

**Employees Table**

| **EmployeeID** | **Name** | **ManagerID** |
| --- | --- | --- |
| 1 | Alice | NULL |
| 2 | Bob | 1 |
| 3 | Charlie | 1 |
| 4 | David | 2 |

**Self Join Query**

SELECT e1.Name AS Employee, e2.Name AS Manager FROM Employees e1

LEFT JOIN Employees e2 ON e1.ManagerID = e2.EmployeeID;

**Result**

| **Employee** | **Manager** |
| --- | --- |
| Alice | NULL |
| Bob | Alice |
| Charlie | Alice |
| David | Bob |

**Explanation**

* Alice has **no manager** (NULL).
* Bob and Charlie report to Alice.
* David reports to Bob.

The **self join** allows us to retrieve relationships within the same table!

**How to use Multiple JOIN FETCH**

Entities are given below.

@Entity(name = "Company") @Table(name = "company")

@NoArgsConstructor @Setter @Getter

**public** **class** Company {

@Id

@GeneratedValue

**private** Long id;

**private** String name;

**public** Company(String name) {

**this**.name = name;

}

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

**private** Set<Department> departments = **new** HashSet<>();

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

**private** Set<Location> locations = **new** HashSet<>();

}

@Entity(name = "Department") @Table(name = "dept")

@Getter @Setter @ToString @NoArgsConstructor

**public** **class** Department {

@Id

@GeneratedValue

**private** Long id;

**private** String name;

**public** Department(String name) {

**this**.name = name;

}

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

**private** Set<Employee> employees = **new** HashSet<>();

}

@Entity(name="Employee") @Table(name = "emp")

@Getter @Setter @ToString @NoArgsConstructor

**public** **class** Employee {

@Id @GeneratedValue

**private** Long id;

**private** String name;

**public** Employee(String name) {

**this**.name = name;

}

}

@Entity(name="Location") @Table(name = "location")

@Getter @Setter @ToString @NoArgsConstructor

**public** **class** Location {

@Id @GeneratedValue

**private** Long id;

**private** String name;

**public** Location(String name) {

**this**.name = name;

}

}

Repository class is given below.

@Repository

**public** **interface** CompanyRepo **extends** CrudRepository<Company, Long> {

@Query(value = "select c from Company c **join fetch c.departments** join fetch **c.departments.employees** **join fetch c.locations** where c.id = :id") //Correct

Optional<Company> findById(Long id);

}

You can use like this

**@Query(value = "select c from Company c join fetch c.departments join fetch c.locations where c.id = :id")**

**@Query(value = "select c from Company c join fetch c.departments join fetch c.departments.employees join fetch c.locations where c.id = :id")**

**join fetch c.departments join fetch c.locations 🡸 with space**

**join fetch c.departments join fetch c.departments.employees 🡸 employees inside department**

**Simplified Query generated by Hibernate will be like this.**

**SELECT c.id AS company\_id, c.name AS company\_name, d.id AS department\_id, d.name AS department\_name,**

**e.id AS employee\_id, e.name AS employee\_name, l.id AS location\_id, l.name AS location\_name**

**FROM company c**

**JOIN company\_departments cd ON c.id = cd.company\_id**

**JOIN dept d ON d.id = cd.departments\_id**

**JOIN dept\_employees de ON d.id = de.department\_id**

**JOIN emp e ON e.id = de.employees\_id**

**JOIN company\_locations cl ON c.id = cl.company\_id**

**JOIN location l ON l.id = cl.locations\_id**

**WHERE c.id = ?**