El problema de las n+1 Queries es uno de los problemas más habituales cuando trabajamos con Domain Driven Design y frameworks de Persistencia . ¿Que es una n+1 Query y como nos podemos encontrar en una situación límite con ellas?. Vamos a explicarlo paso a paso . En primer lugar el problema aparece normalmente cuando trabajamos con algún framework de Persistencia . Puede ser Hibernate o puede ser cualquier otro como EntityFramework o Larabel dependiendo de la plataforma y hace referencia a estos frameworks abordan los problemas relacionados con consultas que incluyen relaciones . Vamos a construir un ejemplo sencillo .

Para ello partiremos de dos clases . La clase Libro y la clase Capítulo. Se trata de dos clases que van a estar relacionadas con JPA.

```
package com.arquitecturajava.data1;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "Libros")
public class Libro {

    @Id
    private String isbn;
    private String titulo;
    private String autor;
    private double precio;
```

```
    private Date fecha;
    @OneToMany(mappedBy = "libro")
    private List<Capitulo> capitulos = new ArrayList<>();

    // Constructor por defecto
    public Libro() {
        super();
    }

    // Constructor con ISBN
    public Libro(String isbn) {
        this.isbn = isbn;
    }

    // Constructor con todos los atributos
    public Libro(String isbn, String titulo, String autor, double
precio, Date fecha) {
        this.isbn = isbn;
        this.titulo = titulo;
        this.autor = autor;
        this.precio = precio;
        this.fecha = fecha;
    }

    // Getters y Setters
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
```

```
}

public String getTitulo() {
    return titulo;
}

public void setTitulo(String titulo) {
    this.titulo = titulo;
}

public String getAutor() {
    return autor;
}

public void setAutor(String autor) {
    this.autor = autor;
}

public double getPrecio() {
    return precio;
}

public void setPrecio(double precio) {
    this.precio = precio;
}

public Date getFecha() {
    return fecha;
}

public void setFecha(Date fecha) {
```

```
        this.fecha = fecha;
    }

    public List<Capitulo> getCapitulos() {
        return capitulos;
    }

    public void setCapitulos(List<Capitulo> capitulos) {
        this.capitulos = capitulos;
    }

    // Métodos equals y hashCode
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((isbn == null) ? 0 :
isbn.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Libro other = (Libro) obj;
        if (isbn == null) {
```

```
            if (other.isbn != null)
                return false;
        } else if (!isbn.equals(other.isbn))
            return false;
        return true;
    }
}
```

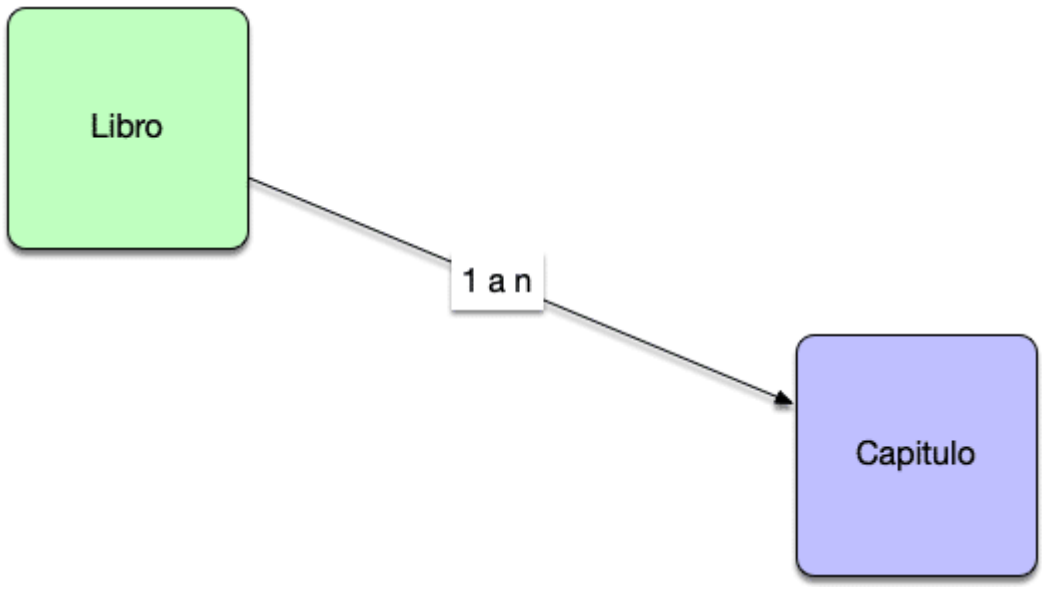
Vamos a ver la otra entidad Capitulos:

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
@Entity
@Table(name="capitulos")
public class Capitulo {
  @Id
  private String titulo;
  private int paginas;
  @ManyToOne
  @JoinColumn(name="libros_isbn")
  private Libro libro;
  public Libro getLibro() {
    return libro;
  }
  public void setLibro(Libro libro) {
    this.libro = libro;
  }
  public String getTitulo() {
    return titulo;
```

```
  }
  public void setTitulo(String titulo) {
    this.titulo = titulo;
  }
  public int getPaginas() {
    return paginas;
  }
  public void setPaginas(int paginas) {
    this.paginas = paginas;
  }
  public Capitulo(String titulo, int paginas) {
    super();
    this.titulo = titulo;
    this.paginas = paginas;
  }
  public Capitulo() {
    super();
  }
  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((titulo == null) ? 0 :
titulo.hashCode());
    return result;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
```

```
      return false;
    if (getClass() != obj.getClass())
      return false;
    Capitulo other = (Capitulo) obj;
    if (titulo == null) {
      if (other.titulo != null)
        return false;
    } else if (!titulo.equals(other.titulo))
      return false;
    return true;
  }
}
```

Como podemos observar se trata de una relación 1 a n . Un libro contiene n capítulos :

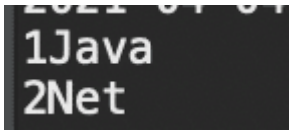


Hasta aquí todo muy muy correcto. Podemos construir una pequeña prueba unitaria que nos solicite los libros y nos imprima los isbn y títulos por la consola. En este caso lo voy a realizar con Spring Boot por comodidad ya que me permite de una forma relativamente sencilla cargar datos :

```
package com.arquitecturajava.data1;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.Transactional;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.jdbc.Sql;

@SpringBootTest
@Sql({ "/schema.sql", "/data.sql" })
class DataPruebas {

    @PersistenceContext
    private EntityManager em;

    @Test
    @Transactional
    public void testLibros() {
        List<Libro> libros = em.createQuery("select l from Libro l",
Libro.class).getResultList();

        for (Libro l : libros) {
            System.out.println(l.getIsbn());
        }
    }
}
```

Realmente no estoy ejecutando ninguna prueba unitaria en sí sino simplemente solicitando

la información de los libros una vez realizada la consulta . Esta información la podremos ver pasar por la consola.



Junto con esa información tenemos también acceso a las consultas que Hibernate generó para obtener los datos . En este caso todo es muy normal ya que realiza un select clásico solicitando la información requerida:

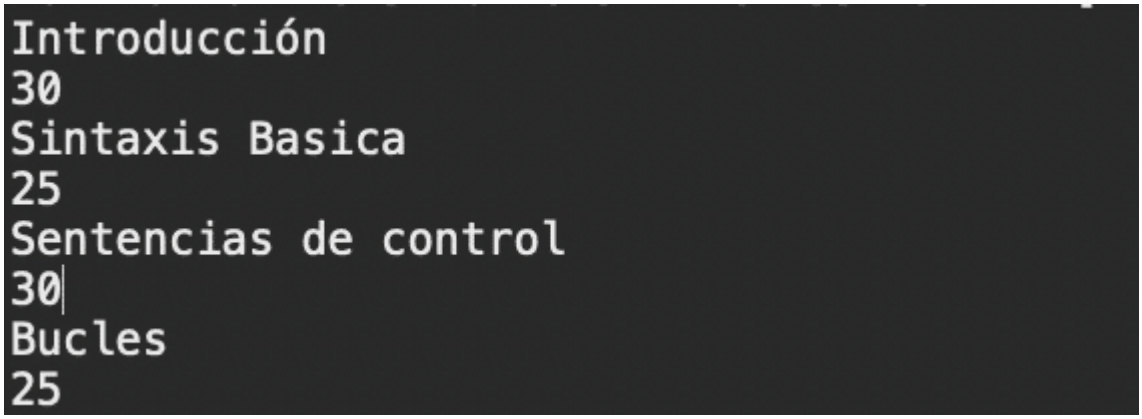
Todo ha funcionado correctamente . ¿Ahora bien que es lo que sucede si yo intento recorrer los objetos de otra forma y deseo acceder a información de los capítulos? .Vamos a verlo en acción , para ello la modificación de nuestro código es muy puntual:

```
package com.arquitecturajava.data1;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.Transactional;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.jdbc.Sql;

@SpringBootTest
@Sql({ "/schema.sql", "/data.sql" })
class DataPruebas {
```

```
    @PersistenceContext
    private EntityManager em;

    @Test
    @Transactional
    public void test() {
        List<Libro> libros = em.createQuery("select l from Libro l",
Libro.class).getResultList();

        for (Libro l : libros) {
            System.out.print(l.getIsbn());
            System.out.println(l.getTitulo());

            for (Capitulo c : l.getCapitulos()) {
                System.out.println(c.getTitulo());
                System.out.println(c.getPaginas());
            }
        }
    }
}
```

El código se ejecuta y nos muestra la información de cada uno de los libros con sus capítulos por la consola :

```
1Java
```



Normalmente . la gente no le da muchas más vueltas y sigue trabajando . Lamentablemente dentro de este código hay un problema importante de n+1 Queries ya que si nos fijamos Hibernate ha ejecutado las siguientes Queries simplemente para traer la información del primer Libro y sus Capítulos.

```
SELECT
    libro0_.isbn AS isbn1_1_,
    libro0_.autor AS autor2_1_,
    libro0_.fecha AS fecha3_1_,
    libro0_.precio AS precio4_1_,
    libro0_.titulo AS titulo5_1_
FROM
    Libros libro0_;

SELECT
    capitulos0_.libros_isbn AS libros_i3_0_0_,
    capitulos0_.titulo AS titulo1_0_0_,
    capitulos0_.titulo AS titulo1_0_1_,
    capitulos0_.libros_isbn AS libros_i3_0_1_,
    capitulos0_.paginas AS paginas2_0_1_
FROM
    capitulos capitulos0_
WHERE
    capitulos0_.libros_isbn = ?
```

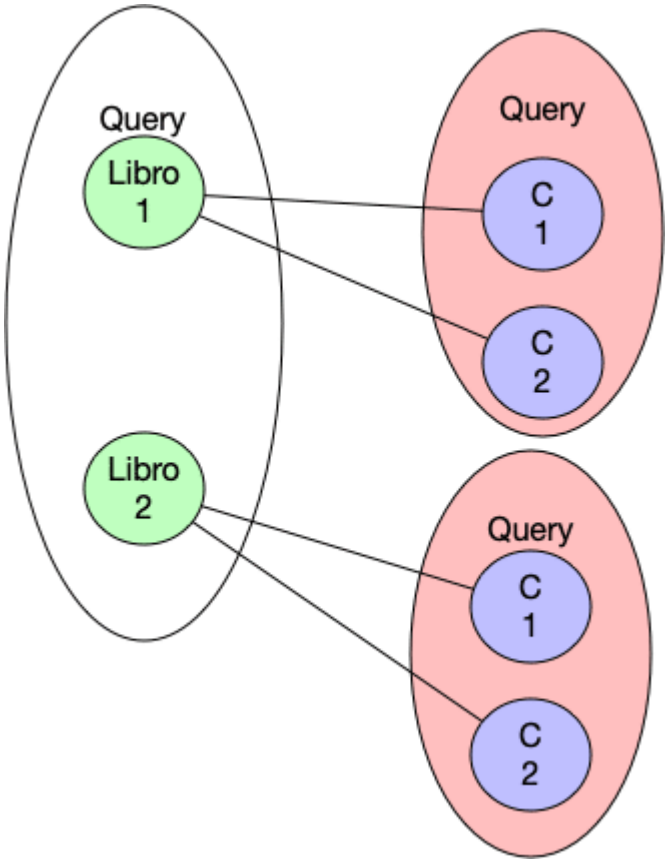
```
SELECT
    capitulos0_.libros_isbn AS libros_i3_0_0_,
    capitulos0_.titulo AS titulo1_0_0_,
    capitulos0_.titulo AS titulo1_0_1_,
    capitulos0_.libros_isbn AS libros_i3_0_1_,
    capitulos0_.paginas AS paginas2_0_1_
FROM
    capitulos capitulos0_
WHERE
    capitulos0_.libros_isbn = ?
```

Aquí podemos ver como consulta primero por todos los Libros y por cada Libro realiza una consulta adicional para traerse sus capítulos . En principio no parece un problema en sí pero la realidad es que es el comienzo del fin de tu aplicación .

Si este tipo de problemas no los atajas en un primer momento y te das cuenta de que tu framework de persistencia no esta realizando las consultas de forma apropiada para traerte la información . Nos encontraremos en un futuro con consultas que se traen de golpe un listado de 2000 libros y por cada libro realiza una consulta adicional solicitando sus capítulos. Por lo tanto tendremos 2001 consultas solo para traernos los Libros con sus capítulos. Cuando multipliquemos esto por el número de usuarios concurrentes nos encontraremos con una situación inabordable a nivel de rendimiento y tendremos que vernos obligados a revisar todos y cada una de las consultas.

¿Qué es lo que ha pasado exactamente ?

Lo que ha pasado es que “sabemos poco” sobre cómo utilizar el framework y lo hemos usado de forma incorrecta . Generando por cada listado de libros una consulta adicional por los capítulos de cada Libro.



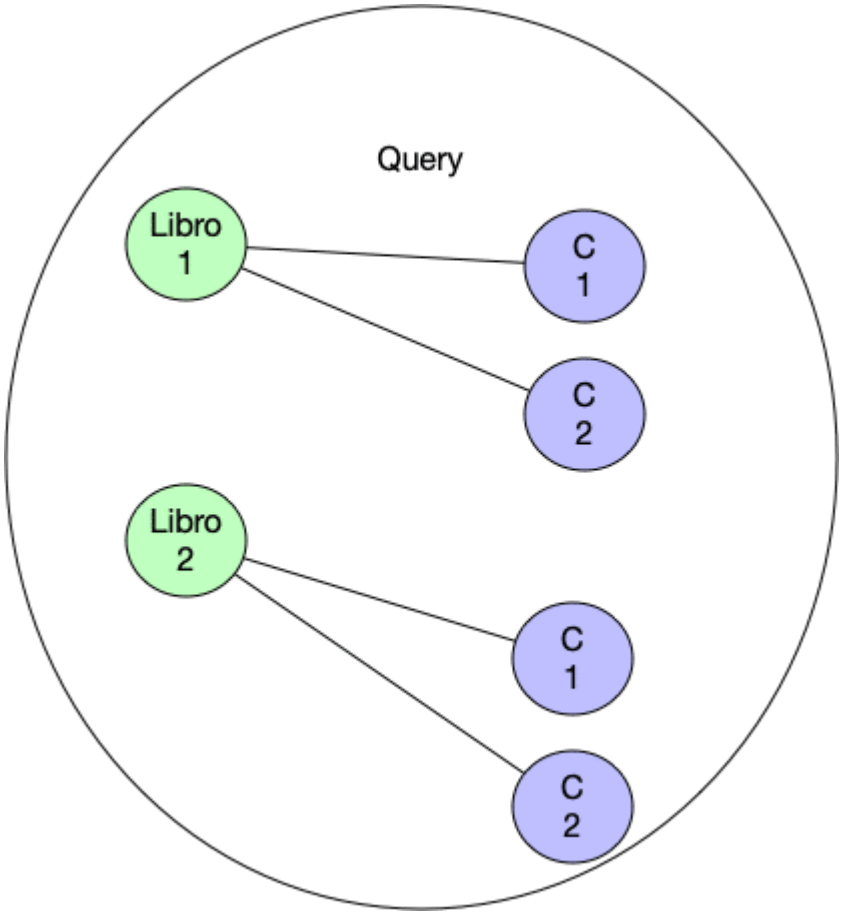
La primera consulta que busca los libros , es en principio correcta pero las consultas Hijas que realiza el framework para poder acceder al contenido de los capítulos es el principio del fin.

Solucionando el problema

¿Como podemos solventar el problema de las n+1 Queries? . Depende mucho del framework y depende mucho de la situación de cada uno . Pero lo importante es revisar la documentación del framework y orientarle a que no realize este tipo de consultas sino que haga una única consulta de golpe. En este caso nos valdría con una consulta de Fetch Join entre Libros y Capítulos:

```
List<Libro> libros = em.createQuery("select l from Libro l join fetch
l.capitulos", Libro.class).getResultList();
```

De esta forma únicamente generaremos una consulta :



El resultado en la consola de la consulta de fetch es :

```
SELECT
    libro0_.isbn AS isbn1_1_0_,
    capitulos1_.titulo AS titulo1_0_1_,
    libro0_.autor AS autor2_1_0_,
    libro0_.fecha AS fecha3_1_0_,
    libro0_.precio AS precio4_1_0_,
    libro0_.titulo AS titulo5_1_0_,
    capitulos1_.libros_isbn AS libros_i3_0_1_,
    capitulos1_.paginas AS paginas2_0_1_,
    capitulos1_.libros_isbn AS libros_i3_0_0__,
```

```
    capitulos1_.titulo AS titulo1_0_0__
FROM
    Libros libro0_
INNER JOIN
    capitulos capitulos1_
ON
    libro0_.isbn = capitulos1_.libros_isbn;
```

Acabamos de solventar nuestro problema de n+1 queries. Tengamos mucho cuidado cuando abordamos este tipo de problemas.

- JPA Streamer y consultas dinámicas
- Codigo Arquitectura Java JPA Domain Driven Design
- Spring Data Projections y Rendimiento
- Mis Libros