5 Common Hibernate Mistakes That Cause Unexpected Queries

1. Not Initializing Lazily Fetched Associations

The initialization of lazily fetched associations is one of the most common reasons for Hibernate to generate additional queries. If you have been using Hibernate for a while, you probably experienced this problem yourself. It's often called the n+1 select issue. Whenever you access an uninitialized, lazily fetched association, Hibernate executes a query that loads all elements of this association from the database.

Initialize all Required Associations

You can easily avoid this problem by initializing the associations you want to use while you retrieve an entity from the database. JPA and Hibernate offer several options to do that. The easiest one is JOIN FETCH or LEFT JOIN FETCH expression which you can use within your JPQL or Criteria Query. You can use it in the same way as a simple JOIN expression. It tells Hibernate not only to join the association within the query but also to fetch all elements of the association.

```
List<Author> authors = em.createQuery(
        "SELECT a FROM Author a LEFT JOIN FETCH a.books",
        Author.class).getResultList();
for (Author a : authors) {
    log.info(a.getFirstName() + " " + a.getLastName()
        + " wrote " + a.getBooks().size() + " books.");
}
```

2. Using the *FetchType.EAGER*

Not only the initialization of lazily fetched associations can cause lots of unexpected queries. That's also the case if you use FetchType.EAGER. It forces Hibernate to initialize the association as soon as it loads the entity. So, it doesn't even matter if you use the association or not. Hibernate will fetch it anyways. That makes it one of the most common performance pitfalls.

And before you tell me that you never declare any eager fetching in your application, please check your to-one associations. Unfortunately, JPA defines eager fetching as the default behavior for these associations.

Use *FetchType.LAZY* for all Associations

You should use FetchType.LAZY for all of your associations. It's the default for all to-many associations, so you don't need to declare it for them explicitly. But you need to do that for all to-one association. You can specify the FetchType with the fetch attribute of the @OneToOne or @ManyToOne association.

```
@Entity
public class Review {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "book_id")
    private Book book;

    ...
}
```

3. Removing Child Entities with CascadeType.Remove

Cascading provides a very comfortable option to remove all child entities when you delete their parent entity. In the example of this article, you could, for example, use it to remove all Books written by an Author, when you delete the Author entity.

Unfortunately, the removal of the associated entities isn't very efficient and can create serious side effects. I, therefore, recommend to **NOT** use CascadeType.Remove in your application.

Remove Child Entities With a Bulk Operation

If you don't use any lifecycle callbacks or EntityListeners and don't use any frameworks that use them, e.g., Hibernate Envers, you can use a JPQL, Criteria or native SQL query to remove all child entities with one bulk operation.

The following code sample shows a JPQL query that removes all Review entities associated with a given Book entity. As you can see, the syntax is pretty similar to SQL.

```
em.flush();
em.clear();

Book b = em.find(Book.class, 2L);

Query q = em.createQuery("DELETE FROM Review r WHERE
r.book.id = :bid");
q.setParameter("bid", 2L);
q.executeUpdate();

em.remove(b);
```

Hibernate doesn’t know which entities this operation removes. So, it can’t remove any of the deleted entities from the persistence context.

You either need to be sure that you didn’t fetch of the removed entities before you executed the bulk operation or you need to call the flush() and clear() methods on the EntityManager before you execute the JPQL query. This tells Hibernate to write all pending changes to the database and to detach all entities from the persistence context before you execute the remove operation. So, you can be sure that your persistence context doesn’t contain any outdated entities.

4. Modeling Many-to-Many Associations as a List

Another common but not that well-known mistake that causes Hibernate to execute lots of addition SQL statements is using the wrong data type to model many-to-many associations. If you map the association to an attribute of type java.util.List, Hibernate deletes all records and then inserts new records whenever you add an element to or remove one from the association.

Use a Set instead of a List

You can easily fix this issue by using a Set<Author> instead of List<Author>.

```
@Entity
public class Book {

    @ManyToMany
    @JoinTable(name = "book_author",
                joinColumns = { @JoinColumn(name = "fk_book") },
                inverseJoinColumns = { @JoinColumn(name = "fk_author")
})
    private Set<Author> authors = new HashSet<Author>();

    ...
}
```

5. Updating or Removing Entities One by One

The last mistake I want to talk about in this article slows down use cases that update or remove multiple database records in the same table. With SQL, you would create 1 SQL statement that updates or removes all affected records. That would also be the most efficient approach with JPA and Hibernate.

But that's not what Hibernate does if you update or remove multiple entities. Hibernate creates an update or remove statement for each entity. So, if you want to remove 100 entities, Hibernate creates and executes 100 SQL DELETE statements.

Use a Bulk Update or Remove Operation

Executing that many SQL statements is obviously not a very efficient approach. It's much better to implement a bulk operation as a native, JPQL or Criteria query.

This is the same approach as I showed you as the solution to mistake 3. You first need to flush and clear your persistence context before you execute a query that removes all entities that fulfill the defined criteria.

```
em.flush();
em.clear();

// Remove all entities referenced in the List ids variable
Query query = em.createQuery("DELETE Author a WHERE id IN
(:ids)");
query.setParameter("ids", ids);
query.executeUpdate();
```

Summary

As you've seen, the most common mistakes are not only easy to make, they are also very easy to avoid:

- When you're defining your associations, you should prefer FetchType.LAZY and map many-to-many associations to a java.util.Set.
- If your use case uses a lazily fetched association, you should initialize it within the query that loads the entity, e.g., with a JOIN FETCH expression.
- Cascading and updating or removing multiple entities require more SQL statements than you might expect. It's often better to implement a bulk operation as a native, JPQL or Criteria query.

By following these recommendations, you will avoid the most common mistakes that cause Hibernate to execute lots of unexpected queries. If you want to learn more about Hibernate performance optimizations, you should join the waiting list of my Hibernate Performance Tuning Online Training. I will reopen it to enroll a new class soon.