Why you should avoid CascadeType.REMOVE

The *CascadeType*s *REMOVE* and *ALL*, which includes *REMOVE*, provide a comfortable option to remove an entity together with all its child entities.

But it creates several issues for to-many associations, and you should only use it for to-one relationships.

Problems for To-Many Associations

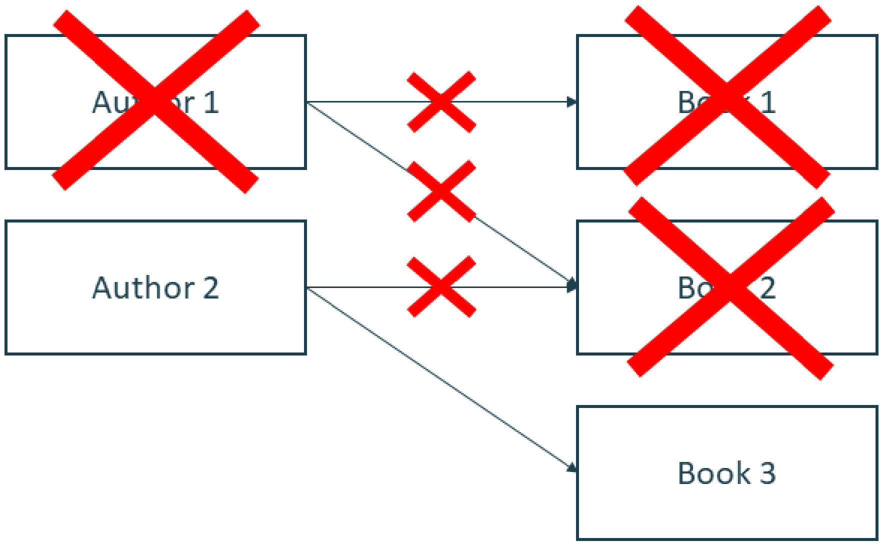
Too Many Queries

When you use CascadeType.REMOVE, Hibernate loads all associated entities and deletes them one by one. Depending on the size of the relationship, this might require a lot of queries and slow down your application.

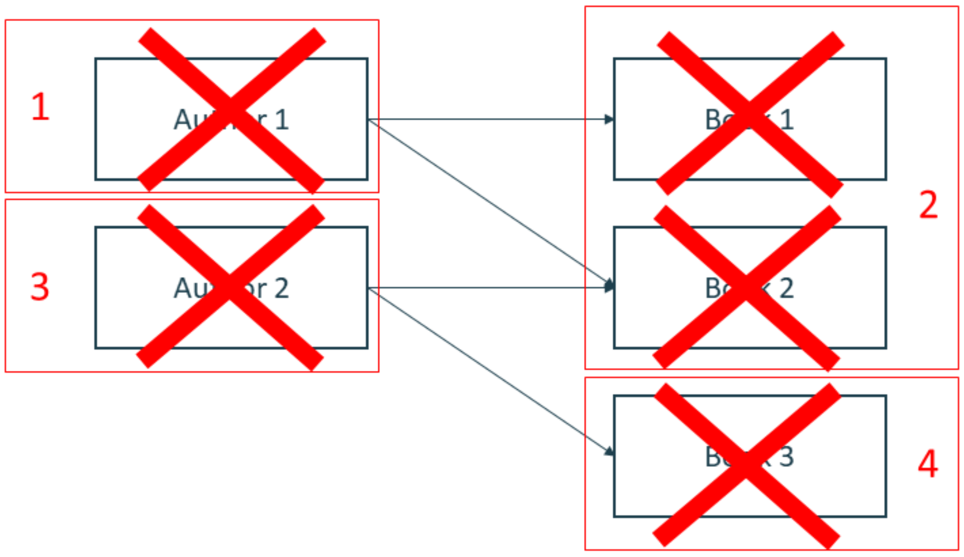
Remove More Than You Expected

When you use *CascadeType.REMOVE* on a many-to-many association, Hibernate removes all child entities. That includes the entities that are still associated to other parent entities.

So, in the following example Hibernate removes *Book* 1 and *Book* 2, even so *Book* 2 was still associated to *Author* 2.



And it gets even worse when you specify the *CascadeType.REMOVE* on both ends of the association. Hibernate then keeps cascading the remove operation until there are no associated entities left.



Solution

First of all, you shouldn't use the *CascadeType.REMOVE* for to-many associations. And that's also the case for *CascadeType.ALL* which includes the *CascadeType.REMOVE*.

When you don't use cascading, you need to delete the associated entities yourself. You can either do that by calling the remove method of the *EntityManager* for each entity or with a bulk operation.

Remove One By One

That is the easiest but not the most efficient approach. But you can, at least, be sure that you don't delete any records by accident.

You need to iterate through the list of associated *Book*s and check if it's associated with any other *Author*. If it's not, you call the remove method for it. Otherwise, you just delete the association to the *Author* entity.

```
Author a = em.find(Author.class, 1);
for (Book b : a.getBooks()) {
        if (b.getAuthors().size() == 1) {
                em.remove(b);
        } else {
                b.getAuthors().remove(a);
        }
}
```

As I said, this is not a very efficient approach. Hibernate has to perform 2 SQL DELETE operations for each *Book* entity you want to remove. One to remove the *Book* entity and another one to remove the records from the association table. And if you just want to remove the association to the *Author* entity, Hibernate has to delete the record from the association table.

Bulk Remove

When your association contains a lot of entities, it's better to remove them with a few queries.

This approach is much more complicated than the one I showed you before. But it needs a fixed number of queries and performs much better for huge associations.

You first get the ids of all *Book*s that *Author* 1 wrote alone and store them in a *List*. These are the ones you need to delete in a later step.

Then you remove all records from the association table that are linked to *Author* 1. That allows you to remove *Book* 1 without violating a foreign key constraint.

Now you use the *List* you got in step 1 and remove all *Book*s that *Author* 1 wrote alone.

And in the final step, you remove the *Author* entity.

```
Author a = em.find(Author.class, 1);

// get all books that this author wrote alone
Query q = em.createNativeQuery("SELECT ba.bookId FROM
BookAuthor ba JOIN Book b ON ba.bookId = b.id JOIN
BookAuthor ba2 ON b.id = ba2.bookId WHERE ba2.authorId
= ? GROUP BY ba.bookId HAVING count(ba.authorId) = 1");
q.setParameter(1, a.getId());
List<Integer> bookIds = (List<Integer>)q.getResultList();

// remove all associations for this author
q = em.createNativeQuery("DELETE FROM BookAuthor ba
WHERE ba.authorId = ?");
q.setParameter(1, a.getId());
q.executeUpdate();

// remove all books that this author wrote alone
q = em.createNativeQuery("DELETE FROM Book b WHERE
b.id IN (:ids)");
q.setParameter("ids", bookIds);
q.executeUpdate();

// remove author
em.remove(a);
```