6 Hibernate Mappings to Avoid for High-Performance Applications

Hibernate provides lots of mapping features that allow you to map complex domain and table models. But the availability of these features doesn't mean that you should use them in all of your applications. Some of them might be a great fit for smaller applications that are only used by a few users in parallel. But you should definitely not use them if you need to create a high-performance persistence layer.

In this article, I will show you 6 mapping features that will slow down your persistence layer. Let's start with some of Hibernate's and JPA's standard features.

1. Avoid FetchType.EAGER (and be cautious about to-one associations)

FetchType.EAGER tells your persistence provider to fetch a managed association as soon as you load the entity. So, it gets loaded from the database, whether or not you use the association in your business code. For most of your use cases, that means that you execute a few unnecessary database queries, which obviously slows down your application.

You can easily avoid that by using *FetchType.LAZY*. Hibernate will then only fetch the associated entities if you use the managed relationship in your business code. This is the default behavior for all to-many associations. For to-one associations, you need to set the FetchType and in your association mapping explicitly.

```
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String comment;
    @ManyToOne(fetch = FetchType.LAZY)
    private Book book;

    ...
}
```

When you do that, you need to pay special attention to one-to-one associations. As explained in a recent Hibernate Tip, lazy loading of one-to-one associations only works reliably for the entity that maps the foreign key column.

2. Don't map Many-to-Many associations to a List

Hibernate can map a many-to-many association to a *java.util.List* or a *java.util.Set*. Most developers expect that the mapping to a *java.util.List* is the easier and more efficient one. But that's not the case!

Removing an entry from a many-to-many association that you mapped to a *List*, is very inefficient. Hibernate will remove all records from the association table before it adds the remaining ones.

That's obviously not the most efficient approach. If you remove only one association from the List, you would expect that Hibernate only deletes the corresponding record from the association table and keeps all other records untouched. You can achieve that by mapping the association as a *java.util.Set*.

```
@Entity
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    @Version
    private int version;

    private String title;
    @ManyToMany
    private Set<Author> authors = new HashSet<Author>();

    ...

}
```

If you now remove an associated entity from the *Set*, Hibernate only executes the expected SQL DELETE statement.

```
06:09:32,412 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_1_0_,
        book0_.title as title2_1_0_,
        book0_.version as version3_1_0_
    from
        Book book0_
    where
        book0_.id=?
06:09:32,414 DEBUG [org.hibernate.SQL] -
    select
        authors0_.books_id as books_id1_2_0_,
        authors0_.authors_id as authors_2_2_0_,
        author1_.id as id1_0_1_,
        author1_.firstName as firstNam2_0_1_,
        author1_.lastName as lastName3_0_1_,
        author1_.version as version4_0_1_
    from
        Book_Author authors0_
    inner join
        Author author1_
            on authors0_.authors_id=author1_.id
    where
        authors0_.books_id=?
06:09:32,417 DEBUG [org.hibernate.SQL] -
    update
        Book
    set
        title=?,
        version=?
    where
        id=?
        and version=?
```

```
06:09:32,420 DEBUG [org.hibernate.SQL] -
    delete
    from
        Book_Author
    where
        books_id=?
        and authors_id=?
```

3. Don't use bidirectional One-to-One mappings

I briefly mentioned lazy loading of one-to-one associations in the first section. But it's important and tricky enough to get into more details on it.

For all managed associations, you can use the *fetch* attribute of the defining annotation to set the *FetchType*. But even though that includes the *@OneToOne* annotation, that mapping is a little bit special. That's because it's the only relationship for which you can define a to-one association on the entity that doesn't map the foreign key column.

If you do that, Hibernate needs to perform a query to check if it has to initialize the attribute with *null* or a proxy object. And the Hibernate team decided, that if they have to execute a query anyways, it's better to fetch the associated entity instead of just checking if it exists and fetching it later. Due to that, lazy loading doesn't work for this kind of one-to-one association mapping. But it works perfectly fine on the entity that maps the foreign key column.

So, what should you do instead?

You should only model unidirectional one-to-one associations that share the same primary key value on the entity that maps the foreign key column. Bidirectional and unidirectional associations on the entity that doesn't model the foreign key column don't support any lazy fetching.

Modeling a unidirectional one-to-one association with a shared primary key value is pretty simple. You just need to annotate the association with an additional @MapsId annotation. That tells your persistence provider to use the primary key value of the associated entity as the primary key value of this entity.

```
@Entity
public class Manuscript {

    @Id
    private Long id;

    @OneToOne
    @MapsId
    @JoinColumn(name = "id")
    private Book book;

    ...
}
```

Due to the shared primary key value, you don't need a bidirectional association mapping. When you know the primary key value of a *Book* entity, you also know the primary key value of the associated *Manuscript* entity. So, you can simply call the *find* method on your *EntityManager* to fetch the *Manuscript* entity.

```
Book b = em.find(Book.class, 100L);
Manuscript m = em.find(Manuscript.class, b.getId());
```

4. Avoid the @Formula annotation

The *@Formula* annotation enables you to map the return value of an SQL snippet to a read-only entity attribute. It's an interesting feature that you can use in smaller applications that don't need to handle lots of parallel requests. But it's not a great fit for a high-performance persistence layer.

Here you can see an example of the *@Formula* annotation. I use it to calculate the *age* of an *Author* based on her/his date of birth.

```
@Entity
public class Author {

    @Id
    @GeneratedValue
    private Long id;

    @Version
    private int version;

    private String firstName;

    private String lastName;

    private LocalDate dateOfBirth;

    @Formula(value = "date_part('year', age(dateOfBirth))")
    private int age;

    ...
}
```

The main issue with the *@Formula* annotation is that the provided SQL snippet gets executed every time you fetch the entity. But I have never seen an application that used the read-only attributes every time the entity got fetched.

```
06:16:30,054 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_0_,
        author0_.dateOfBirth as dateOfBi2_0_0_,
        author0_.firstName as firstNam3_0_0_,
        author0_.lastName as lastName4_0_0_,
        author0_.version as version5_0_0_,
        date_part('year',
        age(author0_.dateOfBirth)) as formula0_0_
    from
        Author author0_
    where
        author0_.id=?
```

In a smaller application, that isn't an issue. Your database can easily execute the more complex SQL statement. But in a high-performance persistence layer that needs to handle lots of parallel requests, you should avoid any unnecessary complexity. In these cases, you can better call a database function and use a DTO projection.

5. Don't use the @OrderBy annotation

My recommendation for the *@OrderBy* annotation is basically the same as for the *@Formula* annotation: It's a great feature for smaller applications but not a great fit for a high-performance persistence layer.

```
@Entity
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    @Version
    private int version;

    private String title;

    @ManyToMany
    @OrderBy(value = "lastName ASC, firstName ASC")
    private Set<Author> authors = new HashSet<Author>();

    ...
}
```

Using the *@OrderBy* annotation, you can define an *ORDER BY* clause that gets used when Hibernate fetches the associated entities. But not all of your use cases will need to retrieve the association in a specific order. If you don't need it, the ordering creates an overhead that you should avoid, if you need to optimize your persistence layer for performance.

If performance is more important than ease of use of your persistence layer, you should prefer a use case specific JPQL query. By doing that, you can add the ORDER BY clause whenever you need it. In all other use cases, you can fetch the associated entities in an undefined order.

6. Avoid CascadeType.REMOVE for large associations

Cascading tells Hibernate to perform an operation not only on the entity on which you triggered it but also on the associated entities. That makes persist, merge, and remove operations much easier.

But using *CascadeType*.REMOVE on a large association is very inefficient. It requires Hibernate to fetch all associated entities, to change the life cycle state of each entity to removed and execute an SQL DELETE statement for each of them. Doing that for a few dozen or more entities can take a considerable amount of time.

Using a *CriteriaDelete* or a JPQL DELETE statement enables you to remove all associated entities with one statement. That's avoids the life cycle state transitions and drastically reduces the number of executed queries. So, it shouldn't be a surprise that it's also much faster.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaDelete<Book> delete = cb.createCriteriaDelete(Book.class);
Root<Book> book = delete.from(Book.class);
ParameterExpression<Author> p = cb.parameter(Author.class, "author");
delete.where(cb.isMember(p, book.get(Book_.authors)));

Query query = em.createQuery(delete);
query.setParameter(p, em.find(Author.class, 8L));
query.executeUpdate();
```

But please keep in mind, that Hibernate doesn't trigger any life cycle events for these entities and that it doesn't remove entities in your 1st level cache.

Conclusion

Hibernate provides lots of mapping features that can make implementing and using your persistence layer much easier. But not all of them are an excellent fit for a high-performance persistence layer.

In general, you should avoid all mappings that are not required for every use case or that make your mapping more complex. 2 typical examples for that are the *@Formula* and the *@OrderBy* annotations.

In addition to that, you should always monitor the executed SQL statements. It should be evident that the fewer queries your use cases require, the faster they are. So, make sure that Hibernate uses your mappings efficiently.