Mistake 1: Use Eager Fetching

The *FetchType* defines when Hibernate initializes an association. You can specify with the fetch attribute of the *@OneToMany*, *@ManyToOne*, *@ManyToMany* and *@OneToOne* annotation.

```
@ManyToMany(mappedBy="authors",
        fetch=FetchType.LAZY)
private List<Book> books = new ArrayList<Book>();
```

Hibernate loads eagerly fetched associations when it loads an entity. E.g., when Hibernate loads a *Author* entity, it also fetches the associated *Book* entity. That requires an additional query for each *Author* and often accounts for dozens or even hundreds of additional queries.

Better use *FetchType.LAZY* instead. It delays the initialization of the relationship until you use it in your business code.

Mistake 2: Ignore the Default *FetchType* of To-One Associations

The next thing you need to do to prevent eager fetching is to change the default *FetchType* for all to-one associations. Unfortunately, these relationships are eagerly fetched by default.

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

Mistake 3: Don’t Initialize Required Associations

When you use *FetchType.LAZY* for all of your associations to avoid mistake 1 and 2, you will find several n+1 select issues in your code. This problem occurs when Hibernate performs 1 query to select n entities and then has to perform an additional query for each of them to initialize a lazily fetched association.

You can easily avoid that, when you tell Hibernate to initialize the required association. There are several different ways to do that. The easiest one is to add a JOIN FETCH statement to your FROM clause.

```
Author a = em.createQuery("SELECT a FROM Author a "
        + "JOIN FETCH a.books WHERE a.id = 1",
        Author.class).getSingleResult();
```

Mistake 4: Select More Records Than You Need

JPQL doesn’t support the *OFFSET* and *LIMIT* keywords you use in your SQL query. It might seem like you can’t limit the number of records retrieved within a query. But you can, of course, do that. You just need to set this information on the *Query* interface and not in the JPQL statement.

```
List<Author> authors = em.createQuery(
                "SELECT a FROM Author a ORDER BY a.id ASC",
                Author.class)
        .setMaxResults(5)
        .setFirstResult(0)
        .getResultList();
```

Mistake 5: Don’t use Bind Parameters

Most applications execute a lot of the same queries which just use a different set of parameter values in the WHERE clause. Bind parameters allow Hibernate and your database to identify and optimize these queries.

You can use named bind parameters in your JPQL statements. Each named parameter starts with a “:” followed by its name. After you’ve defined a bind parameter in your query, you need to call the setParameter method on the *Query* interface to set the bind parameter value.

```
TypedQuery<Author> q = em.createQuery(
        "SELECT a FROM Author a WHERE a.id = :id",
        Author.class);
q.setParameter("id", 1L);
Author a = q.getSingleResult();
```

Mistake 6: Perform all Logic in Your Business Code

Sometimes, the database is the better place to implement logic that operates on a lot of data. You can do that by calling a function in your JPQL or SQL query or with a stored procedure.

You can use standard functions in your JPQL queries in the same way as you call them in an SQL query.

```
Query q = em.createQuery("SELECT a, size(a.books) "
        + "FROM Author a GROUP BY a.id");
List<Object[]> results = q.getResultList();
```

And with JPA's function *function*, you can also call database-specific or custom database functions.

```
TypedQuery<Book> q = em.createQuery("SELECT b "
                + "FROM Book b "
                + "WHERE b.id = function('calculate', 1, 2)",
        Book.class);
```

Mistake 7: Call the *flush* Method Without a Reason

Hibernate stores all managed entities in the persistence context and tries to delay the execution of write operations as long as possible. That allows Hibernate to combine multiple update operations on the same entity into 1 SQL UPDATE statement, to bundle multiple identical SQL statements via JDBC batching and to avoid the execution of duplicate SQL statements that return an entity which you already used in your current *Session*.

A call of the *flush* method forces Hibernate to perform a dirty check on all managed entities and to create and execute SQL statements for all pending insert, update or delete operations.

That prevents Hibernate from using its internal optimizations and slows down your application.

Mistake 8: Use Hibernate for Everything

Hibernate makes the implementation of most CRUD use cases very easy and efficient. That makes Hibernate a popular and good choice for a lot of projects.

But that doesn't mean that it's a good solution for all kinds of projects. V very complex queries, analysis or reporting use cases or write operations on a huge number of records are not a good fit for JPA's and Hibernate's query capabilities and the lifecycle based entity management.

You can still use Hibernate if these use cases are just a small part of your application. But in general, you should take a look at other frameworks, like jOOQ or Querydsl, which are closer to SQL and avoid any object relational mappings.

Mistake 9: Update or Delete Huge Lists of Entities One by One

Updating or deleting entities one by one becomes very inefficient when you have to do that for several hundreds or thousands of entities. It’s faster to use a a JPQL, native SQL or Criteria query.

But it has a few side-effects that you should be aware of. You perform the update or delete operation in your database without using your entities. That provides the better performance but it also ignores the entity lifecycle and Hibernate can’t update any caches.

To make it short, you shouldn’t use any lifecycle listeners and call the flush and clear methods on your EntityManager before you execute a bulk update. The flush method will force Hibernate to write all pending changes to the database before the clear method detaches all entities from the current persistence context.

```
em.flush();
em.clear();
Query query = em.createQuery(
        "UPDATE Book b SET b.price = b.price*1.1");
query.executeUpdate();
```

Mistake 10: Use Entities for Read-Only Operations

JPA and Hibernate support several different projections. You should make use of that if you want to optimize your application for performance. The most obvious reason is that you should only select the data that you need in your use case.

But that's not the only reason. As I showed in a recent test, DTO projections are a lot faster than entities, even if you read the same database columns.