1. Use a projection that fits your use case

JPA and Hibernate support more projections than just entities. There are 3 different kinds of them, and each one has its advantages and disadvantages:

1.1 Entities

You should use the entity projection when you need all attributes of the entity and for update or delete operations that affect only a small number of entities.

```
em.find(Author.class, 1L);
```

1.2 POJOs

POJO projections allow you to create use case specific representations of the database record. This is especially useful if you only need a small subset of the entity attributes or if you need attributes from several related entities.

```
List<BookPublisherValue> bookPublisher = em.createQuery(
        "SELECT new "
        + "org.thoughts.on.java.model.BookPublisherValue("
        + "b.title, b.publisher.name) FROM Book b",
        BookPublisherValue.class).getResultList();
```

1.3 Scalar Values

Scalar value projections present the values as *Object[]*s. You should only use them if you want to select a small number of attributes and directly process them in your business logic.

```
List<Object[]> authorNames = em.createQuery(
        "SELECT a.firstName, a.lastName FROM Author a")
        .getResultList();
```

2. Use the kind of query that fits your use case

JPA and Hibernate offer multiple implicit and explicit options to define a query. None of them is a good fit for every use case, and you should, therefore, make sure to select the one that fits best.

2.1 *EntityManager.find()*

The *EntityManager.find()* method is the best and easiest way to get an entity by its primary key.

```
em.find(Author.class, 1L);
```

2.2 JPQL

JPQL is very similar to SQL, but it operates on entities and their relationships instead of database tables. You can use it to create queries of low and moderate complexity.

```
TypedQuery<Author> q = em.createQuery(
      "SELECT a FROM Author a JOIN a.books b "
            + "WHERE b.title = :title",
      Author.class);
```

2.3 Criteria API

The Criteria API provides you an easy to API to dynamically define queries at runtime.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Author> q = cb.createQuery(Author.class);
Root<Author> author = q.from(Author.class);
q.select(author);
```

2.4 Native Queries

Native queries provide you the chance to write and execute plain SQL statements.

```
MyEntity e = (MyEntity) em.createNativeQuery(
        "SELECT * FROM myentity e "
                + "WHERE e.jsonproperty->'longProp' = '456'",
        MyEntity.class).getSingleResult();
```

I explain native queries in more detail in Native Queries – How to call native SQL queries with JPA and How to use native queries to perform bulk updates.

3. Use bind parameters

You should use parameter bindings for your query parameters instead of adding the values directly to the query String. This provides several advantages:

- you do not need to worry about SQL injection,
- Hibernate maps your query parameters to the correct types and
- Hibernate can do internal optimizations to provide better performance.

4. Use static *String*s for named queries and parameter names

This is just a small thing but it's much easier to work with named queries and their parameters if you define their names as static *String*s. I prefer to define them as attributes of the entities with which you can use them but you can also create a class that holds all query and parameter names.

```
@NamedQuery(
        name = Author.QUERY_FIND_BY_LAST_NAME,
        query = "SELECT a FROM Author a "
                + "WHERE a.lastName = :"
                + Author.PARAM_LAST_NAME)
@Entity
public class Author {


        public static final String
                QUERY_FIND_BY_LAST_NAME =
                        "Author.findByLastName";
        public static final String PARAM_LAST_NAME =
                        "lastName";
```

5. Use JPA Metamodel when working with Criteria API

The JPA Metamodel generates an additional class for each entity. You can use it to reference entity attributes in a type safe way when creating a Criteria query.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Author> q = cb.createQuery(Author.class);
Root<Author> author = q.from(Author.class);
q.select(author);
q.where(cb.equal(author.get(Author_.lastName), lastName));
```

I explain the JPA Metamodel and how you can generate its classes in Create type-safe queries with the JPA static metamodel.

6. Use surrogate keys and let Hibernate generate new values

The main advantage of a surrogate primary key (or technical ID) is that it is one simple number and that all involved systems can handle it very efficiently. Hibernate can also use existing database features, like sequences or auto-incremented columns, to generate unique values for new entities.

```
@Id
@GeneratedValue
@Column(name = “id”, updatable = false, nullable = false)
private Long id;
```

7. Specify natural identifier

You should specify natural identifiers, even if you decide to use a surrogate key as your primary key. A natural identifier nevertheless identifies a database record and an object in the real world. A lot of use cases use them instead of an artificial, surrogate key.

It is, therefore, good practice to model them as unique keys in your database. Hibernate also allows you to model them as a natural identifier of an entity and provides an extra API for retrieving them from the database.

The only thing you have to do to model an attribute is a natural id, is to annotate it with *@NaturalId*.

```
@Entity
public class Book {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @Column(name = “id”, updatable = false, nullable = false)
  private Long id;

  @NaturalId
  private String isbn;

  …
}
```

You can read more about natural identifiers and Hibernate's proprietary API in How to map natural IDs with Hibernate.

8. Use SQL scripts to create the database schema

The database schema has a huge influence on the performance and size of your database. You, therefore, should design and optimize the database schema yourself and export it as an SQL script.

The following snippet shows a persistence.xml file which tells Hibernate to run the create.sql script to setup the database.

```
<persistence>
  <persistence-unit name=”my-persistence-unit” transaction-type=”JTA”>

    <properties>
      <property name=
        ”javax.persistence.schema-generation.scripts.action”
        value=”create”/>
      <property name=
        ”javax.persistence.schema-generation.scripts.create-target”
        value=”./create.sql”/>
    </properties>
  </persistence-unit>
```

You can learn more about the different configuration parameters in Standardized schema generation and data loading with JPA 2.1.

9. Log and analyze all queries during development

Hibernate hides all database interactions behind its API, and it's often difficult to guess how many queries it will perform for a given use case. The best way to handle this issue is to log all SQL statements during development and analyze them before you finish your implementation task. You can do that by setting the log level of the *org.hibernate.SQL* category to *DEBUG*.

I explain Hibernate's most important log categories and provide detailed recommendations for a development and a production configuration in my Hibernate Logging Guide.

10. Don't use FetchType.EAGER for to-many relationships

Eager fetching is another common reason for Hibernate performance issues. It tells Hibernate to initialize a relationship when it fetches an entity from the database.

The main issue is, that Hibernate will fetch the related entities whether or not they are required for the given use case. That creates an overhead which slows down the application and often causes performance problems.

You should use *FetchType.LAZY* instead and fetch the related entities only if you need them for your use case.

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

11. Initialize required lazy relationships with the initial query

FetchType.LAZY tells Hibernate to fetch the related entities only when they're used. This helps you to avoid certain performance issues. But it's also the reason for the *LazyInitializationException* and the n+1 select issue which occurs when Hibernate has to perform an additional query to initialize a relationship for each of the selected n entities.

The best way to avoid both issues is to fetch an entity together with the relationships you need for your use case. One option to do that is to use a JPQL query with a *JOIN FETCH* statement.

```
List<Author> authors = em.createQuery(
    "SELECT DISTINCT a "
        + "FROM Author a JOIN FETCH a.books b",
    Author.class).getResultList();
```

I explain several other options and their benefits in 5 ways to initialize lazy relationships and when to use them.

12. Avoid cascade remove for huge relationships

CascadeType.REMOVE tells Hibernate to also delete the related entities when it deletes this one. This makes it often difficult to understand what exactly happens if you delete an entity. And that's something you should always avoid.

If you have a closer look at how Hibernate deletes the related entities, you will find another reason to avoid it. Hibernate performs 2 SQL statements for each related entity: 1 SELECT statement to fetch the entity from the database and 1 DELETE statement to remove it. This might be OK, if there are only 1 or 2 related entities but creates performance issues if there are large numbers of them.

13. Use @Immutable when possible

Hibernate regularly performs dirty checks on all entities that are associated with the current *PersistenceContext* to detect required database updates. This is a great thing for all mutable entities. But not all entities have to be mutable.

Entities can also map read-only database views or tables. Performing any dirty checks on these entities is an overhead that you should avoid.

You can do this by annotating the entity with *@Immutable*. Hibernate will then ignore it in all dirty checks and will not write any changes to the database.

```
@Entity
@Immutable
public class BookView {

    …

}
```