The most efficient way to improve the performance of a database query is to avoid it by retrieving the data from a local cache. That's why Hibernate offers 3 different caches:

- The 1st level cache contains all entities loaded and created within the current session.
- The 2nd level cache is a shared, session-independent cache for entities.
- The query cache is a shared, session-independent cache for the results of Criteria and JPQL queries.

In this article, I will focus on the query cache. It's Hibernate-specific and the only one that doesn't store entities. That makes it your only option if you want to cache DTO projections.

How to Activate Hibernate's Query Cache

To use Hibernate's query cache you first need to activate it in your persistence.xml file. In the next step you need to explicitly activate caching for your query.

This 2-step activation is necessary because most of your queries are not good candidates for caching.

You should only cache the result of a query, which you often call with the same set of bind parameter values. In addition to that, the data on which you perform your query should only rarely change. Otherwise, your query cache will not be very effective. It will spend more time adding and removing entries than actually returning cached query results.

How to Activate Your Query Cache

You can activate the query cache by setting the hibernate.cache.use_query_cache parameter in your persistence.xml to true. And if you want to use the query cache for any query that returns entities, you should also make sure to configure the 2nd level cache for these entities. But that's a topic for

another article, and I explain it in great details in the Hibernate Performance Tuning Online Training.

```
<persistence>
    <persistence-unit name="my-persistence-unit">
        <description>Hibernate Performance Tuning</description>
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <!--  enable selective 2nd level cache -->
        <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
        <properties>
            ...

            <!--  configure caching -->
            <property name="hibernate.cache.use_query_cache" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

How to Activate Caching for a Query

After you activated the query cache, you need to tell Hibernate to cache the result of the query. You can do that in 2 ways:

1. If you're using JPA's Query interface, you can set the query hint QueryHints.CACHEABLE to true.
2. If you're using Hibernate's Query interface, you can call the setCacheable method with true.

Both options create the same result. Hibernate checks the query cache before it executes the query. If the cache contains the result, Hibernate returns it without performing the query. If the result isn't cached, Hibernate executes the query and stores the result in the query cache.

The following query uses a constructor expression and selects the title attribute of the Book entity and the name attribute of the Publisher entity. For each record returned by this query, Hibernate calls the constructor of the BookPublisherValue class. In this example, I use JPA's Query interface and activate the query cache with a query hint.

```
TypedQuery<BookPublisherValue> q = em
        .createQuery(
                "SELECT new org.thoughts.on.java.model.BookPublisherValue(b.title, p.name) FROM Book b JOIN b.publisher p WHERE b.id = :id",
                BookPublisherValue.class);
q.setHint(QueryHints.CACHEABLE, true);
q.setParameter("id", 1L);
BookPublisherValue value = q.getSingleResult();
```

If you activate the Hibernate statistics and the logging of SQL statements, you can see that Hibernate executes the query and stores the result in the cache.

```
19:28:04,853 INFO [org.hibernate.engine.internal.StatisticalLoggingSessionEventListener] - Session Metrics {
    28300 nanoseconds spent acquiring 1 JDBC connections;
    27201 nanoseconds spent releasing 1 JDBC connections;
    307300 nanoseconds spent preparing 1 JDBC statements;
    1204200 nanoseconds spent executing 1 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    3333200 nanoseconds spent performing 1 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    1897000 nanoseconds spent performing 1 L2C misses;
    0 nanoseconds spent executing 0 flushes (flushing a total of 0 entities and 0 collections);
    78800 nanoseconds spent executing 1 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```