Hi,

I am Thorben, the author of thoughts-on-java.org, and I want to thank you for signing up and downloading the “Hibernate Performance Tuning” cheat sheet!

The most important step when fixing performance issues is always to recognize their existence and their cause. In my experience, Hibernate Statistics are most often very helpful for this. Therefore activating and analyzing the statistics is always a good thing to start and also the first part of this cheat sheet.

After that you obviously need to find and fix the cause of the performance issue. The most common ones are:

- Slow queries,
- Using the wrong FetchType,
- Loading the same data multiple times and
- Updating one entity after the other instead of using bulk operations.

This cheat sheet gives you an overview about some of the available solutions to these issues and links to more detailed information.

If you like to get in touch and discuss some performance tuning technics, write me an email at thorben@thoughts-on-java.org or tweet me at @thjanssen123.

Take care,

Thorben

## Find performance issues with Hibernate Statistics

Hibernate Statistics provide some very useful information when searching and analyzing performance issues, like the number of SQL queries performed within a Hibernate session, the time spend for these queries and how many entities were retrieved from the cache.

```
2015-03-03 20:28:52,484 DEBUG
[org.hibernate.stat.internal.ConcurrentStatisticsImpl] (default task-1)
HHH000117: HQL: Select p From Product p, time: 0ms, rows: 10
2015-03-03 20:28:52,484 INFO
[org.hibernate.engine.internal.StatisticalLoggingSessionEventListener]
(default task-1) Session Metrics {
   8728028 nanoseconds spent acquiring 12 JDBC connections;
   295527 nanoseconds spent releasing 12 JDBC connections;
   12014439 nanoseconds spent preparing 21 JDBC statements;
   5622686 nanoseconds spent executing 21 JDBC statements;
   0 nanoseconds spent executing 0 JDBC batches;
   0 nanoseconds spent performing 0 L2C puts;
   0 nanoseconds spent performing 0 L2C hits;
   0 nanoseconds spent performing 0 L2C misses;
   403863 nanoseconds spent executing 1 flushes (flushing a total of 10
entities and 0 collections);
   25529864 nanoseconds spent executing 1 partial-flushes (flushing a total of
10 entities and 10 collections)
}
```

The Hibernate Statistics need to be activated by setting the system property: `hibernate.generate_statistics = true` and activating `DEBUG` logging for `org.hibernate.stat`.
Activating the statistics can have a very huge performance impact and should not be done on any production system!

Read more: http://bit.ly/1QWS8Ik

## Improve slow queries

Slow queries are not a real JPA or Hibernate issue. These kind of problems occurs with every framework, even with plain SQL over JDBC and need to be analyzed on the SQL and database level.

Slow queries can be improved by:

- Analyzing the generated SQL,
- Checking the execution plan, indexes, etc.,
- Optimizing the queries based on this information.

Native SQL queries can be used, if JPQL does not provide the required feature set. The `Object[]` returned by native queries can be declaratively mapped with `@SqlResultSetMapping`.

Read more: http://bit.ly/1LHENA9

## Choose the right FetchType

The `FetchType` is specified in the entity mapping and defines when a relationship will be loaded. Using the wrong `FetchType` can result in a huge number of queries that are performed to load the required entities.

```
@ManyToMany( mappedBy="authors",
             fetch=FetchType.LAZY)
```

The main problem of the `FetchType` definition is, that you can only define one `FetchType` for a relationship, which will be used every time an entity gets fetched from the database.
The best solution is to use `FetchType.LAZY` for to-many relationships and specify eager loading for specific queries

## Use query specific fetching

If you require entities with initialized relationships, you should define this for the specific query instead of using `FetchType.EAGER`. This can be done:

- As a part of a JPL statement with `FETCH JOIN`,
```
SELECT DISTINCT a
FROM Author a JOIN FETCH a.books b
```

- Via annotations with `@NamedEntityGraph`,
```
@NamedEntityGraph(
     name = "graph.AuthorBooksReviews",

     attributeNodes = @NamedAttributeNode(
                              value = "books")
)
```
Read more: http://bit.ly/1GWjxEP

- Based on a Java API with `Entity Graph`.
```
EntityGraph graph =
     this.em.createEntityGraph(Author.class);
Subgraph<Book> bookSubGraph =
     graph.addSubgraph(Author_.books);
bookSubGraph.addSubgraph(Book_.reviews);
```
Read more: http://bit.ly/1MUMRSp

## Let the database handle data heavy operations

The database can handle huge datasets very efficiently and it can be more efficient to perform some operations within the database instead of the Java application.

Simple operations can be performed within a JPQL or native SQL query.
If you need more complex operations, you can call stored procedures. These queries can be defined:

- Via annotations with `@NamedStoredProcedureQuery`,

```
@NamedStoredProcedureQuery(name = "getBooks",
                  procedureName = "get_books",
                  resultClasses = Book.class,
                  parameters          =          {
@StoredProcedureParameter(mode                   =
ParameterMode.REF_CURSOR, type = void.class) })
```

  Read more: http://bit.ly/1hWRIpn

- A Java API with `StoredProcedureQuery`

```
StoredProcedureQuery query =
    this.em.createStoredProcedureQuery
        ("get_books", Book.class);
query.registerStoredProcedureParameter(1,
    void.class, ParameterMode.REF_CURSOR);

query.execute();
```

  Read more: http://bit.ly/1OKxwWn

## Use caches to avoid repeatedly reading the same data

Modular applications and parallel user sessions often result in reading the same data multiple times. This data can be efficiently cached, if it does not change too often.

Hibernate offers 3 different kinds of caches:

- 1st level cache
  - This 1st level cache is activated by default and contains all entities that were used within the session.
- 2nd level cache
  - The second level cache also stores entities and is session independent.
  - It needs to be activated via the `shared-cache-mode` property in the persistence.xml
  - The caching of specific entities can be activated by adding the `javax.persistence.Cacheable` or the `org.hibernate.annotations.Cache` annotation to the entity.
- Query cache
  - The query caches stores query results and is session independent.
  - It does only store scalar values and entity references and should always be used together with the 2nd level cache.
  - It needs to be activated in the `persistence.xml` via the `hibernate.cache.use_query_cache` property and the `cacheable` property on the `Query`.

## Perform updates and deletes in bulks

Updating and deleting entities one by one can be very inefficient. SQL supports update and delete statements that affect multiple records at once.

`CriteriaUpdate` and `CriteriaDelete` can be used to update and delete multiple records at once.

```
CriteriaUpdate<Order> update =
            cb.createCriteriaUpdate(Order.class);
CriteriaDelete<Order> delete =
            cb.createCriteriaDelete(Order.class);
```

Read more: http://bit.ly/1AwX55x