**Hibernate Querying 101 : tips and tricks**

Table of Contents

* [Introduction](http://www.javalobby.org/articles/hibernate-query-101/#introduction)
* [Background](http://www.javalobby.org/articles/hibernate-query-101/#background)
* [Hibernate Query Methods](http://www.javalobby.org/articles/hibernate-query-101/#query)
* [More advanced techniques](http://www.javalobby.org/articles/hibernate-query-101/#advanced)
* [Sidenote : Unit testing](http://www.javalobby.org/articles/hibernate-query-101/#sidenote)
* [Conclusion](http://www.javalobby.org/articles/hibernate-query-101/#conclusion)

This article presents a few real-life tips and tricks for Hibernate querying. It is designed to be a 'from-the-trenches' account of the methods and best-practices we use and that we have found to work.

**Background**

First a little background. We are currently working on a project using a JSTL / Struts / Hibernate / PostgreSQL stack. The application requires many PDF reports. For the PDF (and Excel) report generation, we use JasperReports. Maybe I'll write another article on JasperReports some other time. But for now, I want to talk about the Hibernate queries.

The reports tend to be complicated. So are the queries. The database is big enough to be slow if you don't make your HQL queries slick and efficient. Why don't we use plain old SQL queries for the reports? Well, if we did that, we'd loose the abstraction level we get with Hibernate, and the queries would be much longer and more complicated. Nevertheless, in practice, you do have to fine-tune each query individally in order to get the best performance, just as you would a plain old SQL query.

**Hibernate Query Methods**

The queries we general use can be divided up into three main categories :

1. Simple HQL queries using one class (where no joins are required)
2. HQL queries which require joins over several classes
3. HQL queries which cannot be efficiently done using joins

**Simple HQL queries**

The first and simplest query is the plain old HQL query on one object, as you can find in all the Hibernate introductory examples :

from Sale sale where sale.date > :startDate

This type of query is not really the subject of this article. If it works, fine. However, there are often cases where this type of query will pose problems for inexperienced Hiberate programmers.

**Queries using joins**

Sometimes, a Hibernate query will take an excessive amount of time to execute. A Hibernate query which takes more than a few hundred milliseconds to execute should be immediately considered with suspicion. Under the hood (looking at the log files), the typical symptom will often be an excessive number of SQL queries (potentially as many as *Kn + 1*, where *n* is the number of objects returned). This means that for each object that Hibernate loads, it needs to do one or more extra SQL queries to load associated objects. This is generally regarded as a bad thing.

Consider the following (simplified) Hibernate mappings :

<class name="Sale" dynamic-update="true" table="t\_sale" >

<id column="sale\_ch\_id" name="id" unsaved-value="null"

type="string">

<generator class="uuid.hex"/>

</id>

<property column="sale\_d\_date" name="date" type="date"

not-null="true"/>

<property column="sale\_n\_price" name="price"

type="big\_decimal" not-null="true"/>

<many-to-one name="product" class="Product"

column="prod\_ch\_id".../>

...

</class>

Now lets go back to the previous query. This query will generate (at least) one SQL query for the initial select,

select sale\_ch\_id, sale\_d\_date, sale\_n\_price...from t\_sale...

and then a sequence of selects from the associated tables :

select prod\_ch\_id...from t\_product...

select prod\_ch\_id...from t\_product...

select prod\_ch\_id...from t\_product...

...

This problem is well-known in Hibernate circles, and the solution is straight-forward once you know it. The associated objects must be loaded in the initial query, for example by using the 'left join fetch' construction :

from Sale sale

where sale.date > :startDate

left join fetch sale.product

This usually works well for classes with only a handful of associated objects. In practice, there may be an issue in cases where there are a *lot* of associated tables, and where many of the associated tables contain only limited amounts of static data.

from Sale sale

where sale.date > :startDate

left join fetch sale.salesman

left join fetch sale.product

left join fetch sale.product.color

left join fetch sale.product.category

...

Notice how we may need to perform left joins on second-level associations (eg. sale.product.color). But do we really need to do left joins on the *whole* object tree from Sale down ? What are the pros and cons ?

In practice, if you want to be sure of having a unique SQL query, you will potentially need left joins on all directly and indirectly associated objects (I'm not talking about collections, which should use lazy-loading, and which are not the subject of this discussion). However, if you have a lot of objects to load, you may have to be more selective about what you fetch, and about how big your SQL query gets.

For example, if your Color table contains 10 lines, omitting the join on sale.product.color will result in (at most) 10 extra SQL queries, bringing the total to 11. Once all the Color instances are loaded into the Hibernate session cache, you can trust Hibernate not to do any extra queries.

Bigger queries return more data, and when large data sets are involved, it is quite possible that the overhead of the weighty SQL query outweighs the overhead of reading a few small tables into memory and caching the results. In practice, these smaller tables should probably be cached using the second-level hibernate caching anyway. The bottom line is that the query structure should be designed and optimised via unit testing and concrete metrics (System.currentTimeMillis() will do fine).

**Cases where joined queries aren't enough**

Now suppose we have complex, multiple, bi-directional relations between two table. For example, suppose we have two classes : Company and City. A company is located in a city. A city outsources different types of work (water installation management, electricity management, etc.) to different companies. Each company is in turn located in a city.

In this situation, the presence of many-to-one relations means that Hibernate will potentially generate an uncontrollable number of SQL queries, and left join fetching is helpless to resolve the problem : you would end up with a query looking something like this :

from company c

left join fetch c.city

left join fetch c.city.waterCompany

left join fetch c.city.electrityCompany

left join fetch c.city.waterCompany.city

left join fetch c.city.electrityCompany.city

etc...

In this case, you basically have two main options :

1. Let the client wait half an hour for his reports
2. Use an optimised HQL query to return only the columns you really need.

Generally, the first option isn't greatly appreciated by the client. So lets look at the second option. This approach involves determining exactly which columns you really need, and instanciating data-transfer JavaBean objects containing exactly those columns.

select new CityItem(city.id, city.name, city.electrityCompany.name)

from City city

...

This technique is fast and efficient, and avoids the overheads of handling associated objects and of loading large numbers of persistent objects into the Hibernate session cache. The only downside is the need to create a dedicated data-transfer class for each query.

**More advanced techniques**

Finally, I will present a variation on the previous technique, which avoids the need to create a new data-transfer class for each query. The idea is to retrieve a list of instances of a Hibernate-persisted business class, using a given HQL query, but with only a specified set of columns being instanciated. This allows fast, light-weight queries which return only the minimum necessary information, and avoid complex joins in the HQL queries. For example :

String query =

"select com.id, com.label, com.postCode, com.mayor.name

from Community as com

left join com.mayor

order by com.label ";

results = HibernateUtils.find(query);

or

String query =

"select com.id, com.label, com.postCode, com.mayor.name

from Community as com

left join com.mayor

where com.label like ?

order by com.label ";

results = HibernateUtils.find(query,

new Object[] {name + "%"},

new Type [] {Hibernate.STRING});

This query will return a list of Community objects, but with only the 4 specified fields instanciated. Note that this is a powerful technique, as the presentation layer may use ordinary business classes instead of data transfer objects, without having to know the details of the querying techniques being used. This method should be reserved for pure read-only/display actions, however, as data manipulation with incompletely-instanciated objects is generally a very bad idea.

The implementation of such a method is fairly simple, and based on introspection and the Apache BeanUtils classes. For the curious, here is a simplified version of the implementation of the first method (error handling and optimised caching code has been removed for simplicity) :

public List find(final String hqlQuery) throws Exception {

List results = new ArrayList();

//

// Prepare a Hibernate query

//

Query query = SessionManager.currentSession().createQuery(hqlQuery);

//

// Determine the return type for this query

//

Type beanType = query.getReturnTypes()[0];

Class beanClass = beanType.getReturnedClass();

//

// Extract the list of columns returned by this query

//

String[] columns = extractColumns(hqlQuery);

//

// Pre-process bean attribute names, stripping spaces 'as' clauses

//

String[] attributeNames = getAttributeFieldNames(columns);

//

// Pre-process result field names, stripping spaces and retaining

// alias field names instead of the original column names where necessary

//

String[] resultFieldNames = getResultFieldNames(columns);

//

// Execute query and build result list

//

Iterator iter = query.iterate();

while(iter.hasNext()) {

Object[] row = (Object[]) iter.next();

Object bean = beanClass.newInstance();

for (int j = 0; j < row.length; j++) {

if (row[j] != null) {

initialisePath(bean, attributeNames[j]);

PropertyUtils.setProperty(bean, attributeNames[j], row[j]);

}

}

results.add(bean);

}

return results;

}

private static void initialisePath(final Object bean,

final String fieldName)

throws Exception {

int dot = fieldName.indexOf('.');

while (dot >= 0) {

String attributeName = fieldName.substring(0, dot);

Class attributeClass = PropertyUtils.getPropertyType(bean, attributeName);

if (PropertyUtils.getProperty(bean, attributeName) == null) {

PropertyUtils.setProperty(bean, attributeName, attributeClass.newInstance());

}

dot = fieldName.indexOf('.', dot + 1);

}

}

The implementation of the version using parameters is left as an exercise to the reader.

**Sidenote : Unit testing**

There are plenty of good articles on the merits of unit testing and of test-driven development, so I won't write much about this here. But, a word for the wise : in my experience, you cannot efficently code Hibernate queries without using a test-driven approach. OK, maybe I exaggerate a little. But the fact is that it's a lot easier to fine-tune your Hibernate queries against a local database, using timers and/or by studying the generated SQL queries, than if you wait until the solution is deployed in a WAR and the page takes 5 minutes to appear.

There is also much debate on how to test the Hibernate layers (hibernate mappings and DAOs). In our case, we are lucky enough to have a test database containing legacy data. We use a dedicated test database, which is also used for the test server.

1. Unit tests *must not* change the state of the database (so any created objects should be removed in the tearDown() method)
2. Unit tests may use existing test data (useful to test real cases), but only for read-only query operations (existing data must not be modified). It is the developers responsibility to only use stable existing test data, and to update the test cases if necessary.

**Conclusion**

Hibernate is a powerful object/relational persistence library, and it has an equally powerful query mechanism. However, the power of Hibernate does not dispense the developer from optimising the HQL queries. Indeed, the very simplicity of HQL can sometimes hide serious performance issues for the unwary. However, when well tuned, and using appropriate querying techniques, Hibernate can result in cleaner, simpler code as well as performance equaling that of an optimised SQL query via JDBC.