Hibernate Performance Tuning

Fetching strategies

A *fetching strategy* is the strategy Hibernate will use for retrieving associated objects if the application needs to navigate the association. Fetch strategies may be declared in the O/R mapping metadata, or over-ridden by a particular HQL or Criteria query.

Hibernate3 defines the following fetching strategies:

- ***Join fetching*** - Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.
- ***Select fetching*** - a second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying *lazy="false"*, this second select will only be executed when you actually access the association.
- ***Subselect fetching*** - a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you actually access the association.
- ***Batch fetching*** - an optimization strategy for select fetching - Hibernate retrieves a batch of entity instances or collections in a single SELECT, by specifying a list of primary keys or foreign keys.

Hibernate also distinguishes between:

- ***Immediate fetching*** - an association, collection or attribute is fetched immediately, when the owner is loaded.
- ***Lazy collection fetching*** - a collection is fetched when the application invokes an operation upon that collection. (This is the default for collections.)
- ***"Extra-lazy" collection fetching*** - individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed (suitable for very large collections)
- ***Proxy fetching*** - a single-valued association is fetched when a method other than the identifier getter is invoked upon the associated object.
- ***"No-proxy" fetching*** - a single-valued association is fetched when the instance variable is accessed. Compared to proxy fetching, this approach is less lazy (the association is fetched even when only the identifier is accessed) but more transparent, since no proxy is visible to the application. This approach requires buildtime bytecode instrumentation and is rarely necessary.
- ***Lazy attribute fetching*** - an attribute or single valued association is fetched when the instance variable is accessed. This approach requires buildtime bytecode instrumentation and is rarely necessary.

Solving the n+1 selects problem

The biggest performance killer in applications that persist objects to SQL databases is the *n+1 selects problem*. When you tune the performance of a Hibernate application, this problem is the first thing you'll usually need to address. Its normal (and recommended) to map almost all associations for lazy initialization.

This means you generally set all collections to *lazy="true"* and even change some of the one-to-one and many-to-one associations to not use outer joins by default. This is the only way to avoid retrieving all objects in the database in every transaction. Unfortunately, this decision exposes you to the n+1 selects problem.

It's easy to understand this problem by considering a simple query that retrieves all Items for a particular user:

```
Iterator items = session.createCriteria(Item.class)
.add( Expression.eq("item.seller", user) )
.list()
.iterator();
```

This query returns a list of items, where each collection of bids is an uninitialized collection wrapper. Suppose that we now wish to find the maximum bid for each item. The following code would be one way to do this:

```
List maxAmounts = new ArrayList();
while (items.hasNext()) {
Item item = (Item) items.next();
BigDecimal maxAmount = new BigDecimal("0");
for ( Iterator b = item.getBids().iterator(); b.hasNext(); ) {
Bid bid = (Bid) b.next();
if ( bid.getAmount().compareTo(maxAmount) == 1 )
maxAmount = bid.getAmount();
}
maxAmounts.add( new MaxAmount( item.getId(), maxAmount ) );
}
```

But there is a huge problem with this solution (aside from the fact that this would be much better executed in the database using aggregation functions):

Each time we access the collection of bids, Hibernate must fetch this lazy collection from the database for each item. If the initial query returns 20 items, the entire transaction requires 1 initial select that retrieves the items plus 20 additional selects to load the bids collections of each item. This might easily result in unacceptable latency in a system that accesses the database across a network. Usually you don't explicitly create such operations, because you should quickly see doing so is suboptimal. However, the n+1 selects problem is often hidden in more complex application logic, and you may not recognize it by looking at a single routine.

Batch fetching

The first attempt to solve this problem might be to enable *batch fetching.* We change our mapping for the bids collection to look like this:

```
<set name="bids" lazy="true" inverse="true" batch-size="10">
```

With batch fetching enabled, Hibernate pre-fetches the next 10 collections when the first collection is accessed. This reduces the problem from n+1 selects to n/10 + 1 selects. For many applications, this may be sufficient to achieve acceptable latency. On the other hand, it also means that in some other transactions, collections are fetched unnecessarily. It isn't the best we can do in terms of reducing the number of round trips to the database.

HQL aggregation

A much, much better solution is to take advantage of HQL aggregation and perform the work of calculating the maximum bid on the database. Thus we avoid the problem:

```
String query = "select MaxAmount( item.id, max(bid.amount) )" + " from Item item join item.bids bid" +
" where item.seller = :user group by item.id";
List maxAmounts = session.createQuery(query).setEntity("user", user).list();
```

Unfortunately, this isn't a complete solution to the generic issue. In general, we may need to do more complex processing on the bids than merely calculating the maximum amount. We'd prefer to do this processing in the Java application.

Eager fetching

We can try enabling eager fetching at the level of the mapping document:

```
<set name="bids" inverse="true" outer-join="true">
```

The outer-join attribute is available for collections and other associations. It forces Hibernate to load the association eagerly, using an SQL outer join.

Note that, as previously mentioned, HQL queries ignore the outer-join attribute; but we might be using a criteria query.

This mapping avoids the problem as far as this transaction is concerned; we're now able to load all bids in the initial select. Unfortunately, any other transaction that retrieves items using *get()*, *load()*, or a criteria query will also retrieve all the bids at once. Retrieving unnecessary data imposes extra load on both the database server and the application server and may also reduce the concurrency of the system, creating too many unnecessary read locks at the database level.

Hence we consider eager fetching at the level of the mapping file to be almost always a bad approach. The outer-join attribute of collection mappings is arguably a misfeature of Hibernate (fortunately, it's disabled by default). Occasionally it makes sense to enable outer-join for a *<many-to-one>* or *<one-to-one>* association (the default is auto), but we'd never do this in the case of a collection.

Runtime (code-level) declarations

The recommended solution for this problem is to take advantage of Hibernate's support for runtime (code-level) declarations of association fetching strategies. The example can be implemented like this:

```
List results = session.createCriteria(Item.class)
.add( Expression.eq("item.seller", user) )
.setFetchMode("bids", FetchMode.EAGER)
.list();
// Make results distinct
Iterator items = new HashSet(results).iterator();
List maxAmounts = new ArrayList();
for ( ; items.hasNext(); ) {
Item item = (Item) items.next();
BigDecimal maxAmount = new BigDecimal("0");
for ( Iterator b = item.getBids().iterator(); b.hasNext(); ) {
Bid bid = (Bid) b.next();
if ( bid.getAmount().compareTo(maxAmount) == 1 )
maxAmount = bid.getAmount();
}
maxAmounts.add( new MaxAmount( item.getId(), maxAmount ) );
}
```

We disabled batch fetching and eager fetching at the mapping level; the collection is lazy by default. Instead, we enable eager fetching for this query alone by calling *setFetchMode()*. As discussed earlier in this chapter, this is equivalent to a fetch join in the from clause of an HQL query.

The previous code example has one extra complication: The result list returned by the Hibernate criteria query isn't guaranteed to be distinct. In the case of a query that fetches a collection by outer join, it will contain duplicate items. It's the application's responsibility to make the results distinct if that is required. We implement this by adding the results to a HashSet and then iterating the set.

So, we have established a general solution to the n+1 selects problem. Rather than retrieving just the top-level objects in the initial query and then fetching needed associations as the application navigates the object graph, we follow a two step process:

1 Fetch all needed data in the initial query by specifying exactly which associations will be accessed in the following unit of work.

2 Navigate the object graph, which will consist entirely of objects that have already been fetched from the database.

This is the only true solution to the mismatch between the object-oriented world, where data is accessed by navigation, and the relational world, where data is accessed by joining.

Lazy fetching

By default, Hibernate3 uses lazy select fetching for collections and lazy proxy fetching for single-valued associations. These defaults make sense for almost all associations in almost all applications.

However, lazy fetching poses one problem that you must be aware of. Access to a lazy association outside of the context of an open Hibernate session will result in an exception.

Since the lazy collection was not initialized when the Session was closed, the collection will not be able to load its state. Hibernate does not support lazy initialization for detached objects. The fix is to move the code that reads from the collection to just before the transaction is committed. Alternatively, we

could use a non-lazy collection or association, by specifying *lazy="false"* for the association mapping. However, it is intended that lazy initialization be used for almost all collections and associations. If you define too many non-lazy associations in your object model, Hibernate will end up needing to fetch the entire database into memory in every transaction!

Initializing collections and proxies

A *LazyInitializationException* will be thrown by Hibernate if an uninitialized collection or proxy is accessed outside of the scope of the Session, ie. when the entity owning the collection or having the reference to the proxy is in the detached state.

Sometimes we need to ensure that a proxy or collection is initialized before closing the Session. Of course, we can always force initialization by calling *item.getBids()* or *item.getBids().size()*, for example. But that is confusing to readers of the code and is not convenient for generic code.
The static methods *Hibernate.initialize()* and *Hibernate.isInitialized()* provide the application with a convenient way of working with lazily initialized collections or proxies. *Hibernate.initialize(item)* will force the initialization of a proxy, item, as long as its Session is still open.
Hibernate.initialize(item.getBids()) has a similar effect for the collection of bids.

Another option is to keep the Session open until all needed collections and proxies have been loaded. In some application architectures, particularly where the code that accesses data using Hibernate, and the code that uses it are in different application layers or different physical processes, it can be a problem to ensure that the Session is open when a collection is initialized. There are two basic ways to deal with this issue:

- In a web-based application, a servlet filter can be used to close the Session only at the very end of a user request, once the rendering of the view is complete (the Open Session in View pattern). Of course, this places heavy demands on the correctness of the exception handling of your application infrastructure. It is vitally important that the Session is closed and the transaction ended before returning to the user, even when an exception occurs during rendering of the view.
- In an application with a separate business tier, the business logic must "prepare" all collections that will be needed by the web tier before returning. This means that the business tier should load all the data and return all the data already initialized to the presentation/web tier that is required for a particular use case. Usually, the application calls *Hibernate.initialize()* for each collection that will be needed in the web tier (this call must occur before the session is closed) or retrieves the collection eagerly using a Hibernate query with a FETCH clause or a *FetchMode.JOIN* in Criteria. This is usually easier if you adopt the Command pattern instead of a Session Facade.
- You may also attach a previously loaded object to a new Session with *merge()* or *lock()* before accessing uninitialized collections (or other proxies). No, Hibernate does not, and certainly should not do this automatically, since it would introduce ad hoc transaction semantics!

The Second Level Cache

A Hibernate Session is a transaction-level cache of persistent data. It is possible to configure a cluster or JVMlevel (SessionFactory-level) cache on a class-by-class and collection-by-collection basis. You may even plugin a clustered cache. Be careful. Caches are never aware of changes made to the persistent store by another application (though they may be configured to regularly expire cached data).
You have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements *org.hibernate.cache.CacheProvider* using the property *hibernate.cache.provider_class. Hibernate* comes bundled with a number of built-in integrations with open-source cache providers (listed below); additionally, you could implement your own and plug it in as outlined above.

Cache Providers

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	org.hibernate.cache.HashtableCacheProvider	memory		yes
EHCache	org.hibernate.cache.EhCacheProvider	memory, disk		yes
OSCache	org.hibernate.cache.OSCacheProvider	memory, disk		yes
SwarmCache	org.hibernate.cache.SwarmCacheProvider	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	org.hibernate.cache.TreeCacheProvider	clustered (ip multicast), transactional	yes (replication)	yes (clock sync req.)

Cache mappings

The *<cache>* element of a class or collection mapping has the following form:

```
<cache
usage="transactional|read-write|nonstrict-read-write|read-only" (1)
region="RegionName" (2)
include="all|non-lazy" (3)
/>
```

(1) usage (required) specifies the caching strategy: transactional, read-write, nonstrict-read-write or read-only
(2) region (optional, defaults to the class or collection role name) specifies the name of the second level cache region

(3) include (optional, defaults to all) non-lazy specifies that properties of the entity mapped with lazy="true" may not be cached when attribute-level lazy fetching is enabled

Alternatively (preferably?), you may specify <class-cache> and <collection-cache> elements in hibernate.cfg.xml.

Strategy: read only

If your application needs to read but never modify instances of a persistent class, a read-only cache may be used. This is the simplest and best performing strategy. It's even perfectly safe for use in a cluster.

Strategy: read/write

If the application needs to update data, a read-write cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required. If the cache is used in a JTA environment, you must specify the property *hibernate.transaction.manager_lookup_class*, naming a strategy for obtaining the JTA TransactionManager. In other environments, you should ensure that the transaction is completed when *Session.close()* or *Session.disconnect()* is called. If you wish to use this strategy in a cluster, you should ensure that the underlying cache implementation supports locking. The built-in cache providers do not.

Strategy: nonstrict read/write

If the application only occasionally needs to update data (ie. if it is extremely unlikely that two transactions would try to update the same item simultaneously) and strict transaction isolation is not required, a nonstrictread-write cache might be appropriate. If the cache is used in a JTA environment, you must specify *hibernate.transaction.manager_lookup_class*. In other environments, you should ensure that the transaction is completed when *Session.close()* or *Session.disconnect()* is called.

Strategy: transactional

The transactional cache strategy provides support for fully transactional cache providers such as *JBoss TreeCache*. Such a cache may only be used in a JTA environment and you must specify *hibernate.transaction.manager_lookup_class*.

Cache Concurrency Strategy Support

Cache	read-only	read-only nonstrictread-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes			yes

Managing the caches

Whenever you pass an object to *save()*, *update()* or *saveOrUpdate()* and whenever you retrieve an object using *load(), get(), list(), iterate()* or *scroll()*, that object is added to the internal cache of the Session. When *flush()* is subsequently called, the state of that object will be synchronized with the database. If you do not want this synchronization to occur or if you are processing a huge number of objects and need to manage memory efficiently, the *evict()* method may be used to remove the object and its collections from the *firstlevelcache*.

```
ScrollableResult items = sess.createQuery("from Item as item").scroll(); //a huge result set
while ( items.next() ) {
Item item = (Item) items.get(0);
doSomethingWithAItem(item);
sess.evict(item);
}
```

The Session also provides a *contains()* method to determine if an instance belongs to the session cache. To completely evict all objects from the session cache, call *Session.clear()* For the second-level cache, there are methods defined on SessionFactory for evicting the cached state of an instance, entire class, collection instance or entire collection role.

```
sessionFactory.evict(Item.class, itemId); //evict a particular Item
sessionFactory.evict(Item.class); //evict all Items
sessionFactory.evictCollection("Item.bids", itemId); //evict a particular collection of bids
sessionFactory.evictCollection("Item.bids"); //evict all bid collections
```

The CacheMode controls how a particular session interacts with the second-level cache.

- **CacheMode.NORMAL** - read items from and write items to the second-level cache
- **CacheMode.GET** - read items from the second-level cache, but don't write to the second-level cache except when updating data
- **CacheMode.PUT** - write items to the second-level cache, but don't read from the second-level cache
- **CacheMode.REFRESH** - write items to the second-level cache, but don't read from the second-level cache, bypass the effect of *hibernate.cache.use_minimal_puts*, forcing a refresh of the second-level cache for all items read from the database

To browse the contents of a second-level or query cache region, use the Statistics API:

```
Map cacheEntries = sessionFactory.getStatistics()
.getSecondLevelCacheStatistics(regionName)
.getEntries();
```

You'll need to enable statistics, and, optionally, force Hibernate to keep the cache entries in a more human understandable format:

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

The Query Cache

Query result sets may also be cached. This is only useful for queries that are run frequently with the same parameters.

To use the query cache you must first enable it:

```
hibernate.cache.use_query_cache true
```

This setting causes the creation of two new cache regions - one holding cached query result sets (*org.hibernate.cache.StandardQueryCache*), the other holding timestamps of the most recent updates to queryable tables (*org.hibernate.cache.UpdateTimestampsCache*). Note that the query cache does not cache the state of the actual entities in the result set; it caches only identifier values and results of value type. So the query cache should always be used in conjunction with the second-level cache.
Most queries do not benefit from caching, so by default queries are not cached. To enable caching, call *Query.setCacheable(true)*. This call allows the query to look for existing cache results or add its results to the cache when it is executed.

By Nima Goudarzi - July, 2007

References:
- Hibernate3 reference
- Hibernate in Action (Manning 2005)