Hibernate Caching

"Code with Passion!"

Topics

- Types of Hibernate cache
- First-level cache
- Second-level cache
  - Second-Level Caching implementations
  - Second-Level Caching strategies
- Query cache
- Hibernate statistics

Types of Hibernate Cache

What is Caching?

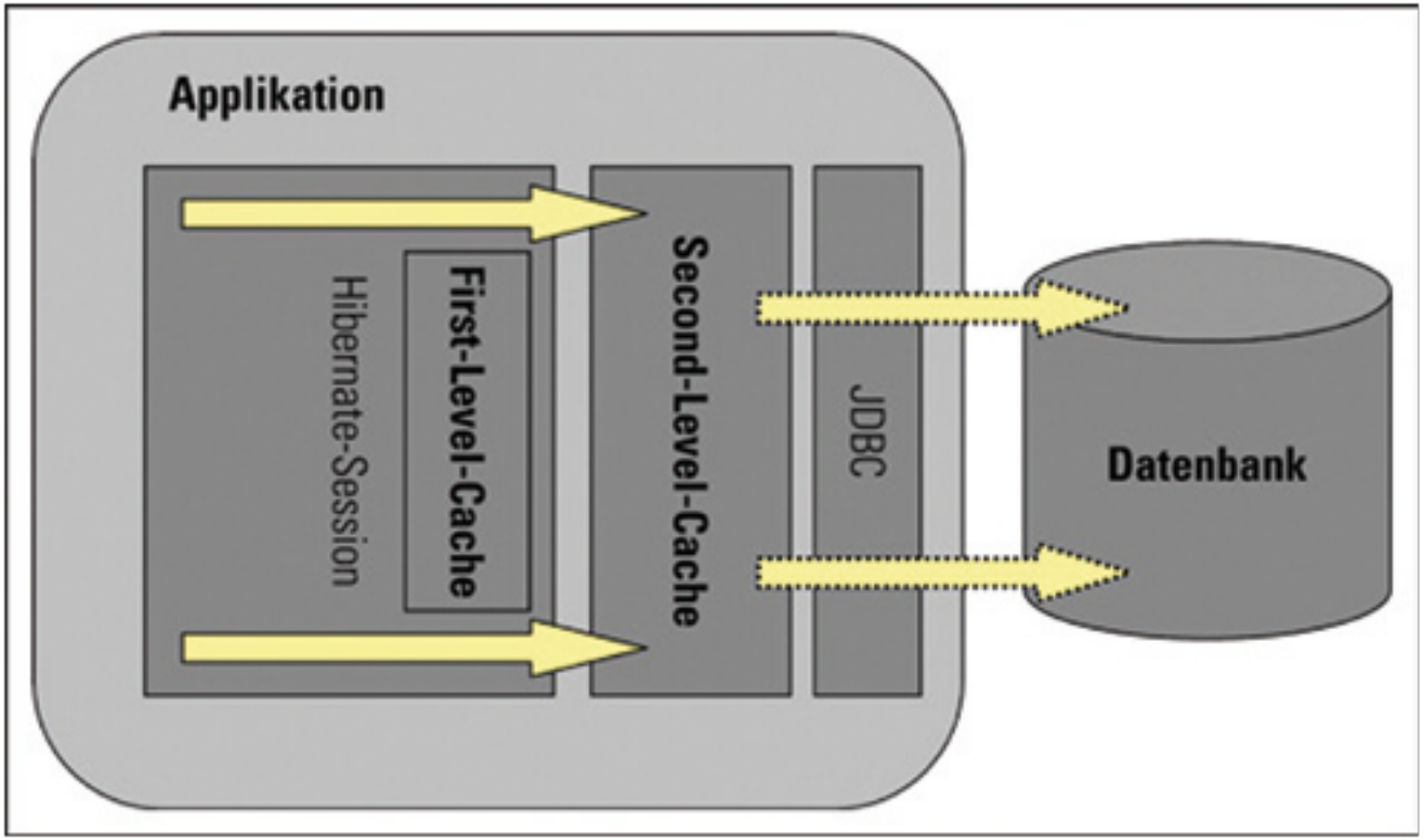
- Used for optimizing database applications.
- A cache is designed to reduce traffic between your application and the database by retaining data already loaded from the database.
  - Database access is necessary only when retrieving data that is not currently available in the cache.
  - The application may need to empty (invalidate) the cache from time to time if the database is updated or modified in some way, because it has no way of knowing whether the cache is up to date

Types of Hibernate Cache

- For caching objects
  - First-level cache - associated with the *Session* object
  - Second-level cache - associated with the *SessionFactory* object
- For caching queries and their results
  - Query cache

First-level cache

First-level and Second-level Cache



First-level Cache

- Maintained per a session
- session.clear() - removes all entries from the first-level cache
- session.evict(..) - remove an entry from the first-level cache
- session.contains(..) - check if an entry is in the first-level cache

How Hibernate Use First-Level Cache (1)

1. First level cache is associated with “session” object and other session objects in application can not see it.
2. The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
3. First level cache is enabled by default and you can not disable it.
4. When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
5. If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.

How Hibernate Use First-Level Cache (1)

6. The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
7. The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

Lab:

Exercise 1: First Level Cache
3518_hibernate_caching.zip

Second-level cache

Second-level Cache

- Second-level cache keeps loaded objects at the Session Factory level across transaction boundaries
- The objects in the second-level cache are available to the whole application, not just to the user running the query
  - This way, each time a query returns an object that is already loaded in the cache, one or more database transactions potentially are avoided.
- Second-level Cache provider can be pluggable
  - 3rd-party implementation can be pluggable

How Hibernate Use Second-Level Cache (1)

1. Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.

How Hibernate Use Second-Level Cache (1)

5. If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
6. Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
7. If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again.

Second-level Cache Implementations

Second-level Cache Implementations

- Hibernate supports these open-source cache implementations out of the box
  - EHCache (org.hibernate.cache.EhCacheProvider)
  - OSCache (org.hibernate.cache.OSCacheProvider)
  - SwarmCache (org.hibernate.cache.SwarmCacheProvider)
  - JBoss TreeCache (org.hibernate.cache.TreeCacheProvider)

EHCache

- Fast, lightweight, and easy-to-use in-process cache
- Supports read-only and read/write caching, and memory- and disk-based caching.
- However, it does not support clustering

OSCache

- Part of a larger package, which also provides caching functionalities for JSP pages or arbitrary objects.
- It is a powerful and flexible package, which, like EHCache, supports read-only and read/write caching, and memory- and disk-based caching.
- It also provides basic support for clustering via either JavaGroups or JMS

SwarmCache

- Is a simple cluster-based caching solution based on JavaGroups.
- Supports read-only or nonstrict read/write caching (the next section explains this term).
- This type of cache is appropriate for applications that typically have many more read operations than write operations.

JBoss TreeCache

- Is a powerful replicated (synchronous or asynchronous) and transactional cache.
- Use this solution if you really need a true transaction-capable caching architecture

Second-level Cache Strategies

Second-level Caching Strategies

- Read-only
- Read/write
- Nonstrict read/write
- Transactional

Caching Strategy: Read-only

- Useful for data that is read frequently but never updated.
- Simplest and best-performing cache strategy.

Caching Strategy: Read-only

```
<hibernate-mapping package="com.wakaleo.articles.caching.businessobjects">
   <class name="Country" table="COUNTRY" dynamic-update="true">
      <meta attribute="implement-equals">true</meta>
      <cache usage="read-only"/>

      <id name="id" type="long" unsaved-value="null" >
         <column name="cn_id" not-null="true"/>
         <generator class="increment"/>
      </id>

      <property column="cn_code" name="code" type="string"/>
      <property column="cn_name" name="name" type="string"/>

      <set name="airports" >
         <cache usage="read-only"/>
         <key column="cn_id"/>
         <one-to-many class="Airport"/>
      </set>
   </class>
</hibernate-mapping>
```

Caching Strategy: Read/Write

- Appropriate if your data needs to be updated.
- They carry more overhead than read-only caches.
- In non-JTA environments, each transaction should be completed when *Session.close()* or *Session.disconnect()* is called.

Caching Strategy: Read/Write

```
<hibernate-mapping package="com.wakaleo.articles.caching.businessobjects">
   <class name="Employee" table="EMPLOYEE" dynamic-update="true">
      <meta attribute="implement-equals">true</meta>
      <cache usage="read-write"/>

      <id name="id" type="long" unsaved-value="null" >
         <column name="emp_id" not-null="true"/>
         <generator class="increment"/>
      </id>
      <property column="emp_surname" name="surname" type="string"/>
      <property column="emp_firstname" name="firstname" type="string"/>
      <many-to-one name="country"
      column="cn_id"
      class="com.wakaleo.articles.caching.businessobjects.Country"
      not-null="true" />
      <!-- Lazy-loading is disactivated to demonstrate caching behavior -->
      <set name="languages" table="EMPLOYEE_SPEAKS_LANGUAGE" lazy="false">
         <cache usage="read-write"/>
         <key column="emp_id"/>
         <many-to-many column="lan_id" class="Language"/>
      </set>
   </class>
</hibernate-mapping>
```

Caching Strategy: Nonstrict Cache

- Does not guarantee that two transactions won't simultaneously modify the same data.
  - Therefore, it may be most appropriate for data that is read often but only occasionally modified.

Caching Strategy: Transactional

- A fully transactional cache that may be used only in a JTA environment

Lab:

Exercise 2: Second Level Caching
3518_hibernate_caching.zip

Query Cache

Query Cache

- Use it when you need to cache actual query results, rather than just persistent objects
- Always used with second-level cache

Query-level Cache Example

- Suppose
  - Query query = session.createQuery("from Person as p where p.parent.id=? and p.firstName=?");
  - query.setInt(0, Integer.valueOf(1));
  - query.setString(1, "Joey");
  - query.setCacheable(true);
  - List l = query.list();
- The query cache then contains
  - The combination of the query and the values provided as parameters to that query is used as a key, and the value is the list of identifiers for that query
  - ["from Person as p where p.parent.id=? and p.firstName=?", [1 , "Joey"]] -> [2]

Lab:

Exercise 3: Query Cache
3518_hibernate_caching.zip

Hibernate Statistics

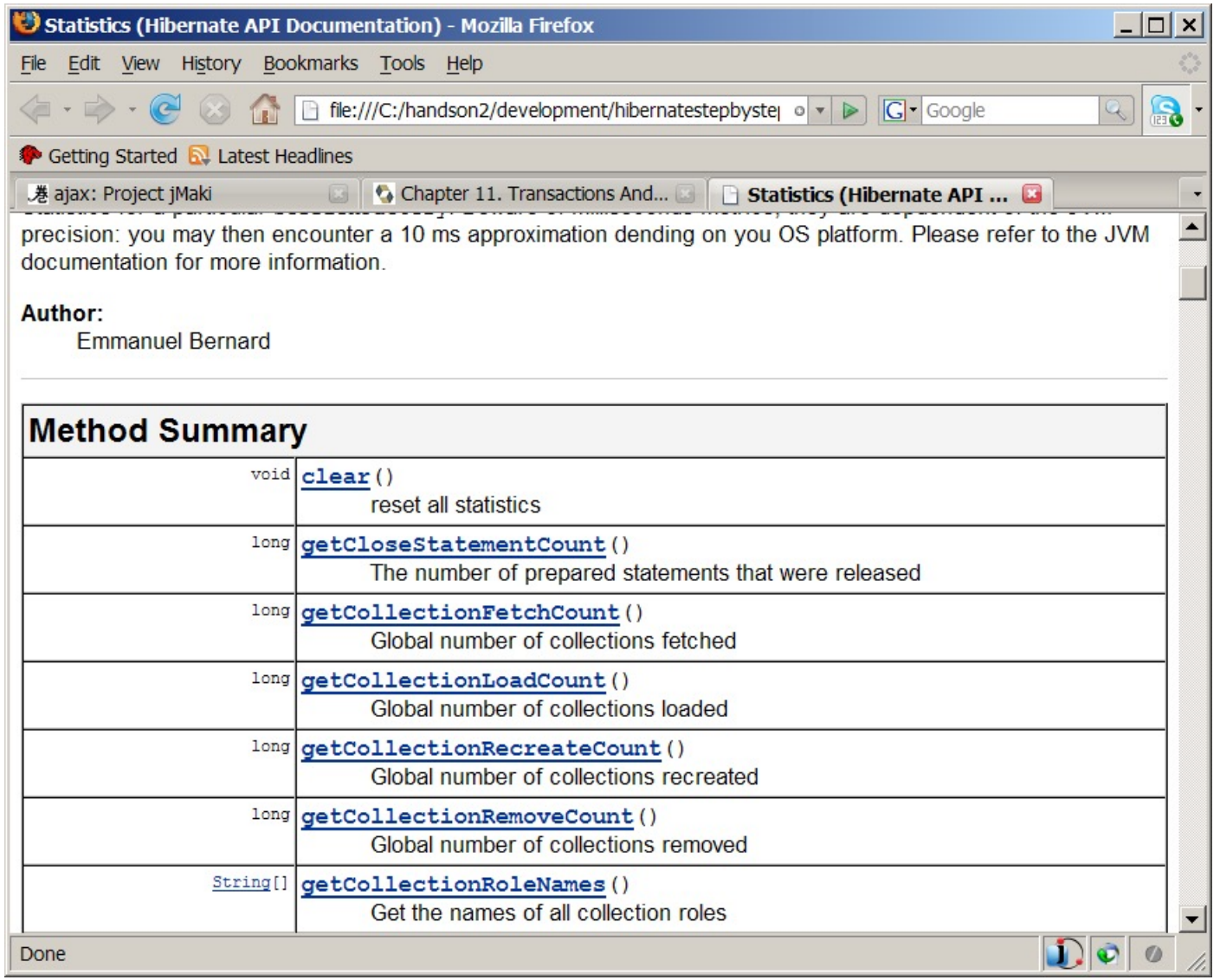
Statistics

- Hibernate provides a full range of figures about its internal operations.
- Statistics in Hibernate are available per *SessionFactory*.
  - Option1: Call *sessionFactory.getStatistics()* and read or display the Statistics yourself.
  - Option2: Through JMX

Statistics Interface

- Hibernate provides a number of metrics, from very basic to the specialized information only relevant in certain scenarios.
- All available counters are described in the Statistics interface API, in three categories:
  - Metrics related to the general Session usage, such as number of open sessions, retrieved JDBC connections, etc.
  - Metrics related to he entities, collections, queries, and caches as a whole (aka global metrics),
  - Detailed metrics related to a particular entity, collection, query or cache region.

Statistics Interface



Example Code

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + chang
```

Lab:

Exercise 4: Statistics
3518_hibernate_caching.zip

Code with Passion!