CHAPTER 10 Performance

When you begin experimenting with Hibernate, one of the first tasks you are likely to perform is the installation of a monitor to see the generated SQL. This is especially important if you want to understand how Hibernate generates SQL for such features as collections and lazy loading of data. This chapter describes how to gather performance metrics for the use of Hibernate in the field.

Finding and Solving Problems

Hibernate affords a basic SQL monitoring capability, but for real development you are best advised to use a tool with a bit more sophistication. By definition, every interaction between your application and the database is translated through a JDBC driver. A pass-through driver is used to analyze the data. The pass-through driver does not change the data, but records all of the interaction for analysis. In this section, we will look at the pass-through JDBC driver p6spy and the use of IronTrack SQL to understand the data it generates.

IronTrack SQL

IronTrack SQL is an open-source Apache-licensed tool that works in conjunction with the p6spy driver monitor. Using p6spy (included with IronTrack SQL), every interaction between the application and the database is logged. IronTrack SQL, in turn, allows you to view these generated logs (either at runtime via TCP/IP or by opening generated log files).

Configuring IronTrack SQL

IronTrack SQL can be downloaded free from http://www.irongrid.com/. You will obtain a file with a name such as `irontracksql-installer-1_0_172.jar`. Once you have saved this file to your system, you can install it

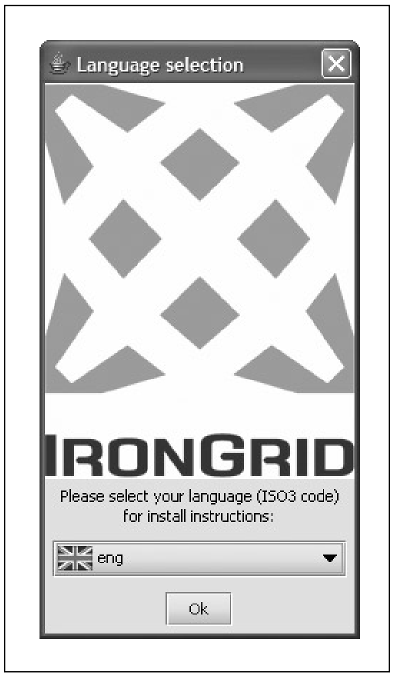

Language selection
IRONGRID
Please select your language (ISO3 code) for install instructions:
eng
Ok

Figure 10.1. Starting IronTrack SQL Installation

with the command `java -jar irontracksql-installer-1_0_172.jar`. The installer will launch, presenting a language screen, as shown in Figure 10.1.

You can accept the defaults throughout the installation, although you may wish to specify a shorter, alternative destination path for the installation, as shown in Figure 10.2, because you will be placing libraries present in the installation in your application path.

If you are using an application server, the precise installation process for IronTrack SQL varies (see http://www.irongrid.com/documentation/). To use IronTrack with a standalone application, you will need to place the following files on your class path:

```
ironeyesql.jar
p6spy.jar
```

Next, you will need to update your Hibernate.properties to point to the p6spy driver (or whatever mechanism you are using to specify JDBC connectivity). You will observe that the line with the default driver has been commented out with a # character, not deleted. The log files generated by p6spy can become quite large

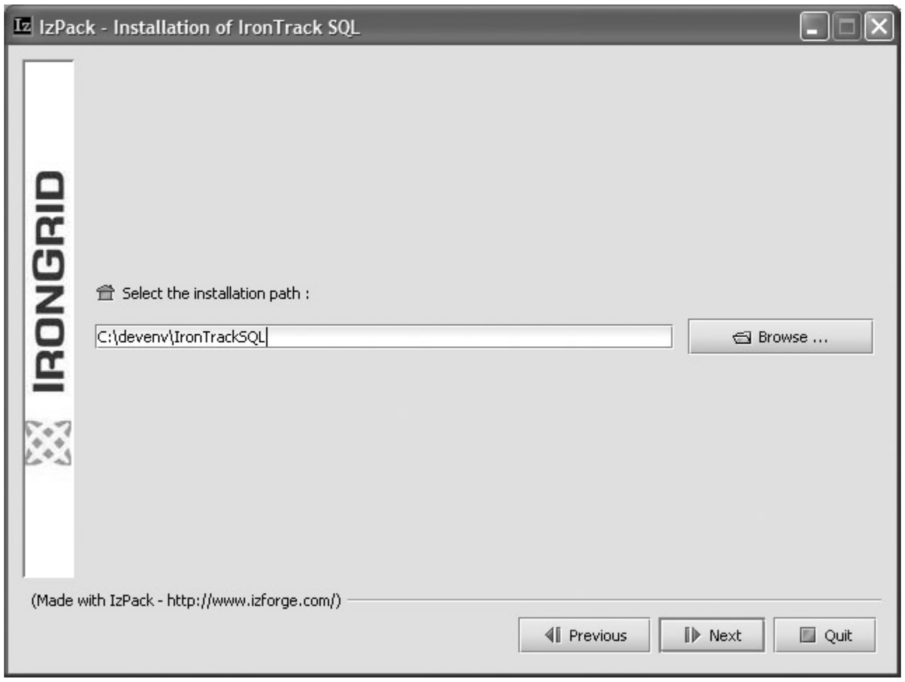

IzPack - Installation of IronTrack SQL
IRONGRID
Select the installation path :
C:\devenv\IronTrackSQL
Browse ...
(Made with IzPack - http://www.izforge.com/)
Previous
Next
Quit

Figure 10.2. Alternative Installation Directory

(especially with full logging and stack trace tracking turned on). Therefore, you'll want to keep your standard driver class close at hand for when you wish to switch to production use. Listing 10.1 shows the Hibernate properties that should be set to make use of p6spy.

Listing 10.1 Configuring p6spy Properties

```
#hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.driver_class=com.p6spy.engine.spy.
P6SpyDriver
hibernate.connection.url=jdbc:mysql://localhost/hibernate
hibernate.connection.username=root
hibernate.connection.password=
hibernate.dialect=net.sf.hibernate.dialect.MySQLDialect
hibernate.show_sql=false
```

Finally, you will need to place a `spy.properties` file in your class path (typically next to your `hibernate.properties`). This file is used to configure the logging produced by p6spy. You should start by copying the `spy.properties` file included with the IronTrack SQL distribution. The most important thing is to set the `spy.properties` to use the correct driver, as in `realdriver=com.mysql.jdbc.Driver`.

After changing these configuration options, simply run your application as you normally would. The default p6spy options will log every SQL statement to a log file (`spy.log`) in the application root directory.

WHERE WAS THAT SQL GENERATED?

p6spy will generate a stack trace pointing to the class that generated a SQL statement if you set `stacktrace=true` in the `spy.properties` file. This will slow your application down, because generating a stack trace is expensive, but it can be very helpful if you are working with a large, unfamiliar application and are having trouble tracking down a particular statement.

Using IronTrack SQL

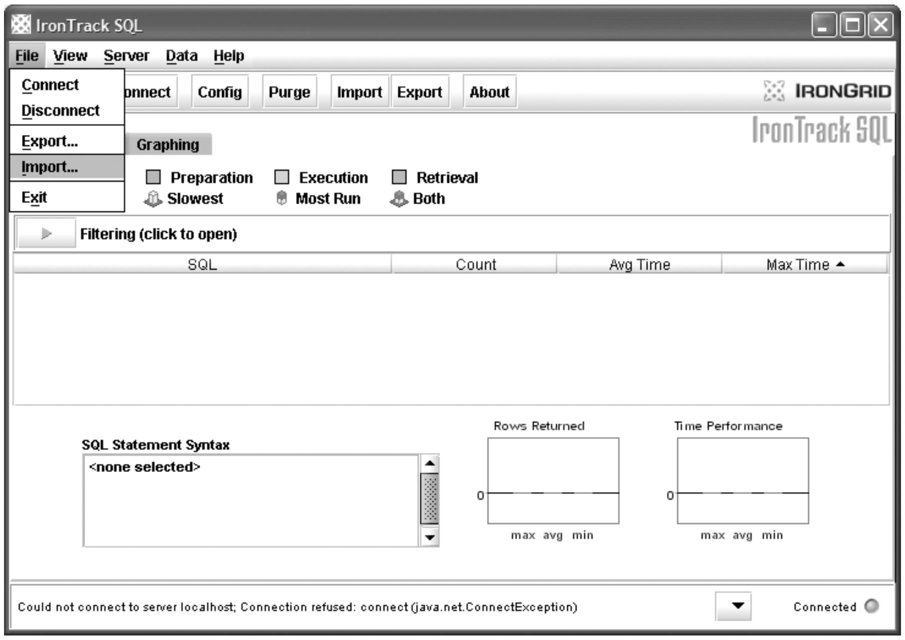
If you are running your application in a long-lived environment (for example, in the context of an application server), you can use the IronTrack SQL graphical user interface to view your data at runtime via TCP/IP. Alternatively, you can simply load the generated `spy.log` file. This would be appropriate if your application runs and then terminates (as do several of the examples in this book) or, to cite another example, if you are unable to connect to the server via TCP/IP (perhaps due to a firewall installed on the server).

You may have a shortcut already created that can launch IronTrack SQL. If not, you can launch IronTrack SQL from the command line with the command `java -jar irontracksql.jar`. Once you've launched the IronTrack SQL interface, you can either connect to a running application via TCP/IP or you can import a generated log file. Figure 10.3 shows IronTrack SQL launched, with the Import… command selected.

To view the generated log files, you'll need to change the Files of Type option to `spy.log` files, as shown in Figure 10.4.

IronTrack allows you to sort and filter the loaded SQL statements. For example, Figure 10.5 shows the results of a run of the sample application shown in Chapter 3. As can be seen, the `ALTER TABLE` statements are relatively expensive, but so are our `INSERT` statements.

Clicking the Graphing tab on the IronTrack SQL main interface allows us to see a graph of the generated SQL statements. As shown in Figure 10.6, the load on the server can be viewed at different points in time (useful for identifying certain operations that may be highly performance intensive).



Could not connect to server localhost; Connection refused: connect (java.net.ConnectException)

Figure 10.3. IronTrack SQL Import

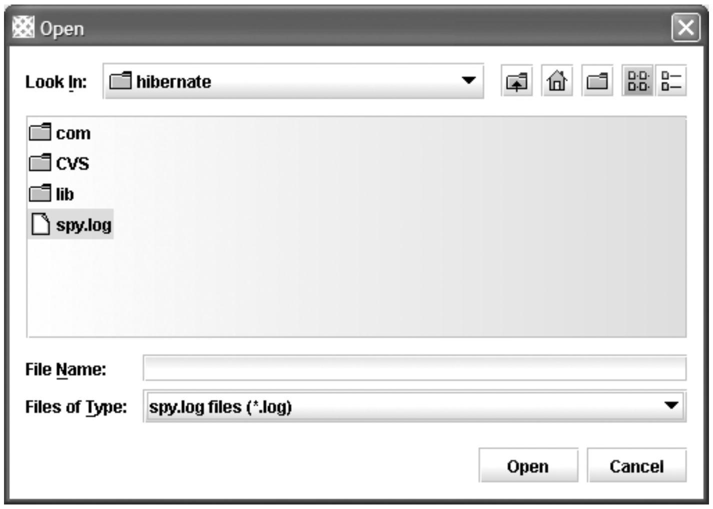
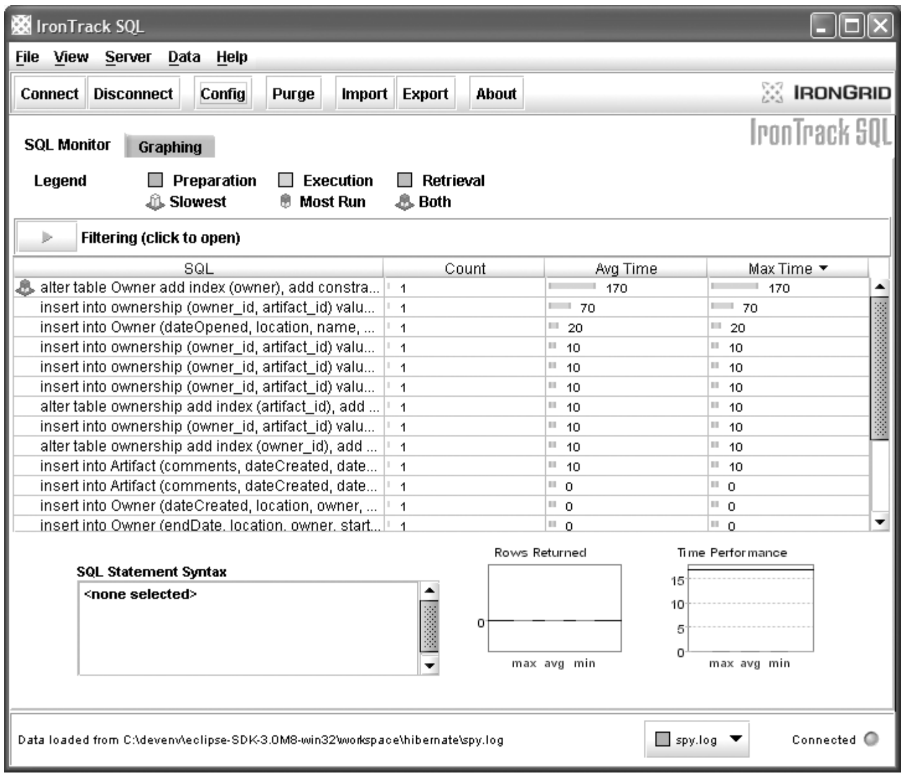


Figure 10.4. Selecting a spy.log File



SQL	Count	Avg Time	Max Time
alter table Owner add index (owner), add constra...	1	170	170
insert into ownership (owner_id, artifact_id) valu...	1	70	70
insert into Owner (dateOpened, location, name, ...	1	20	20
insert into ownership (owner_id, artifact_id) valu...	1	10	10
insert into ownership (owner_id, artifact_id) valu...	1	10	10
insert into ownership (owner_id, artifact_id) valu...	1	10	10
alter table ownership add index (artifact_id), add ...	1	10	10
insert into ownership (owner_id, artifact_id) valu...	1	10	10
alter table ownership add index (owner_id), add ...	1	10	10
insert into Artifact (comments, dateCreated, date...	1	10	10
insert into Artifact (comments, dateCreated, date...	1	0	0
insert into Owner (dateCreated, location, owner, ...	1	0	0
insert into Owner (endDate, location, owner, start...	1	0	0

<none selected>

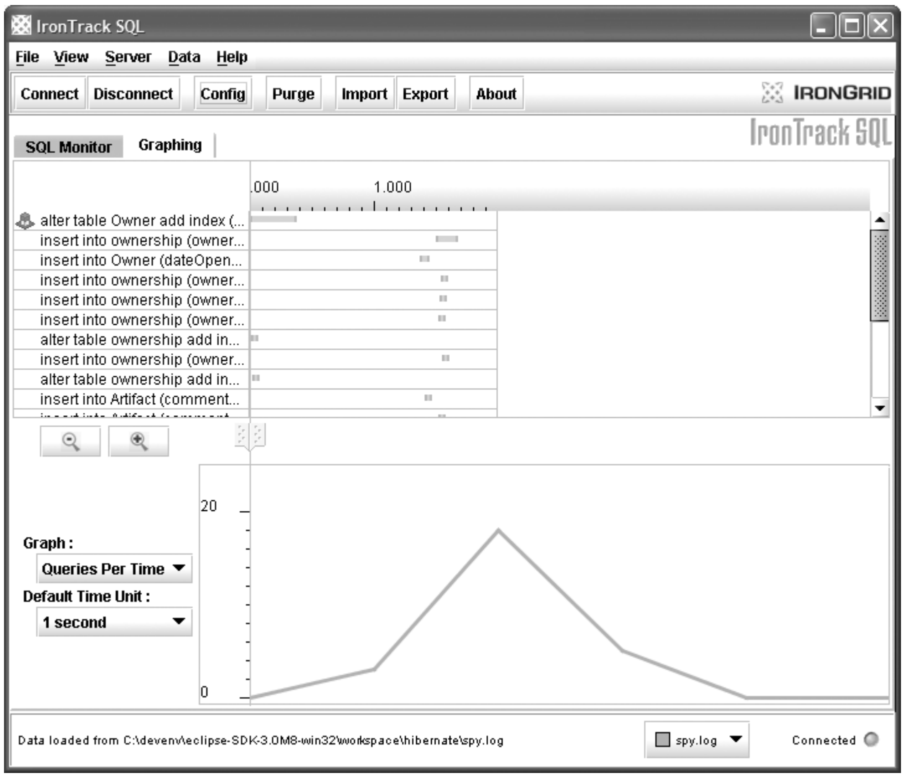
Data loaded from C:\devenv\eclipse-SDK-3.0M8-win32\workspace\hibernate\spy.log

Figure 10.5. Viewing SQL Statements

Queries

You may wish to use Hibern8 IDE and IronTrack SQL in conjunction to test the SQL generated by your HQL queries. Simply launch the Hibern8 IDE as described in Chapter 8, specifying `hibernate.properties` with the p6spy configuration, as shown in Listing 10.1.

After loading the first `*.hbm.xml` file, you can connect to the Hibern8 IDE instance with the IronTrack SQL monitor via TCP/IP. Assuming that you are using the default configuration values and are running your application on your local system, you will then be able to connect to the Hibern8 IDE instance and see real-time results of your HQL—both the generated SQL and the resulting timing information.



```
alter table Owner add index (...
insert into ownership (owner...
insert into Owner (dateOpen...
insert into ownership (owner...
insert into ownership (owner...
insert into ownership (owner...
alter table ownership add in...
insert into ownership (owner...
alter table ownership add in...
insert into Artifact (comment...
```

Data loaded from C:\devenv\eclipse-SDK-3.0M8-win32\workspace\hibernate\spy.log

Figure 10.6. IronTrack SQL Import

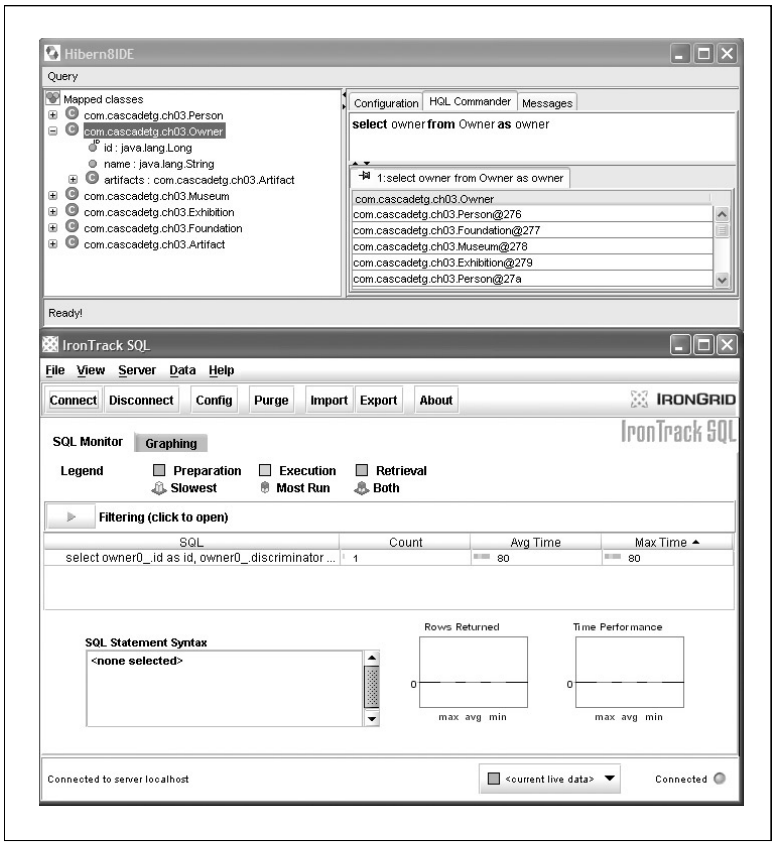
When using Hibern8 IDE and IronTrack SQL in conjunction, you may obtain better results if you disable your cache and connection pool settings.

Two areas are of special interest in regard to query performance—lazy objects and collections.

Lazy Objects

When designing your application, you should generally default to `lazy="true"` whenever possible, and then tune your application to ensure that your queries return the object set as needed (see the `class` tag in Chapter 5 for more information).

As shown in Chapter 8, it's easy to write a query that uses the `fetch` outer join command to have Hibernate automatically load the child objects of a collection that has been marked `lazy="true"`. Thus the rule of thumb should be: only



```
select owner from Owner as owner
```

1:select owner from Owner as owner

com.cascadetg.ch03.Owner
com.cascadetg.ch03.Person@276
com.cascadetg.ch03.Foundation@277
com.cascadetg.ch03.Museum@278
com.cascadetg.ch03.Exhibition@279
com.cascadetg.ch03.Person@27a

SQL	Count	Avg Time	Max Time ▲
select owner0_.id as id, owner0_.discriminator ...	1	80	80

<none selected>

Figure 10.7. Real-Time HQL Testing

use `lazy="false"` if you expect to actually need access to the collection on every possible read.

This is likely to be an area of some confusion when you start working with Hibernate. For example, given a teacher -> student relationship, if `lazy="false"`, loading the teacher will load the entire class. Similarly, if `lazy="true"` and the students aren't pre-fetched by a `fetch` statement (or the `Criteria.setFetchMode()`), iterating through the teacher's student list will generate a new SQL `SELECT` statement for each student.

Collections

Many of the performance issues pertaining to collections derive from the semantic collision between what most developers think of as a collection and the actual contracts of a collection. For example, duplicates are not allowed when you are inserting a value into a set, so when adding a new element, Hibernate needs to at least know the primary-key identifier(s). Similarly, for a map, the keys need to be loaded to ensure the proper ordering. The index values must be known for a list (and other indexed collections). The only collection that doesn't have any of these rules is bag, but it offers poor performance when loading data.

After reading the rules regarding collections, you may find that it would be better (or even required) to model your data with a class declared for the collection table, an example of which is shown in Chapter 4. In this case, instead of letting Hibernate to manage the collection for you behind the scenes using the collections contracts, you are free to implement your own queries and retrieval policies.

An area of special interest when working with collections is the extent to which outer joins (as described in Chapter 8) and lazy fetching are used to optimize performance. You can use the `lazy="true"` attribute (as described above and in Chapter 5) to reduce the amount of collection data returned and various outer joins to control the results more carefully, as described in Chapter 8.

Inserts

Bulk inserts of data are a type of operation best **not** performed by Hibernate. For example, a user may have 100,000 records that have to be imported into a single table. Don't use Hibernate for this sort of operation—use your database's built-in import tools instead. The built-in import will be faster than Hibernate (or, for that matter, handwritten JDBC).

If, for some reason, you do need to do a bulk import via Hibernate, take account of the following tips:

- Make sure the `hibernate.jdbc.batch_size` option (specified in your `hibernate.properties`, as described in Chapter 6) is turned on and set to a reasonably large value.
- Consider `Session.commit()` on to break up the transactional overhead. Presumably you will do this only if you are very confident that it will succeed.
- Make sure that you call `Session.close()` it or `Session.clear()` after each call to `Session.commit()`. Otherwise, Hibernate will attempt to maintain the inserted object in the session-level cache.
- Consider the `seqhilo` or `assigned` generator to optimize key generation.

Connection Pooling

Opening a connection to a database is generally much more expensive than executing an SQL statement. A connection pool is used to minimize the number of connections opened between application and database. It serves as a librarian, checking out connections to application code as needed. Much like a library, your application code needs to be strict about returning connections to the pool when complete, for if it does not do so, your application will run out of available connections.

STARVING A POOL

When using connection pooling, it is important to remember that a chunk of bad code that neglects to return connections can starve the rest of the application, causing it to eventually run out of connections and hang (potentially failing nowhere near the actual problem). To test for this, set the maximum connections in your pool to a small number (as low as 1), and use tools like p6spy and IronTrack SQL (described above) to look for statements that fail to close.

This problem can be avoided by **always** using a `finally` block to close your connection, as shown throughout this book.

Hibernate supports a variety of connection pooling mechanisms. If you are using an application server, you may wish to use the built-in pool (typically a connection is obtaining using JNDI). If you can't or don't wish to use your application server's built-in connection pool, Hibernate supports several other connection pools, as shown in Table 10.1.

STATEMENT CACHE

Certain connection pools, drivers, databases, and other portions of the system may provide an additional cache system, known as a statement cache. This cache stores a partially compiled version of a statement in order to increase performance. By reusing the parsed or precompiled statement, the application is able to trade an increase in memory usage for a boost in performance.

You should consider using a statement cache if one is available, but keep in mind that a statement cache is not the same thing as the other forms of caching described later in this chapter.

Table 10.1. Hibernate-Supported Connection Pools

c3p0	http://sourceforge.net/projects/c3p0	Distributed with Hibernate
Apache DBCP	http://jakarta.apache.org/commons/dbcp/	Apache Pool
Proxool	http://proxool.sourceforge.net/	JDBC Pooling Wrapper

The choice of a connection pool is up to you, but be sure to remember that a connection pool is necessary for every production use.

If you wish to use c3p0, the version distributed with Hibernate 2.1.2 (0.8.3) is out of date (and GPL is a problem if you wish to distribute a non-GPL application). If you wish to distribute an application that makes use of c3p0, make sure to download the latest (LGPL) release, `c3p0-0.8.4-test1` or later.

Because Hibernate ships with c3p0, configuration is a simple matter of adding a few Hibernate configuration properties to your `hibernate.properties` (or `hibernate.cfg.xml`) file. Listing 10.2 shows an example of the configuration of c3p0.

Listing 10.2 Sample Hibernate c3p0 Configuration

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate
hibernate.connection.username=root
hibernate.connection.password=
hibernate.dialect=net.sf.hibernate.dialect.MySQLDialect
hibernate.show_sql=false

hibernate.c3p0.max_size=1
hibernate.c3p0.min_size=0
hibernate.c3p0.timeout=5000
hibernate.c3p0.max_statements=100
hibernate.c3p0.idle_test_period=300
hibernate.c3p0.acquire_increment=2
hibernate.c3p0.validate=false
```

The properties shown in Listing 10.2 are as described in Table 10.2.

If you prefer to use Apache DBCP, make sure that the Apache DBCP library is on your class path, and add the properties to your hibernate.properties file, as shown in Table 10.3.

Finally, if you wish to use Proxool as your connection pool provider, you will need to specify `hibernate.properties` values as shown in Table 10.4.

Table 10.2. c3p0 Configuration Options

Property Meaning	Property	Example
Maximum number of database connections to open	`hibernate.c3p0.max_size`	`15`
Initial number of database connections	`hibernate.c3p0.min_size`	`3`
Maximum idle time for a connection (in seconds)	`hibernate.c3p0.timeout`	`5000`
Maximum size of c3p0 statement cache (0 to turn off)	`hibernate.c3p0.max_ statements`	`0`
Number of connections in a clump acquired when pool is exhausted	`hibernate.c3p0.acquire_ increment`	`3`
Idle time before a c3p0 pooled connection is validated (in seconds)	`hibernate.c3p0.idle_test_ period`	`300`
Validate the connection on checkout. Recommend setting the `hibernate .c3p0.idle_test_period` property instead. Defaults to `false`	`hibernate.c3p0.validate`	`true | false`

Unlike c3p0 and DBCP, you will need to include additional configuration options as described at http://proxool.sourceforge.net/configure.html.

Caching

So you've got a performance problem, and you're pretty sure that it lies in a bottleneck between your database and your application server. You've used IronTrack SQL or some other tool to analyze the SQL sent between your application and the database, and you're pretty sure that there isn't much advantage to be squeezed from refining your queries. Instead, you feel certain that the problems are due to the amount of traffic between your application and the database. The solution in this case may be a cache. By storing the data in a cache instead of relying solely on the database, you may be able to significantly reduce the load on the database, and possibly to increase overall performance as well.

Understanding Caches

Generally speaking, anything you can do to minimize traffic between a database and an application server is probably a good thing. In theory, an application ought to be able to maintain a cache containing data already loaded from the database, and only hit the database when information has to be updated. When the database is hit, the changes may invalidate the cache.

Table 10.3. Apache DBCP Configuration Options

Property Meaning	Property	Example
Maximum number of checked-out database connections	`hibernate.dbcp .maxActive`	`8`
Maximum number of idle database connections for connection pool	`hibernate.dbcp .maxIdle`	`8`
Maximum idle time for connections in connection pool (expressed in ms). Set to -1 to turn off	`hibernate.dbcp.max Wait`	`-1`
Action to take in case of an exhausted DBCP connection pool. Set to `0` to fail, `1` to block until a connection is made available, or `2` to grow)	`hibernate.dbcp .whenExhaustedAction`	`1`
Validate connection when borrowing connection from pool (defaults to `true`)	`hibernate.dbcp.test OnBorrow`	`true | false`
Validate connection when returning connection to pool (optional, true, or false)	`hibernate.dbcp.test OnReturn`	`true | false`
Query to execute for connection validation (optional, requires either `hibernate.dbcp.testOn Borrow` or `hibernate.dbcp .testOnReturn`)	`hibernate.dbcp .validationQuery`	Valid SQL SELECT statement (e.g., `SELECT 1+1`)
Maximum number of checked-out statements	`hibernate.dbcp.ps .maxActive`	`8`
Maximum number of idle statements	`hibernate.dbcp.ps .maxIdle`	`8`
Maximum idle time for statements (in ms)	`hibernate.dbcp.ps .maxWait`	`1000 * 60 * 30`
Action to take in case of an exhausted statement pool. Set to `0` to fail, `1` to block until a statement is made available, or `2` to grow)	`hibernate.dbcp.ps .whenExhaustedAction`	`1`

FIRST-LEVEL AND SECOND-LEVEL CACHES

Hibernate actually implements a simple session-level cache, useful on a per-transaction basis. This cache is primarily used to optimize the SQL generated by Hibernate. It is sometimes referred to as a first-level Hibernate cache. For more information on the relationship between a session and the underlying SQL, see Chapter 9.

(*continues*)

Table 10.4. Proxool Configuration Options

Property Meaning	Property	Example
Configure Proxool provider using an XML file	`hibernate.proxool.xml`	`/path/to/file.xml`
Configure the Proxool provider using a properties file `.properties`	`hibernate.proxool.properties`	`/path/to/proxool`
Configure the Proxool provider from an existing pool	`hibernate.proxool.existing_pool`	`true | false`
Proxool pool alias to use (required for `hibernate.proxool.existing_pool`, `hibernate.proxool.properties`, `hibernate.proxool.xml`)	`hibernate.proxool.pool_alias`	As set by Proxool `configuration`

The JVM and distributed cache discussed in this section is referred to as a second-level cache in other sources. Since you will never need to configure the first-level cache, the discussion in the rest of this chapter will refer to the second-level cache simply as “the cache.”

Let’s start by looking at Hibernate without a cache, as shown in Figure 10.8. Data is transferred between Hibernate and the database, and transactions are managed by the database. Hibernate assumes that the data in memory should be refreshed on every access (a reasonable assumption, especially if Hibernate does not have exclusive access to the database).

Figure 10.9 shows Hibernate operating with a single JVM cache used to minimize traffic between Hibernate and the database. This will increase the perfor-

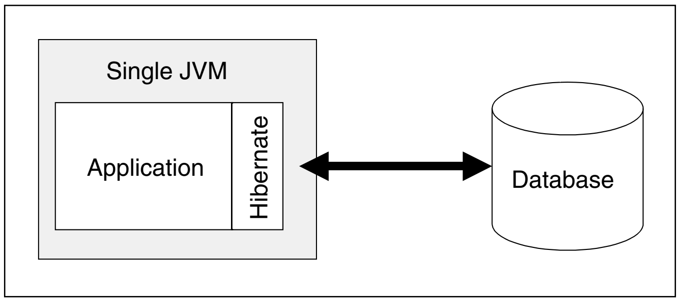


Figure 10.8. Hibernate without a Cache

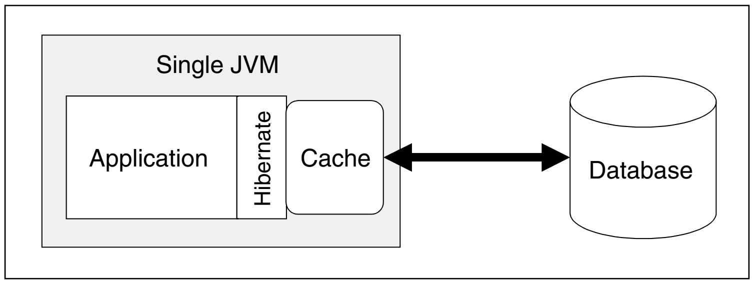


Figure 10.9. Hibernate with a Cache

mance of the application and minimize the load on the database, but at the cost of a bit more configuration complexity (described later in this chapter) and memory usage.

You may wonder how to use Hibernate to perform multithreaded object access and begin pondering strategies for sharing persistent objects across threads. The short answer is: don't! Instead, if you are interested in sharing object data across threads, simply use a cache, as shown in Figure 10.9. If you try to implement your own, the odds are good that you'll have to implement a complex, difficult-to-manage set of thread management, only to end up with cache and concurrency problems.

Figure 10.10 illustrates a problem that may arise when you use a cache. If your application does not have exclusive access to the database (a common situation in

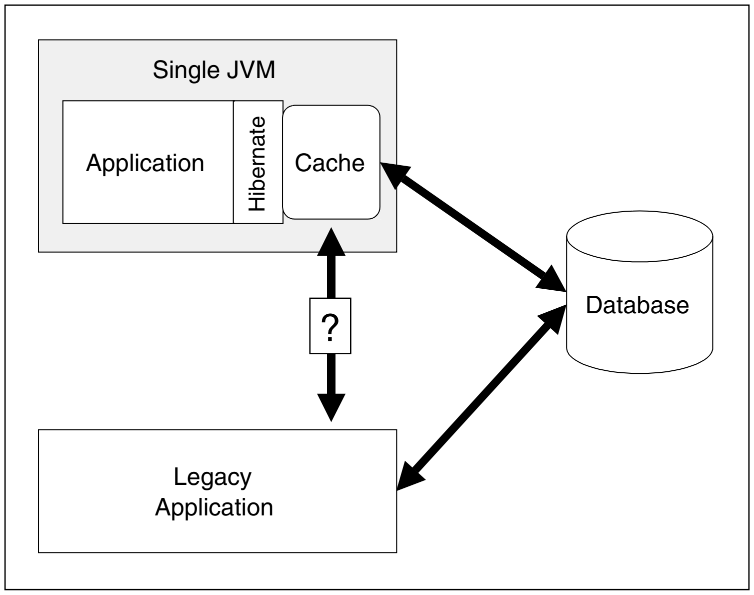


Figure 10.10. Hibernate and a Legacy System

an enterprise environment), your cache can easily become out of sync with the database. If a legacy application updates a record stored in the cache, there is no notification that the data is stale, and therefore the data in the cache will be incorrect.

MULTIPLE `SESSIONFACTORY` OBJECTS

A JVM cache, as described here, is actually a `SessionFactory`-level cache (see Chapter 9 for more information on the scope of a `Session Factory`). There is normally no reason not to share a `SessionFactory` instance throughout your JVM instance, but if for some reason your application uses more than one `SessionFactory`, you're effectively building a multiple JVM system, and therefore will need to use a distributed cache.

Similarly, if you have multiple JVMs running on a single physical system, that still counts as a distributed system.

Unfortunately, there is no ideal solution to the problem of distributed object cache in conjunction with a legacy system. If your Hibernate application has a read-only view of the database, you may be able to configure some cache system to periodically expire data.

If you are able to control all the access to a particular database instance, you may be able to use a distributed cache to ensure that the data traffic is properly synchronized. An example of this is shown in Figure 10.11. Take care when choosing a distributed cache to ensure that the overhead of the cache traffic does not overwhelm the advantages of the cached data.

As a final note, keep in mind that a distributed cache is only one of several possible solutions to a performance problem. Some databases, for example, support an internal distribution mechanism, allowing for the distribution complexity to be entirely subsumed by the database infrastructure (thereby letting the application continue to treat a multisystem database as a single data source).

Configuring a Cache

Applications that perform a large number of read operations in relation to the number of write operations generally benefit the most from the addition of a cache.

The type of cache that would be best depends on such factors as the use of JTA, transaction isolation-level requirements, and the use of clusters. Because of their broad possible needs and uses, Hibernate does not implement caches, but instead relies on a configurable third-party library.

Table 10.5. Supported Cache Environments

Cache	Type	URL
EHCache (Easy Hibernate Cache)	In Process	http://ehcache.sourceforge.net/
OSCache (Open Symphony)	In Process OR Cluster	http://www.opensymphony.com/oscache/
SwarmCache	Cluster	http://swarmcache.sourceforge.net/
JBoss TreeCache	Cluster	http://jboss.org/wiki/Wiki.jsp?page=JBossCache

Standard Caches

In addition to the open-source caches described above, you may wish to investigate Tangosol Coherence, a commercial cache. For more information, see http://hibernate.org/132.html and http://tangosol.com/.

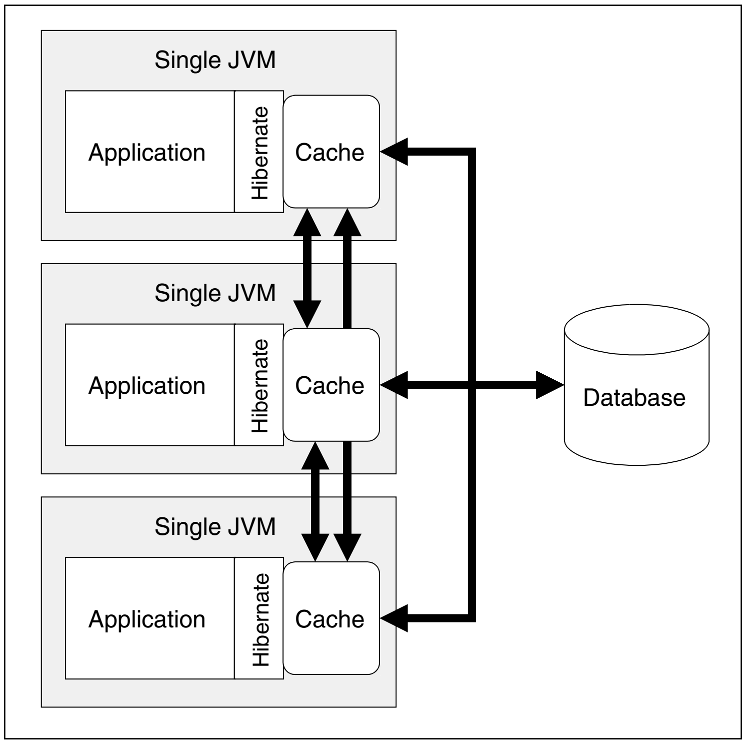


Figure 10.11. Hibernate and a Distributed Cache

Table 10.6 shows the proper setting for the `hibernate.cache.provider_class` property to be passed via the `hibernate.properties` file to enable the use of a cache.

Each cache offers different capabilities in terms of memory and disk-based cache storage and a wide variety of possible configuration options.

Regardless of which cache you choose, you will need to tell Hibernate what sort of cache rules should be applied to your data. This is defined using the `cache` tag (as described in Chapter 5). You can place the `cache` tag in your `*.hbm.xml` files or in the `hibernate.cfg.xml` file. Alternatively, you can configure cache settings programmatically using the `Configuration` object. Table 10.7 shows the values allowed for the `usage` attribute of the `cache` tag.

Conceptually, you are using the options in Table 10.7 to set the per-table read-write options for your data.

Some providers do not support every cache option. Table 10.8 shows which options the various providers support.

JAVA TRANSACTION API (JTA)

According to Sun's documentation, JTA "specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications." In other words, JTA provides for transactions that span multiple application servers—a powerful capability for scaling. Covering JTA is beyond the scope of this text (see http://java.sun.com/products/jta/), but you may wish to consult Chapter 9 for more information on transactions.

Table 10.6. Specifying a Cache

Cache	Property Value
EHCache (Easy Hibernate Cache)	`net.sf.ehcache.hibernate.Provider` (default)
OSCache (Open Symphony)	`net.sf.hibernate.cache.OSCacheProvider`
SwarmCache	`net.sf.hibernate.cache.SwarmCacheProvider`
JBoss TreeCache	`net.sf.hibernate.cache.TreeCacheProvider`
Custom (User-Defined)	Fully qualified class name pointing to a `net.sf.hibernate.cache.CacheProvider` implementation

Table 10.7. Cache Options

Option	Comment
`read-only`	Only useful if your application reads (but does not update) data in the database. Especially useful if your cache provider supports automatic, regular cache expiration. You should also set `mutable=false` for the parent class/collection tag (see Chapter 5).
`read-write`	If JTA is not used, ensure that `Session.close()` or `Session.disconnect()` is used to complete all transactions.
`nonstrict-read-write`	Does not verify that two transactions will not affect the same data; this is left to the application. If JTA is not used, ensure that `Session.close()` or `Session.disconnect()` is used to complete all transactions.
`transactional`	Distributed transaction cache.

Using a Custom Cache

Understanding the interaction between a cache and your application can be very difficult. To help make it clearer, we have included below an example cache implementation that generates logging and statistics about your application's use of the cache (as generated by Hibernate).

Needless to say, don't use this custom cache in a production system.

Configuring the Custom Cache

For this test application, set the property `hibernate.cache.provider_class=com.cascadetg.ch10.DebugHashtableCacheProvider` in your `hibernate.properties` file.

Table 10.8. Cache Options Supported by Provider

Cache	read-only	nonstrict-read-write	read-write	transactional
EHCache	Yes	Yes	Yes	
OSCache	Yes	Yes	Yes	
SwarmCache	Yes	Yes		
JBoss TreeCache	Yes			Yes

Custom Cache Provider

Listing 10.3 shows the options for our simple cache provider. Note that the statistical details are tracked for the allocated caches.

Listing 10.3 Custom Cache Provider

```
package com.cascadetg.ch10;

import java.util.Hashtable;

public class DebugHashtableCacheProvider implements
        net.sf.hibernate.cache.CacheProvider
{

    private static Hashtable caches = new Hashtable();

    public static Hashtable getCaches()
    {
        return caches;
    }

    public static String getCacheDetails()
    {
        StringBuffer newResult = new StringBuffer();
        java.util.Enumeration myCaches = caches.keys();
        while (myCaches.hasMoreElements())
        {
            String myCacheName = myCaches.nextElement()
                    .toString();
            newResult.append(myCacheName);
            newResult.append("\n");

            DebugHashtableCache myCache = (DebugHashtableCache)
                 caches.get(myCacheName);

            newResult.append(myCache.getStats());
            newResult.append("\n\n");
        }

        return newResult.toString();
    }

    /** Creates a new instance of DebugHashtable */
    public DebugHashtableCacheProvider()
    {
    }
```

Listing 10.3 Custom cache provider (*continued*)

```
    public net.sf.hibernate.cache.Cache buildCache(String str,
            java.util.Properties properties)
    {
        System.out.println("New Cache Created");
        DebugHashtableCache newCache = new
             DebugHashtableCache();
        caches.put(str, newCache);
        return newCache;
    }

    public long nextTimestamp()
    {
        return net.sf.hibernate.cache.Timestamper.next();
    }

}
```

Custom Cache Implementation

Listing 10.4 shows the implementation of our simple cache. It's a pretty dumb cache—it just uses a `java.util.Hashtable` as the backing store. Of more interest is the use of `long` values to keep track of the number of accesses to the various cache methods. This can be useful for understanding the kind of access a section of code is generating. For example, you may wish to consider a different approach if your code generates a tremendous number of reads relative to writes.

Listing 10.4 Custom Cache Implementation

```
package com.cascadetg.ch10;

import net.sf.hibernate.cache.CacheException;
import net.sf.hibernate.cache.Timestamper;
import java.util.Hashtable;
import java.util.Map;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class DebugHashtableCache implements
        net.sf.hibernate.cache.Cache
{

    private static Log log = LogFactory
            .getLog(DebugHashtableCache.class);

```

(*continues*)

ch10.qxd 11/3/04 8:20 AM Page 300

Listing 10.4 Custom Cache Implementation (*continued*)

```
    private Map hashtable = new Hashtable(5000);

    public void addStat(StringBuffer in, String label, long
    value)
    {
        in.append("\t");
        in.append(label);
        in.append(" : ");
        in.append(value);
        in.append("\n");
    }

    public String getStats()
    {
        StringBuffer result = new StringBuffer();

        addStat(result, "get hits", get_hits);
        addStat(result, "get misses", get_misses);
        addStat(result, "put replacements", put_hits);
        addStat(result, "put new objects", put_misses);
        addStat(result, "locks", locks);
        addStat(result, "unlocks", unlocks);
        addStat(result, "remove existing", remove_hits);
        addStat(result, "remove unknown", remove_misses);
        addStat(result, "clears", clears);
        addStat(result, "destroys", destroys);

        return result.toString();
    }

    long get_hits = 0;

    long get_misses = 0;

    long put_hits = 0;

    long put_misses = 0;

    long locks = 0;

    long unlocks = 0;

    long remove_hits = 0;

    long remove_misses = 0;
```

Listing 10.4 Custom Cache Implementation (*continued*)

```
    long clears = 0;

    long destroys = 0;

    public Object get(Object key) throws CacheException
    {
        if (hashtable.get(key) == null)
        {
            log.info("get " + key.toString() + " missed");
            get_misses++;
        } else
        {
            log.info("get " + key.toString() + " hit");
            get_hits++;
        }

        return hashtable.get(key);
    }

    public void put(Object key, Object value)
            throws CacheException
    {
        log.info("put " + key.toString());
        if (hashtable.containsKey(key))
        {
            put_hits++;
        } else
        {
            put_misses++;
        }
        hashtable.put(key, value);
    }

    public void remove(Object key) throws CacheException
    {
        log.info("remove " + key.toString());
        if (hashtable.containsKey(key))
        {
            remove_hits++;
        } else
        {
            remove_misses++;
        }
        hashtable.remove(key);
    }
```

(*continues*)

Listing 10.4 Custom Cache Implementation (*continued*)

```
    public void clear() throws CacheException
    {
        log.info("clear ");
        clears++;
        hashtable.clear();
    }

    public void destroy() throws CacheException
    {
        log.info("destroy ");
        destroys++;
    }

    public void lock(Object key) throws CacheException
    {
        log.info("lock " + key.toString());
        locks++;
    }

    public void unlock(Object key) throws CacheException
    {
        log.info("unlock " + key.toString());
        unlocks++;
    }

    public long nextTimestamp()
    {
        return Timestamper.next();
    }

    public int getTimeout()
    {
        return Timestamper.ONE_MS * 60000; //ie. 60 seconds
    }

}
```

Cache Test Object

Listing 10.5 shows a simple mapping file used to test our object. In particular, note the use of the `cache` tag to indicate the type of cache management that should be performed.

Listing 10.5 Simple Performance Test Object Mapping File

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-
2.0.dtd">

<hibernate-mapping>
    <class name="com.cascadetg.ch10.PerfObject"
        dynamic-update="false" dynamic-insert="false">
        <cache usage="read-write" />

        <id name="id" column="id" type="long" >
            <generator class="native" />
        </id>

        <property name="value" type="java.lang.String"
            update="true" insert="true" column="comments" />
    </class>
</hibernate-mapping>
```

Listing 10.6 shows the source generated from the mapping file shown in Listing 10.5.

Listing 10.6 Simple Performance Test Object Java Source

```
package com.cascadetg.ch10;

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/** @author Hibernate CodeGenerator */
public class PerfObject implements Serializable {

    /** identifier field */
    private Long id;

    /** nullable persistent field */
    private String value;

    /** full constructor */
```

(*continues*)

Listing 10.6 Simple Performance Test Object Java Source (*continued*)

```
    public PerfObject(String value) {
        this.value = value;
    }

    /** default constructor */
    public PerfObject() {
    }

    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getValue() {
        return this.value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public String toString() {
        return new ToStringBuilder(this)
            .append("id", getId())
            .toString();
    }

    public boolean equals(Object other) {
        if ( !(other instanceof PerfObject) ) return false;
        PerfObject castOther = (PerfObject) other;
        return new EqualsBuilder()
            .append(this.getId(), castOther.getId())
            .isEquals();
    }

    public int hashCode() {
        return new HashCodeBuilder()
            .append(getId())
            .toHashCode();
    }

}
```

Testing the Cache

Listing 10.7 shows a simple program that tests the cache. If you wish to test this using a larger number of objects, simply change `objects = 5` to a higher value.

Listing 10.7 Testing Cache Hits

```
package com.cascadetg.ch10;

/** Various Hibernate-related imports */
import java.io.FileInputStream;
import java.util.logging.LogManager;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;
import net.sf.hibernate.tool.hbm2ddl.SchemaUpdate;
import net.sf.hibernate.tool.hbm2ddl.SchemaExport;

public class CacheTest
{

    static long objects = 5;

    /** We use this session factory to create our sessions */
    public static SessionFactory sessionFactory;

    /**
     * Loads the Hibernate configuration information, sets up
     * the database and the Hibernate session factory.
     */
    public static void initialization()
    {
        System.out.println("initialization");
        try
        {
            Configuration myConfiguration = new
                 Configuration();

            myConfiguration.addClass(PerfObject.class);

            new SchemaExport(myConfiguration).drop(true, true);

            // This is the code that updates the database to
            // the current schema.
```

(*continues*)

Listing 10.7 Testing Cache Hits (*continued*)

```
            new SchemaUpdate(myConfiguration)
                    .execute(true, true);
            // Sets up the session factory (used in the rest
            // of the application).
            sessionFactory = myConfiguration
                    .buildSessionFactory();

        } catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    public static void createObjects()
    {
        System.out.println();
        System.out.println("createObjects");

        Session hibernateSession = null;
        Transaction myTransaction = null;
        try
        {
            hibernateSession = sessionFactory.openSession();

            for (int i = 0; i < objects; i++)
            {
                myTransaction = hibernateSession
                        .beginTransaction();

                PerfObject myPerfObject = new PerfObject();
                myPerfObject.setValue("");

                hibernateSession.save(myPerfObject);
                hibernateSession.flush();

                myTransaction.commit();
            }
        } catch (Exception e)
        {
            e.printStackTrace();
            try
            {
                myTransaction.rollback();
            } catch (Exception e2)
            {
                // Silent failure of transaction rollback
```

Listing 10.7 Testing Cache Hits (*continued*)

```
            }
        } finally
        {
            try
            {
                hibernateSession.close();
            } catch (Exception e2)
            {
                // Silent failure of session close
            }
        }

        // Explicitly evict the local session cache
        hibernateSession.clear();
    }

    public static void loadAllObjects()
    {
        System.out.println();
        System.out.println("loadAllObjects");

        Session hibernateSession = null;
        Transaction myTransaction = null;

        try
        {
            hibernateSession = sessionFactory.openSession();
            myTransaction =
                 hibernateSession.beginTransaction();

            // In this example, we use the Criteria API. We
            // could also have used the HQL, but the
            // Criteria API allows us to express this
            // query more easily.

            // First indicate that we want to grab all of
            // the artifacts.
            Criteria query = hibernateSession
                    .createCriteria(PerfObject.class);

            // This actually performs the database request,
            // based on the query we've built.
```

(*continues*)

Listing 10.7 Testing Cache Hits (*continued*)

```
            java.util.Iterator results = query.list().iterator();

            PerfObject myPerfObject;

            // Because we are grabbing all of the artifacts and
            // artifact owners, we need to store the returned
            // artifacts.

            java.util.LinkedList retrievedArtifacts = new
                 java.util.LinkedList();
            while (results.hasNext())
            {
                // Note that the result set is cast to the
                // Animal object directly - no manual
                // binding required.
                myPerfObject = (PerfObject) results.next();
                if (!retrievedArtifacts.contains(myPerfObject))
                        retrievedArtifacts.add(myPerfObject);

            }

            myTransaction.commit();
            hibernateSession.clear();
        } catch (Exception e)
        {
            e.printStackTrace();
            try
            {
                myTransaction.rollback();
            } catch (Exception e2)
            {
                // Silent failure of transaction rollback
            }
        } finally
        {
            try
            {
                if (hibernateSession != null)
                        hibernateSession.close();
            } catch (Exception e)
            {
                // Silent failure of session close
            }
        }
```

Listing 10.7 Testing Cache Hits (*continued*)

```
    }
    public static void main(String[] args)
    {
        initialization();
        createObjects();

        long timing = System.currentTimeMillis();
        loadAllObjects();
        System.out.println("Timing #1 : "
                + (System.currentTimeMillis() - timing));

        timing = System.currentTimeMillis();
        loadAllObjects();
        System.out.println("Timing #2 : "
                + (System.currentTimeMillis() - timing));

        timing = System.currentTimeMillis();
        loadAllObjects();
        System.out.println("Timing #3 : "
                + (System.currentTimeMillis() - timing));

        timing = System.currentTimeMillis();
        loadAllObjects();
        System.out.println("Timing #4 : "
                + (System.currentTimeMillis() - timing));

        timing = System.currentTimeMillis();
        loadAllObjects();
        System.out.println("Timing #5 : "
                + (System.currentTimeMillis() - timing));

        System.out.println(DebugHashtableCacheProvider
                .getCacheDetails());

    }
}
```

As can be seen from the output of the program shown in Listing 10.7, our simple application was able to cache the results from the first `loadAllObjects()` method, leading to lower timing values for the remaining access. This is reflected in the statistics for the cache, shown in terms of gets, puts, and so on.

ch10.qxd 11/3/04 8:20 AM Page 310

Listing 10.8 Testing Cache Hits

```
initialization
New Cache Created

createObjects

loadAllObjects
Timing #1 : 40

loadAllObjects
Timing #2 : 10

loadAllObjects
Timing #3 : 0

loadAllObjects
Timing #4 : 10

loadAllObjects
Timing #5 : 0
com.cascadetg.ch10.PerfObject
      get hits : 20
      get misses : 5
      put replacements : 0
      put new objects : 5
      locks : 25
      unlocks : 25
      remove existing : 0
      remove unknown : 0
      clears : 0
      destroys : 0
```