OJB Performance

by Armin Waibel, Thomas Mahler

Table of contents

1. Introduction

" There is no such thing as a free lunch."
(North American proverb)

Object/relational mapping tools hide the details of relational databases from the application developer. The developer can concentrate on implementing business logic and is liberated from caring about RDBMS related coding with JDBC and SQL.

O/R mapping tools allow to separate business logic from RDBMS access by forming an additional software layer between business logic and RDBMS. Introducing new software layers always eats up additional computing resources.
In short: the price for using O/R tools is performance.

Software architects have to take in account this tradeoff between programming comfort and performance to decide if it is appropiate to use an O/R tool for a specific software system.

This document describes the *OJB Performance Test Suite* which was created to lighten the decision between native JDBC, OJB (the different OJB API's) and other O/R mapper.

2. The Performance Test Suite

The *OJB Performance Test Suite* allows to compare OJB against native JDBC programming against your RDBMS of choice and run OJB in a virtual multithreaded environment. Further on it is possible to compare OJB against any O/R mapping tool using a simple framework.

All tests are integrated in the OJB build script, you only need to perform the according ant target:

```
ant target
```

The following 'targets' exist:

- `perf-test` multithreaded performance/stress test of PB/OTM/ODMG api against native JDBC
- `performance` older single threaded test, OJB API implementations (PB, ODMG) against native JDBC
- [`performance3` multithreaded test against two different databases - developers test]

By changing the JdbcConnectionDescriptor in the configuration files you can point to your specific RDBMS. Please refer to this document for details (../../docu/guides/platforms.html) .

3. Interpreting test results

Interpreting the result of these benchmarks carefully will help to decide whether using OJB is viable for specific application scenarios or if native JDBC programming should be used for performance reasons.

Take care of compareable configuration properties when run performance tests with different O/R tools.

If the decision made to use an O/R mapping tool the comparison with other tools helps to find the best one for the thought scenario. But performance shouldn't be the only reason to take a specific O/R tool. There are many other points to consider:

- Usability of the supported API's
- Flexibility of the framework
- Scalability of the framework
- Community support
- The different licences of Open Source projects
- etcetera ...

4. How OJB compares to native JDBC programming?

OJB is shipped with tests compares native JDBC with ODMG and PB-API implementation. This part of the test suite is integrated into the OJB build mechanism.
A single client test you can invoke it by typing `ant performance` or `ant performance`.

If running OJB out of the box the tests will be performed against the Hypersonic SQL shipped with OJB. A typical output looks like follows:

```
performance:
      [ojb] .[performance] INFO: Test for PB-api
      [ojb] [performance] INFO:
      [ojb] [performance] INFO: inserting 2500 Objects: 3257 msec
      [ojb] [performance] INFO: updating 2500 Objects: 1396 msec
      [ojb] [performance] INFO: querying 2500 Objects: 1322 msec
      [ojb] [performance] INFO: querying 2500 Objects: 26 msec
      [ojb] [performance] INFO: fetching 2500 Objects: 495 msec
      [ojb] [performance] INFO: deleting 2500 Objects: 592 msec
      [ojb] [performance] INFO:
      [ojb] [performance] INFO: inserting 2500 Objects: 869 msec
      [ojb] [performance] INFO: updating 2500 Objects: 1567 msec
      [ojb] [performance] INFO: querying 2500 Objects: 734 msec
      [ojb] [performance] INFO: querying 2500 Objects: 20 msec
      [ojb] [performance] INFO: fetching 2500 Objects: 288 msec
      [ojb] [performance] INFO: deleting 2500 Objects: 447 msec
      [ojb] [performance] INFO:
      [ojb] [performance] INFO: inserting 2500 Objects: 979 msec
```

```
     [ojb] [performance] INFO: updating 2500 Objects: 1240 msec
     [ojb] [performance] INFO: querying 2500 Objects: 741 msec
     [ojb] [performance] INFO: querying 2500 Objects: 18 msec
     [ojb] [performance] INFO: fetching 2500 Objects: 289 msec
     [ojb] [performance] INFO: deleting 2500 Objects: 446 msec

     [ojb] Time: 18,964

     [ojb] OK (1 test)

    [jdbc] .[performance] INFO: Test for native JDBC
    [jdbc] [performance] INFO:
    [jdbc] [performance] INFO: inserting 2500 Objects: 651 msec
    [jdbc] [performance] INFO: updating 2500 Objects: 775 msec
    [jdbc] [performance] INFO: querying 2500 Objects: 616 msec
    [jdbc] [performance] INFO: querying 2500 Objects: 384 msec
    [jdbc] [performance] INFO: fetching 2500 Objects: 49 msec
    [jdbc] [performance] INFO: deleting 2500 Objects: 213 msec
    [jdbc] [performance] INFO:
    [jdbc] [performance] INFO: inserting 2500 Objects: 508 msec
    [jdbc] [performance] INFO: updating 2500 Objects: 686 msec
    [jdbc] [performance] INFO: querying 2500 Objects: 390 msec
    [jdbc] [performance] INFO: querying 2500 Objects: 360 msec
    [jdbc] [performance] INFO: fetching 2500 Objects: 46 msec
    [jdbc] [performance] INFO: deleting 2500 Objects: 204 msec
    [jdbc] [performance] INFO:
    [jdbc] [performance] INFO: inserting 2500 Objects: 538 msec
    [jdbc] [performance] INFO: updating 2500 Objects: 775 msec
    [jdbc] [performance] INFO: querying 2500 Objects: 384 msec
    [jdbc] [performance] INFO: querying 2500 Objects: 360 msec
    [jdbc] [performance] INFO: fetching 2500 Objects: 48 msec
    [jdbc] [performance] INFO: deleting 2500 Objects: 204 msec

    [jdbc] Time: 18,363

    [jdbc] OK (1 test)

    [odmg] .[performance] INFO: Test for ODMG-api
    [odmg] [performance] INFO:
    [odmg] [performance] INFO: inserting 2500 Objects: 12151 msec
    [odmg] [performance] INFO: updating 2500 Objects: 2937 msec
    [odmg] [performance] INFO: querying 2500 Objects: 4691 msec
    [odmg] [performance] INFO: querying 2500 Objects: 2239 msec
    [odmg] [performance] INFO: fetching 2500 Objects: 1633 msec
    [odmg] [performance] INFO: deleting 2500 Objects: 1815 msec
    [odmg] [performance] INFO:
    [odmg] [performance] INFO: inserting 2500 Objects: 2483 msec
    [odmg] [performance] INFO: updating 2500 Objects: 2868 msec
    [odmg] [performance] INFO: querying 2500 Objects: 3272 msec
    [odmg] [performance] INFO: querying 2500 Objects: 2223 msec
    [odmg] [performance] INFO: fetching 2500 Objects: 1038 msec
    [odmg] [performance] INFO: deleting 2500 Objects: 1717 msec
    [odmg] [performance] INFO:
    [odmg] [performance] INFO: inserting 2500 Objects: 2666 msec
```

```
     [odmg] [performance] INFO: updating 2500 Objects: 2841 msec
     [odmg] [performance] INFO: querying 2500 Objects: 2092 msec
     [odmg] [performance] INFO: querying 2500 Objects: 2161 msec
     [odmg] [performance] INFO: fetching 2500 Objects: 1036 msec
     [odmg] [performance] INFO: deleting 2500 Objects: 1741 msec

     [odmg] Time: 55,186
```

Some notes on these test results:

- You see a consistently better performance in the second and third run. this is caused by warming up effects of JVM and OJB.
- PB and native JDBC need about the same time for the three runs although JDBC performance is better for most operations. this is caused by the second run of the querying operations. In the second run OJB can load all objects from the cache, thus the time is **much** shorter. Hence the interesting result: if you have an application that has a lot of lookups, OJB can be faster than a native JDBC application!
- ODMG is much slower than PB or JDBC. This is due to the complex object level transaction management it is doing.
- You can see that for HSQLDB operations like insert and update are much faster with JDBC than with PB (60% and more). This ratio is so high, because HSQLDB is much faster than ordinary database servers (as it's inmemory). If you work against Oracle or DB2 the percentual OJB overhead is going down a lot (10 - 15 %), as the database latency is much longer than the OJB overhead.

It's easy to change target database. Please refer to this document for details (platforms.html) . Also it's possible to change the number of test objects by editing the ant-target in build.xml.

Another test compares PB-api,ODMG-api and native JDBC you can find next section.

5. OJB performance in multi-threaded environments

This test was created to check the performance and stability of the supported API's (PB-api, ODMG-api, JDO-api) in a multithreaded environment. Also this test compares the api's and native JDBC.
Running this test out of the box (a virgin OJB version against hsql) shouldn't cause any problems. To run the JDO-api test too, see JDO tutorial (../../docu/tutorials/jdo-tutorial.html) and comment in the test in target `perf-test` in `build.xml`

FIXME (arminw):
A test for JDO API is missed.

Per default OJB use hsql as database, by changing the JdbcConnectionDescriptor in the repository.xml file you can point to your specific RDBMS. Please refer to this document for

details (../../docu/guides/platforms.html) .

To run the multithreaded performance test call

```
ant perf-test
```

A typical output of this test looks like (OJB against hsql server, 2-tier, 100 MBit network):

```
[ojb] ========================================================
[ojb]         OJB PERFORMANCE TEST SUMMARY
[ojb] 10 concurrent threads, handle 2000 objects per thread
[ojb]         - performance mode
[ojb] ========================================================
[ojb]      API  Period   Total  Insert   Fetch  Update  Delete
[ojb]             [sec]   [sec]  [msec]  [msec]  [msec]  [msec]
[ojb] --------------------------------------------------------
[ojb]     JDBC   7.786   7.374    3951      76    2435     911
[ojb]       PB   9.807   8.333    5096     121    2192     922
[ojb]     ODMG  19.562  18.205    8432    1488    5064    3219
[ojb]      OTM  24.953  21.272   10688     223    4326    6033
[ojb] ========================================================

[ojb] PerfTest takes 191 [sec]
```

To change the test properties go to target `perf-test` in the `build.xml` file and change the program parameter.
The test needs five parameter:
- A comma separated list of the test implementation classes (no blanks!)
- The number of test loops
- The number of concurrent threads
- The number of managed objects per thread
- The desired test mode. `false` means run in performance mode, `true` means run in stress mode (useful only for developer to check stability).

```
<target name="perf-test" depends="prepare-testdb"
            description="Simple performance benchmark and stress test for
PB- and ODMG-api">
    <java fork="yes" classname="org.apache.ojb.performance.PerfMain"
          dir="${build.test}/ojb" taskname="ojb" failonerror="true" >
        <classpath refid="runtime-classpath"/>

        <!-- comma separated list of the PerfTest implementations -->
        <arg value=
        "org.apache.ojb.broker.OJBPerfTest$JdbcPerfTest,\
        org.apache.ojb.broker.OJBPerfTest$PBPerfTest,\
        org.apache.ojb.broker.OJBPerfTest$ODMGPerfTest,\
        org.apache.ojb.broker.OJBPerfTest$OTMPerfTest"
        />
        <!-- test loops, default was 3 -->
```

```
        <arg value="3"/>
        <!-- performed threads, default was 10 -->
        <arg value="10"/>
        <!-- number of managed objects per thread, default was 2000 -->
        <arg value="2000"/>
        <!-- if 'false' we use performance mode, 'true' we do run in stress
mode -->
        <arg value="false"/>

        <jvmarg value="-Xms128m"/>
        <jvmarg value="-Xmx256m"/>
    </java>
    <!-- do some cleanup -->
    <ant target="copy-testdb"/>
</target>
```

6. How OJB compares to other O/R mapping tools?

Many user ask this question and there is more than one answer. But OJB was shipped with a simple performance *"framework"* (independend from OJB) which allows a rudimentarily comparision of OJB with other (java-based) O/R mapping tools.
All involved classes can be found in dirctory *[db-ojb]/src/test* in package `org.apache.ojb.performance`.

Call `ant perf-test-jar` to build the jar file contain all necessary classes to set up a test with an arbitrary O/R mapper. After the build, the `db-ojb-XXX-performance.jar` can be found in `[db-ojb]/dist` directory.

Steps to set up the test for other O/R frameworks:

- Implement a class derived from `PerfTest` (../../api/org/apache/ojb/performance/PerfTest.html)
- Implement a class derived from `PerfHandle` (../../api/org/apache/ojb/performance/PerfHandle.html)
- [If persistent objects used within your mapping tool must be derived from a specific base class or must be implement a specific interface write your own persistent object class by implementing `PerfArticle` (../../api/org/apache/ojb/performance/PerfArticle.html) interface and **override method** `#newPerfArticle()` in your `PerfHandle` implementation class.
  Otherwise a default implementation of `PerfArticle` (../../api/org/apache/ojb/performance/PerfArticle.html) was used]

That's it!

You can find a example implementation called `org.apache.ojb.broker.OJBPerfTest` in the test-sources directory under

`[db-ojb]/src/test` (when using source-distribution). This implementation class is used to compare performance of the PB-API, ODMG-API, OTM-api and native JDBC.

See more section multi-threaded performance. `OJBPerfTest` is made up of inner classes. At each case two inner classes represent a test for one api (as described above).

Run the test
You have two possibilities to run the test:
a) Integration in the OJB build script
Add the full qualified class name of your PerfTest implementation class to the `perf-test` target of the OJB `build.xml` file, add all necessary jar files to `[db-ojb]/lib`. The working directory of the test is `[db-ojb]/target/test/ojb`.
b) Run PerfMain
It's possible to run the test using `org.apache.ojb.performance.PerfMain`.

```
java -classpath CLASSPATH org.apache.ojb.performance.PerfMain
[comma separated list of PerfTest implementation classes, no blanks!]
[number of test loops]
[number of threads]
[number of insert/fetch/delete loops per thread]
[boolean - run in stress mode if set true,
run in performance mode if set false, default false]
```

For example:

```
java -classpath CLASSPATH my.A_PerfTest,my.B_PerfTest 3 10 2000 false
```

This will use `A_PerfTest` and `B_PerfTest` and perform three loops each loop run 10 threads and each thread operate on 2000 objects. The test run in *performance* mode.

Take care of compareable configuration properties when run performance tests with different O/R tools (caching, locking, sequence key generation, connection pooling, ...).

Note:

Please, don't start flame wars by posting performance results to mailing lists made with this simple test. This test was created for OJB QA and to give a clue how good or bad OJB performs, NOT to start discussion like *XY is 12% faster then XZ*!!.

7. What are the best settings for maximal performance?

We don't know, that depends from the environment OJB runs (hardware, database, driver, application server, ...). But there are some settings which affect the performance:

- The API you use, e.g. PB-api is much faster then the ODMG-api. See which API (../../docu/faq.html#differencesBetweenAPI) for more information.
- ConnectionFactory implementation / Connection pooling. See connection pooling

(../../docu/faq.html) for more information.
- PersistentField class implementation.See OJB.properties section 'PersistentFieldClass' (../../OJB.properties.txt) for more information.
- Used sequence manager implementation. See sequence manager (../../docu/guides/sequencemanager.html) for more information.
- Use of batch mode (when supported by the DB). See repository.dtd element 'jdbc-connection-descriptor' (../../repository.dtd.txt) for more information.
- PersistenceBroker pool size. See OJB.properties (../../OJB.properties.txt) for more information.

To test the different settings use the tests of the performance test suite.