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. But on the other hand the biggest performance consumption is the database access itself (database performance, network performance, jdbc driver, driver settings, ...). So the percentage of O/R tool performance consumption isn't big.

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 all supported OJB API's against native single-threaded JDBC programming against your RDBMS of choice and run OJB API's in a virtual multithreaded environment. Further on it is possible to compare OJB against any O/R mapping tool using a simple *performance application*.

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` multi-threaded performance/stress test of PB/OTM/ODMG api against native JDBC
- `performance` single-threaded test, OJB API implementations (PB, ODMG) against native JDBC

By changing the JdbcConnectionDescriptor in the configuration files you can point to your specific RDBMS. Please refer to this document for details.

2.1. 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 ...

2.2. How OJB compares to native JDBC programming - single-threaded

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 (in-memory mode) shipped with OJB. A typical console output looks like follows:

```
performance:
     [ojb] .[performance] INFO: Test for PB-api
     [ojb] [performance] INFO:
     [ojb] [performance] INFO: inserting 1500 Objects: 188 msec
     [ojb] [performance] INFO: updating 1500 Objects: 265 msec
     [ojb] [performance] INFO: querying 1500 Objects: 0 msec
     [ojb] [performance] INFO: querying 1500 Objects: 0 msec
     [ojb] [performance] INFO: fetching 1500 Objects: 16 msec
     [ojb] [performance] INFO: deleting 1500 Objects: 63 msec
....
     [ojb] Time: 5,672
     [ojb] OK (1 test)

    [jdbc] .[performance] INFO: Test for JDBC
    [jdbc] [performance] INFO:
    [jdbc] [performance] INFO: inserting 1500 Objects: 157 msec
    [jdbc] [performance] INFO: updating 1500 Objects: 187 msec
    [jdbc] [performance] INFO: querying 1500 Objects: 94 msec
    [jdbc] [performance] INFO: querying 1500 Objects: 94 msec
    [jdbc] [performance] INFO: fetching 1500 Objects: 16 msec
    [jdbc] [performance] INFO: deleting 1500 Objects: 62 msec
....
    [jdbc] Time: 8,75
    [jdbc] OK (1 test)

    [odmg] .[performance] INFO: Test for ODMG-api
    [odmg] [performance] INFO:
    [odmg] [performance] INFO: inserting 1500 Objects: 266 msec
    [odmg] [performance] INFO: updating 1500 Objects: 359 msec
    [odmg] [performance] INFO: querying 1500 Objects: 531 msec
    [odmg] [performance] INFO: querying 1500 Objects: 531 msec
    [odmg] [performance] INFO: fetching 1500 Objects: 47 msec
    [odmg] [performance] INFO: deleting 1500 Objects: 125 msec
....
    [odmg] Time: 13,75
    [odmg] OK (1 test)
```

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 (without caching extensions)!
- ODMG is much slower than PB or JDBC. This is due to the complex object level transaction

management it is doing and the fact that ODMG doesn't have a specific method to lookup objects by it's identity. The second reason is responsible for slow *querying* results, because in test always a complex query is done for each object. It is possible to use the PB-api within ODMG, then the query by identity will be as fast as in PB-api.

- You can see that for HSQLDB operations like insert and update are faster with JDBC than with PB. This performance difference strongly depends on the used cache implementation and can rise up to 50% when the cache operate on object copies. This ratio is so high, because HSQLDB was running with *in memory mode* and is much faster than ordinary database servers. 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.
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.

2.3. 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 and future API's) in a single/multithreaded environment and to compare the different api's against native JDBC calls.
Running this test out of the box (a virgin OJB version against hsql) shouldn't cause any problems.

Per default OJB use a in-memory hsql database, by changing the JdbcConnectionDescriptor in the *repository.xml* file or modify the *build.properties* file when running OJB out of the box you can point to your specific RDBMS. Please refer to this document for details.

To run the multithreaded performance test call

```
   ant perf-test
```

A typical output of this test, using OJB against in-memory hsql looks like this

```
[ojb]
[ojb]             OJB PERFORMANCE TEST SUMMARY, Thu Dec 29 23:42:20 CET 2005
[ojb]
[ojb]   12 concurrent threads, handle 500 objects per thread
[ojb]   500 INSERT operations per test instance
[ojb]   FETCH collection of 500 objects per test instance
[ojb]   Repeat FETCH collection of 500 objects per test instance
[ojb]   125 get by Identity calls  per test instance
[ojb]   500 UPDATE operations per test instance
[ojb]   500 DELETE operations per test instance
[ojb]   - performance mode - results per test instance (average)
[ojb]
[ojb] API         Total       Insert      Fetch       Fetch 2     by Id
Update      Delete
[ojb]               [%]         [msec]      [msec]      [msec]      [msec]
[msec]      [msec]
[ojb]
[ojb] JDBC          100         475(100%)   26(100%)    23(100%)    209(836%)
477(100%)   197(100%)
[ojb] PB            203         1341(282%)  153(588%)   151(656%)   25(100%)
648(135%)   239(121%)
[ojb] ODMG          250         1469(309%)  104(400%)   105(456%)   527(2108%)
800(167%)   569(288%)
[ojb]
[ojb] PerfTest takes 70 [sec]
```

This test run shows the overhead caused by the O/R layer compared to handcoded sql statements. Most overheads results in populate the two-level cache which is useless when using a in-memory database.

Below you can see the same test against *MaxDB* running on the same machine

```
....
[ojb]
================================================================================================
[ojb] API          Total       Insert      Fetch       Fetch 2     by Id
Update      Delete
[ojb]              [%]         [msec]      [msec]      [msec]      [msec]
[msec]      [msec]
[ojb]
------------------------------------------------------------------------------------------------
[ojb] JDBC         100         5855(100%)  55(100%)    38(100%)    1628(5087%)
5588(184%)  4084(136%)
[ojb] ODMG         117         12043(205%) 180(327%)   294(773%)   754(2356%)
3027(100%)  2988(100%)
[ojb] PB           123         11577(197%) 94(170%)    84(221%)    32(100%)
4240(140%)  3193(106%)
[ojb]
================================================================================================
[ojb] PerfTest takes 440 [sec]
```

Note:

The *performance test* output is written to console and in a file called *OJB-Performance-Result.txt*.

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="${build.test}/ojb" taskname="ojb" failonerror="true" >
        <classpath refid="runtime-classpath"/>
        <!-- comma separated list of the PerfTest implementations -->
        <arg value=
        "org.apache.ojb.compare.OJBPerfTest$JdbcPerfTest,
        org.apache.ojb.compare.OJBPerfTest$PBPerfTest,
        org.apache.ojb.compare.OJBPerfTest$ODMGPerfTest"
        />
        <!-- test loops, default was 6 -->
        <arg value="6"/>
        <!-- performed threads, default was 12 -->
        <arg value="12"/>
        <!-- number of managed objects per thread, default was 500 -->
        <arg value="500"/>
        <!-- if 'false' we use performance mode, 'true' we do run in stress mode
-->
        <arg value="false"/>
        <!-- if 'true' all log messages will be print -->
        <arg value="true"/>
        <jvmarg value="-Xms128m"/>
        <jvmarg value="-Xmx256m"/>
```

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

3. 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 application* (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`
- 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` interface and **override method** `#newPerfArticle()` in your `PerfTest` implementation class.
  Otherwise a default implementation of interface `PerfArticle` was used.
  The default implementation class is
  `org.apache.ojb.performance.PerfArticleImpl`.

That's it!

You can find a example implementation called `org.apache.ojb.compare.OJBPerfTest` in the test-sources directory under `[db-ojb]/src/test` (when using source-distribution).

This implementation class is used to compare the performance of the PB-API, ODMG-API, OTM-api and native JDBC (to bunch all API's, for each API a static inner implementation class of `PerfTest` was used). See more about multi-threaded performance.

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]
[optional boolean - run in stress mode if set true,
run in performance mode if set false, default false]
[optional boolean - if 'true' all log messages will be print, else only a test
summary, default 'true']
```

For example:

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

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 *normal* mode and log all messages.

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*!!.

4. 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 for more information.
- The used cache implementation.
- ConnectionFactory implementation / Connection pooling. See connection pooling for more information.
- The *autocommit* setting of used connections. For best performance it's recommended to use *autocommit* 'false' setting in the *jdbc-driver* to avoid `Connection.setAutoCommit(...)` calls by OJB.
- `PersistentField` class implementation.See OJB.properties section 'PersistentFieldClass' to change the implementation.
- Used sequence manager implementation. See sequence manager for more information.
- Use of *batch mode* (when supported by the DB). See *jdbc-connection-descriptor* *batch-mode* attribute for more information.
- PersistenceBroker pool size. See OJB.properties for more information.
- The *JDBC driver* settings (e.g. statement caching on/off).

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