Best practis to use CastorJDO

To not loose any information I copied together all informations from a corresponding thread on the mailing list.

I know that Gregory Block (another castor commiter) is using castor in a high-volume application. Even if I would call my appliaction a low-volume one, I also had to resolve some locking problems that I could track down to negligences at backgroud threads of my application. There are also some performance bottlenecks we are aware of and are working to solve them at our next refactoring steps at castor. For some of them we have working patches or workarounds available (will come to them later).

I'll start with some general suggestions that you should have a look at. Please don't feel upset if some are really primitive but there may be users listening that are not aware of them.

1. Switch to version 0.9.6 of castor as we have fixed some bugs that may cause some of your problems.

Sidenote: Performance has, generally, improved recently. If you're not seeing performance improvements, then it's worth spending some time thinking about why.

2. Initialize your JDO or JDO2 (will be renamed to JDOManager at next release) instance once and reuse it all over your application. Don't reuse the Database instances.

Again: Never, ever reuse a database instance. Creating them is inexpensive, and JDBC rules state that one thread -> one JDBC connection is the rule. Do not multithread inside of a Database instance; as a corrolary, do not multithread on a single JDBC connection.

3. Use a Datasource instead of a Driver configuration as they enable connection pooling which gives you a great performance improvement.

I highly suggest DBCP, here, with the beneficial use of prepared statement caching.

Should you be running on a system where read performance is critical, feel free to take the SQL code generated by castor, and dumped to logs during the DB mapping load in debug output, and turn those into stored procedures that you then invoke via call to perform those loads; however, I find personally that stored procedures would be a minimal improvement over the DBCP prepared statement cache; your mileage may vary. db.load() has performance benefits that are worth keeping, IMO, and the pleasure of having pretty stored procedures in your database is far outweighed by the nightmare of change management.

have a look at

http://castor.codehaus.org/pooling.html#Jakarta-Commons-DBCP---BasicDataSource

which has details about how to use and configure DBCP with Castor and Tomcat. Wrt 'prepared statement cache', Gregory seems to be referring to the fact that DBCP is JDBC 3.0-compliant product as as such has to support caching of prepared statements. This basically allows the JDBC driver to maintain a pool of prepared statements across all connections.

DBCP setup is generally outside of the scope of this list, but basically, here's my two cent description:

1) Use tomcat 5.5, because mucking about in server.xml sucks.
For those of you working with Tomcat 4.1.x, there's no need to muck about in server.xml, either. Afaik, a web app can be deployed using a web app descriptor copied into $TOMCAT_HOME/webapps, which is the place top define anything specific to a web app context, as outlined by Greg below. Details can, of course,

2) Create a META-INF directory in your WAR deploy scripts, and put a context.xml in it.
3) In that context.xml, describe all of the things you want to be made available via JNDI to your application. These include things like UserTransaction and TransactionManager (for those of us using JOTM), all your database connection pools as datasources, etc. You can also add your JDO factory here, should you choose to do so.
4) Configure Castor to load those JNDI names to retrieve connections.

Hit the deploy button, and bob's your uncle.

4. Always commit or rollback your transactions and close your Database instances properly also in fail situations as suggested by Nick previously.

Just the obvious general rule on Java objects that hold resources: Don't wait for the VM to finalize to have something happen to your objects when you could have released critical resources at the appropriate point in the codebase.

5. Keep your transactions as short as possible. If you have an open transaction that holds a write lock on an object no other transaction can get a write lock on the same object which will lead to a LockNotGrantedException.

```
execute() {
Database db = jdo.getDatabase();
db.begin();
// query objects from database with read only
db.commit();
db. close();

// do some time consuming processing with the data

Database db = jdo.getDatabase();
db.begin();
```

```
// use db.load() to load the objects you need to change again
// create, update or delete some objects
db.commit();
db. close();
}
```

It doesn't make sense to make a own transaction for every change you want to do to an object as this will slow down your application. On the other hand if you have transactions with lots of objects involved taking an valuable amonth of time you may consider to split this transactions to reduce the time an object is locked.

Also keep in mind that folks using lockmode of dblocked do FOR UPDATE calls on things they read while the transaction is open; if you're using dblocked mode, be aware of how your application does things. If you're in one of the other modes, locks happen inside castor, and it's your responsibility to always use the right access mode when accessing content.

If you can, for example, decide at the API layer whether or not an operation is going to ever need to modify an object, and know that you will only ever use an instance in read only mode, load objects with access mode read only, and not shared.

Limit use of read-write objects to situations in which it is likely you will need to perform updates.

Read-write performance will change dramatically once the TransactionContext patches have been checked in; if you'd like to guinea pig them, check JIRA and try the patch out; we're already using it in production here.

Imagine, for a moment, that these transactions were in DBLocked mode - transactions which translate directly into locks on the database.

If you're opening something up for modification on the DB - marking it as select FOR UPDATE - then that row will be locked until you commit. The database would prevent any other transaction that wants to touch that row from doing anything to it, and it would block on your transaction - deadlock at the SQL level.

Castor does the same things internally for its own access modes - Shared and Exclusive. Each has different locking semantics; having good performance means understanding those locking semantics.

For example - read only transactions (should be) cheap. (at least they are post-patch.) So there's no issue with holding those transactions open a long time; because they only translate, for an instant, into a lock. The lock is released the moment the load is completed and the object is dropped into read-only state within your transaction; read only operations therefore operate, pretty much, without locking.

The lock is of course acquired because you might also have it in SHARED or EXCLUSIVE mode on another thread - and that read-only operation isn't safe until those transactions close.

Once the lock is released, you're lock-free again, so the transaction basically has nothing in it (post-patch) that needs anything doing.

That's not to say that holding transactions open is good practice - but transactions should always be thought of as cheap to create and destroy and expensive to hold on to - never do heavy computation inside of one, unless you're willing to live with the consequences that arise from holding transactions on object sets that others might need to access.

6. Query or load your objects read only whenever possible. Even if castor creates a lock on them this does not prevent other threads from reading or writing them. Read only queries are also about 7 times faster compared with default shared mode.

for queries:

```
String oql = "select o from FooBar o";
Query query = db.getOQLQuery(oql);
QueryResults results = query.execute(Database.ReadOnly);
```

to load an object by its identity:

```
Integer id = new Integer(7);
Foo foo = (Foo) db.load(Foo.class, id, Database.ReadOnly);
```

Default accessmode is evaluated as follows:
if specified castor uses access mode from db.load() or query.execute(),
if this is not available it takes access mode specified in class mapping,
if nothing is specified in mapping it defaults to shared.

Cannot stress how important this is: If 99% of your application never writes an object, and you as a programmer know it won't, then do something about it. If you're in a situation where you want the object to be read-only most of the time, and only want a writable every now and then, do so just-in-time by performing a load-modify- store operation in a single transaction for the shareable you want.

In other words: Don't use read-write objects unless you know you're likely to want to write them.

7. If there is a possibility you should prefer db.load(Class, object) over db.execute(String). I suggest that as db.load() first tries to load the requested object from cache and only retrieves it from database when it is not availble there. When executing queries with db.execute() the object will always be loaded from database without looking at the cache. You may gain a improvement by a factor of 10 and more when changing from db.execute() to db.load().

I hope above suggestions help you to resolve the problems you have. If you still need more performance there are 2 areas of improvement that

are more difficult to resolve.

a. If you have a look at http://jira.codehaus.org/browse/CASTOR-1085 where a patch to TransactionContext is attached that improves read/write performance with a factor of 3. Even if the patch passes all tests of castor test framework it requires more testing before we will integrate it in our next major release. As stated in the comment Gregory will use the patch in his production environment sooon.

It's in production now; and several large content imports of guide content have been run through it without any difficulties or problems in the generated content.

b. I will attache a test that shows how read only performance can be improved to http://jira.codehaus.org/browse/CASTOR-732 this evening.

Now, there's lots left to do - there is still the issue, for example, of dependent objects being slightly sub-optimal in performance both in terms of the SQL that gets generated and the way it gets managed - but there will be improvements over time to the way that this and other operations are performed.

But performance **should be good right now**. If it isn't, you'll need to think about whether you are using the optimal set of operations. No environment can predict your requirements - hinting to the system when objects can be safely assumed to be read-only is vital to a high-performance implementation.