

Using Object Database db4o as Storage Provider in Voldemort

by German Viscuso

db4objects (a division of Versant Corporation)

<german@db4o.com>

September 2010

Abstract:

In this article I will show you how easy it is to add support for Versant's object database, [db4o](#) in project Voldemort, an Apache licensed distributed key-value storage system used at [LinkedIn](#) which is useful for certain high-scalability storage problems where simple functional partitioning is not sufficient (Note: Voldemort borrows heavily from [Amazon's Dynamo](#), if you're interested in this technology all the available papers about Dynamo should be useful).

Voldemort's storage layer is completely mockable so development and unit testing can be done against a throw-away in-memory storage system without needing a real cluster or, as we will show next, even a real storage engine like db4o for simple testing.

Note: All db4o related files that are part of this project are publicly available on GitHub: <http://github.com/germanviscuso/voldemort/tree/master/contrib/db4o/>

Index:

1. Voldemort in a Nutshell
2. Key/Value Storage System
 - a. Pros and Cons
3. Logical Architecture
4. db4o API in a Nutshell
5. db4o as Key/Value Storage Provider
6. db4o and BerkeleyDB side by side
 - a. Constructors
 - b. Fetch by Key
 - c. Store Key/Value Pair
 - d. Status of Unit Tests
7. Conclusion

Voldemort in a Nutshell

In one sentence Voldemort is basically *a big, distributed, persistent, fault-tolerant hash table designed to tradeoff consistency for availability.*

It's implemented as a highly available key-value storage system for services that need to provide an "always-on" experience. To achieve this level of availability, Voldemort sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and

application-assisted conflict resolution.

Voldemort targets applications that operate with weak/eventual consistency (the “C” in ACID) if this results in high availability even under occasional network failures. It does not provide any isolation guarantees and permits only single key updates. It is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously.

The system uses a series of known techniques to achieve scalability and availability: data is partitioned and replicated using consistent hashing, and consistency is facilitated by object versioning. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. It employs a gossip based distributed failure detection and membership protocol and is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from without requiring any manual partitioning or redistribution.

Voldemort is not a relational database, it does not attempt to satisfy arbitrary relations while satisfying ACID properties. Nor is it an object database that attempts to transparently map object reference graphs. Nor does it introduce a new abstraction such as document-orientation.

However, given the current complex scenario of persistence solutions in the industry we’ll see that different breeds can be combined to offer solutions that become the right tool for the job at hand. In this case we’ll see how a highly scalable NoSQL solution such as Voldemort can use a fast Java based embedded object database (Versant’s db4o) as the underlying persistence mechanism. Voldemort provides the scalability while db4o provides fast key/value pair persistence.

Key/Value Storage System

To enable high performance and availability Voldemort allows only very simple key-value data access. Both keys and values can be complex compound objects including lists or maps, but none-the-less the only supported queries are effectively the following:

```
value = store.get( key )  
store.put( key, value )  
store.delete( key )
```

This is clearly not good enough for all storage problems, there are a variety of trade-offs:

Cons

- no complex query filters
- all joins must be done in code
- no foreign key constraints
- no triggers/callbacks

Pros

- only efficient queries are possible, very predictable performance
- easy to distribute across a cluster
- service-orientation often disallows foreign key constraints and forces joins to be done in code anyway (because key refers to data maintained by another service)
- using a relational db you need a caching layer to scale reads, the caching layer typically forces you into key-value storage anyway
- often end up with xml or other denormalized blobs for performance anyway
- clean separation of storage and logic (SQL encourages mixing business logic with storage operations for efficiency)
- no object-relational miss-match, no mapping

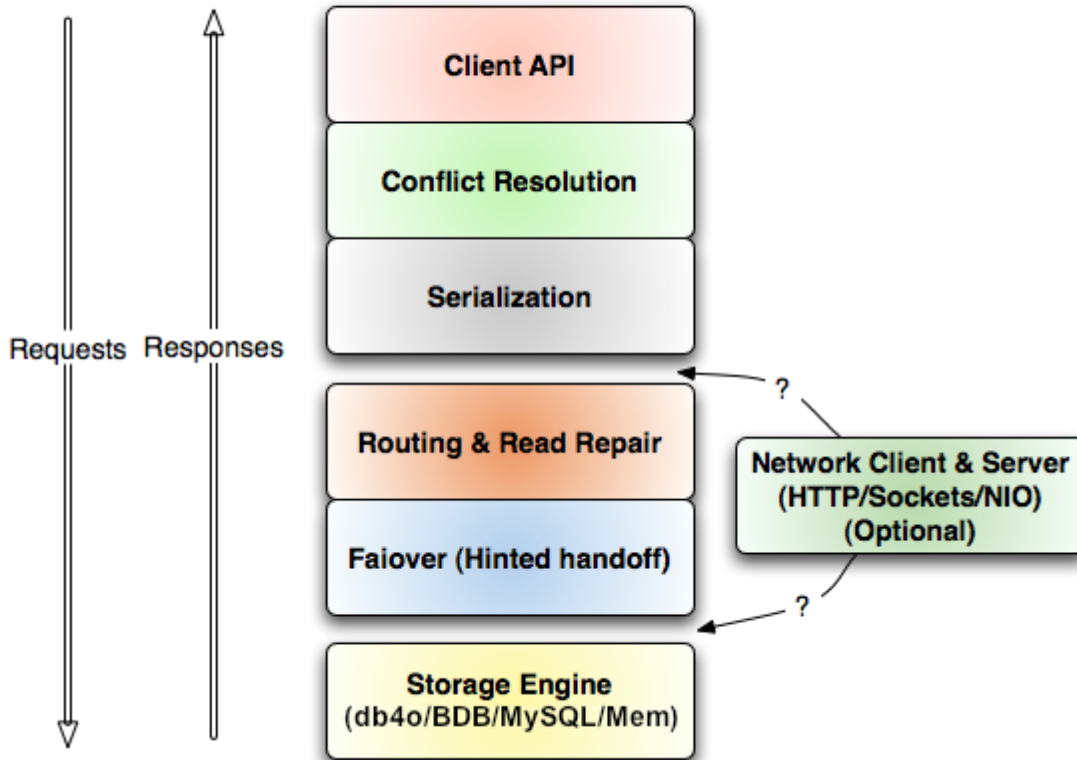
Having a three operation interface makes it possible to transparently mock out the entire storage layer and unit test using a mock-storage implementation that is little more than a HashMap. This makes unit testing outside of a particular container or environment much more practical and was certainly instrumental in simplifying the db4o storage engine implementation.

Logical Architecture

In Voldemort, each storage node has three main software component groups: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

If we take a closer look we'll see that each layer in the code implements a simple storage interface that does **put**, **get**, and **delete**. Each of these layers is responsible for performing one function such as TCP/IP network communication, serialization, version reconciliation, inter-node routing, etc. For example the routing layer is responsible for taking an operation, say a PUT, and delegating it to all the N storage replicas in parallel, while handling any failures.

Logical Architecture



Voldemort's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Oracle's Berkeley Database (BDB), Oracle's MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Voldemort's local persistence engine based on their object size distribution. The majority of Voldemort's production instances currently use BDB.

In this article we'll provide details about the creation of a new storage engine (yellow box layer on the image above) that uses db4o instead of Oracle's Berkeley DB (BDB), Oracle's MySQL or just memory. Moreover we'll show that db4o's performance is on-par with BDB (if not better) while introducing a significant reduction in the implementation complexity.

db4o API in a Nutshell

db4o's basic API is not far away from Voldemort's API in terms of simplicity but it's not limited to the storage of key/value pairs. db4o is a general purpose object persistence engine that can deal with the peculiarities of native POJO persistence that exhibit arbitrarily complex object

references.

The very basic API looks like this:

```
container.store( object )  
container.delete( object )  
objectSet = container.queryByExample( prototype )
```

Two more advanced querying mechanisms are available and we will introduce one of them (SODA) on the following sections. As you can see, in order to provide an interface for key/value storage (what Voldemort expects) we'll have to provide an intermediate layer which will take requests via the basic Voldemort API and will translate that into db4o API calls.

db4o as Key/Value Storage Provider

db4o fits seamlessly into the Voldemort picture as a storage provider because:

- db4o allows free form storage of objects with no special requirements on object serialization which results in a highly versatile storage solution which can easily be configured to store from low level objects (eg byte arrays) to objects of arbitrarily complexity (both for keys and values on key/value pairs)
- db4o's b-tree based indexing capabilities allow for quick retrieval of keys while acting as a key/value pair storage provider
- db4o's simple query system allows the retrieval of key/value pairs with one-line of code
- db4o can persist versioned objects directly which frees the developer from having to use versioning wrappers on stored values

db4o can act as a simpler but powerful replacement to Oracle's Berkeley DB (BDB) because the complexity of the db4o storage provider is closer to Voldemort's "in memory" storage provider than the included BDB provider which results in faster test and development cycles (remember Voldemort is work in progress).

As a first step in the implementation we provide a generic class to expose [db4o as a key/value storage provider](#) which is not dependant on Voldemort's API (also useful to db4o developers who need this functionality in any application). The class is `voldemort.store.db4o.Db4oKeyValueProvider<Key, Value>` and relies on Java generics so it can be used with any sort of key/value objects.

This class implements the following methods which are more or less self explanatory considering they operate on a store of key/value pairs:

```
voldemort.store.db4o.Db4oKeyValueProvider.keyIterator()  
voldemort.store.db4o.Db4oKeyValueProvider.getKeys()  
voldemort.store.db4o.Db4oKeyValueProvider.getValues(Key)  
voldemort.store.db4o.Db4oKeyValueProvider.pairIterator()
```

```

voldemort.store.db4o.Db4oKeyValueProvider.get (Key)
voldemort.store.db4o.Db4oKeyValueProvider.delete (Key)
voldemort.store.db4o.Db4oKeyValueProvider.delete (Key, Value)
voldemort.store.db4o.Db4oKeyValueProvider.delete (Db4oKeyValuePair<Key,
Value>)
voldemort.store.db4o.Db4oKeyValueProvider.put (Key, Value)
voldemort.store.db4o.Db4oKeyValueProvider.put (Db4oKeyValuePair<Key, Value>)
voldemort.store.db4o.Db4oKeyValueProvider.getAll ()
voldemort.store.db4o.Db4oKeyValueProvider.getAll (Iterable<Key>)
voldemort.store.db4o.Db4oKeyValueProvider.truncate ()
voldemort.store.db4o.Db4oKeyValueProvider.commit ()
voldemort.store.db4o.Db4oKeyValueProvider.rollback ()
voldemort.store.db4o.Db4oKeyValueProvider.close ()
voldemort.store.db4o.Db4oKeyValueProvider.isClosed ()

```

Second, we implement a [db4o StorageProvider following Voldemort's API](#), namely

`voldemort.store.db4o.Db4oStorageEngine` which define the key (K) as `ByteArray` and the value (V) as `byte []` for key/value classes.

Since this class inherits from abstract class `voldemort.store.StorageEngine<K, V>` it must (and it does) implement the following methods:

```

voldemort.store.db4o.Db4oStorageEngine.getVersions (K)
voldemort.store.db4o.Db4oStorageEngine.get (K)
voldemort.store.db4o.Db4oStorageEngine.getAll (Iterable<K>)
voldemort.store.db4o.Db4oStorageEngine.put (K, Versioned<V>)
voldemort.store.db4o.Db4oStorageEngine.delete (K, Version)

```

We also implement the class that handles the [db4o storage configuration](#):

`voldemort.store.db4o.Db4oStorageConfiguration`

This class is responsible for providing all the initialization parameters when the db4o database is created (eg. index creation on the key (K) field for fast retrieval of key/value pairs)

Third and last we provide three support classes:

`voldemort.store.db4o.Db4oKeyValuePair<Key, Value>` : a generic key/value pair class. Instances of this class will be what ultimately gets stored in the db4o database.

`voldemort.store.db4o.Db4oKeysIterator<Key, Value>` : an Iterator implementation that allows to iterate over the keys (K).

`voldemort.store.db4o.Db4oEntriesIterator<Key, Value>` : an Iterator implementation that allows to iterate over the values (V).

and a few test classes:

`voldemort.CatDb4oStore`

```
voldemort.store.db4o.Db4oStorageEngineTest (this is the main test class)
voldemort.store.db4o.Db4oSplitStorageEngineTest
```

db4o and BerkeleyDB side by side

Let's take a look at db4o's simplicity by comparing the matching methods side by side with BDB (now both can act as Voldemort's low level storage provider).

Constructors: BdbStorageEngine() vs Db4oStorageEngine()

The db4o constructor gets rid of the serializer object because it can store objects of any complexity with no transformation. The code:

```
this.versionSerializer = new Serializer<Version>() {
    public byte[] toBytes(Version object) {
        return ((VectorClock) object).toBytes();
    }
    public Version toObject(byte[] bytes) {
        return versionedSerializer.getVersion(bytes);
    }
};
```

is no longer necessary and, of course, this impacts all storage operations because db4o requires no conversions “to bytes” and “to object” back and forth:

BDB (fetch by key):

```
DatabaseEntry keyEntry = new DatabaseEntry(key.get());
DatabaseEntry valueEntry = new DatabaseEntry();
List<T> results = Lists.newArrayList();

for(OpStatus status = cursor.getSearchKey(keyEntry, valueEntry, lockMode);
    status == OperationStatus.SUCCESS;
    status = cursor.getNextDup(keyEntry, valueEntry, lockMode)) {
    results.add(serializer.toObject(valueEntry.getData()));
}
return results;
```

db4o (fetch by key)

```
Query query = getContainer().query();
query.constrain(Db4oKeyValuePair.class);
query.descend("key").constrain(key);
return query.execute();
```

in this example we use “SODA”, a low level and powerful graph based query system where you basically build a query tree and pass it to db4o for execution.

BDB (store key/value pair)

```
DatabaseEntry keyEntry = new DatabaseEntry(key.get());
boolean succeeded = false;
transaction = this.environment.beginTransaction(null, null);
// Check existing values
// if there is a version obsoleted by this value delete it
// if there is a version later than this one, throw an exception
DatabaseEntry valueEntry = new DatabaseEntry();
cursor = getBdbDatabase().openCursor(transaction, null);

for (OpStatus status=cursor.getSearchKey(keyEntry, valueEntry, LockMode.RMW);
     status == OperationStatus.SUCCESS;
     status = cursor.getNextDup(keyEntry, valueEntry, LockMode.RMW)) {

        VectorClock clock = new
VectorClock(valueEntry.getData());
        Occured ocured = value.getVersion().compare(clock);
        if(ocured == Occured.BEFORE)
            throw new ObsoleteVersionException();
        else if(ocured == Occured.AFTER)
            // best effort delete of obsolete previous value!
            cursor.delete();
    }

// Okay so we cleaned up all the prior stuff, so now we are good to
// insert the new thing
valueEntry = new DatabaseEntry(versionedSerializer.toBytes(value));
OperationStatus status = cursor.put(keyEntry, valueEntry);
if(status != OperationStatus.SUCCESS)
    throw new PersistenceFailException("Put operation failed:" + status);
succeeded = true;
```

db4o (store key/value pair)

```
boolean succeeded = false;
candidates = keyValueProvider.get(key);

for (Db4oKeyValuePair<ByteArray, Versioned<byte[]>> pair: candidates) {
    Occured ocured = value.getVersion().compare(pair.getValue().getVersion())
};
    if(ocured == Occured.BEFORE)
        throw new ObsoleteVersionException();
    else if(ocured == Occured.AFTER)
        // best effort delete of obsolete previous value!
        keyValueProvider.delete(pair);
}

// Okay so we cleaned up all the prior stuff, so we can now insert
```



```

try {
    keyValueProvider.put(key, value);
} catch(Db4oException de) {
    throw new PersistenceFailException("Put op failed:" + de.getMessage());
}
succeeded = true;

```

Status of Unit Tests

Let's take a look at the current status of unit tests in for the db4o storage provider

(*Db4oStorageEngineTest*):

1. **testPersistence**: single key/value pair storage and retrieval with storage engine close operation in between (it takes some time because this is the first test and the Voldemort system is kick started).
2. **testEquals**: test that retrieval of storage engine instance by name gets you the same instance.
3. **testNullConstructorParameters**: null constructors for storage engine instantiation are illegal. Arguments are storage engine name and database configuration.
4. **testSimultaneousIterationAndModification**: threaded test of simultaneous puts (inserts), deletes and iteration over pairs (150 puts and 150 deletes before start of iteration).
5. **testGetNoEntries**: test that empty storage empty returns zero pairs.
6. **testGetNoKeys**: test that empty storage empty returns zero keys.
7. **testKeyIterationWithSerialization**: test storage and key retrieval of 5 pairs serialized as Strings.
8. **testIterationWithSerialization**: test storage and retrieval of 5 pairs serialized as Strings.
9. **testPruneOnWrite**: stores 3 versions for one key and tests prune (overwriting of previous versions should happen).
10. **testTruncate**: stores 3 pairs and issues a truncate database operation. Then verifies db is empty.
11. **testEmptyByteArray**: stores 1 pair with zeroed key and tests correct retrieval.
12. **testNullKeys**: test that basic operations (get, put, getAll, delete,etc) fail with a null key parameter (5 operations on pairs total).
13. **testPutNullValue**: test that put operation works correctly with a null value (2 operations on pairs total, put and get).
14. **testGetAndDeleteNonExistentKey**: test that key of non persistent pair doesn't return a value.
15. **testFetchedEqualsPut**: stores 1 pair with complex version and makes sure only one entry is stored and retrieved.
16. **testVersionedPut**: tests that obsolete or equal versions of a value can't be stored. Test correct storage of incremented version (13 operations on pairs total).
17. **testDelete**: puts 2 pairs with conflicting versions and deletes one (7 operations on pairs total).

- 18. **testGetVersions**: Test retrieval of different versions of pair (3 operations on pairs total).
- 19. **testGetAll**: puts 10 pairs and tests if all can be correctly retrieved at once (getAll)
- 20. **testGetAllWithAbsentKeys**: same as before but with non persistent keys (getAll returns 0 pairs).
- 21. **testCloseIsIdempotent**: tests that a second close does not result in error.

Test	db4o (ms)	BDB (ms)
testPersistence	6.189	4.711
testEquals	0.789	0.246
testNullConstructorParameters	0.356	0.199
testSimultaneousIterationAndModification	2.103	0.503
testGetNoEntries	0.034	0.064
testGetNoKeys	0.227	1.82
testKeyIterationWithSerialization	0.757	1.026
testIterationWithSerialization	0.27	0.176
testPruneOnWrite	0.581	1.179
testTruncate	0.244	0.048
testEmptyByteArray	0.031	0.117
testNullKeys	0.204	1.318
testPutNullValue	0.175	0.24
testGetAndDeleteNonExistentKey	0.159	0.155
testFetchedEqualsPut	0.236	1.551
testVersionedPut	0.356	0.076
testDelete	0.384	0.279
testGetVersions	0.306	0.403
testGetAll	0.315	3.361
testGetAllWithAbsentKeys	0.537	0.682
testCloseIsIdempotent	0.048	0.111
Total time	14.301	18.265

Conclusion

db4o provides a low level implementation of a key/value storage engine that is both simpler than BDB and on-par (if not better) in performance. Moreover, the implementation shows that different persistence solutions such as the typical highly scalable, eventually consistent NoSQL engine and object databases can be combined to provide a solution that becomes the right tool for the job at hand which makes typical “versus” arguments obsolete.

Finally the short implementation path taken to make db4o act as a reliable key/value storage provider show the power and versatility of native embedded object databases.