Example-driven ZODB

How to use an object database with a Python, a dynamic object-oriented language

Noah Gift
Brandon Craig Rhodes

April 15, 2008

Relational databases are not the only solution available for Python programmers in the enterprise. Often an object database can be a more natural fit for solving certain problems. This article discusses ZODB, a scalable and redundant object database that specializes in storing extensible objects, without the natural Object-relational impedance mismatch that can occur by attempting to make an Object Oriented Language and a Relational Query System map objects to relations.

Introduction

Just like their friends who program in statically typed languages, programmers of Python and in other dynamic languages often use relational databases as a back-end data store. Whether they directly define database tables using SQL or instead use a schema language that their framework or ORM provides as an alternative, all such solutions offer a similar workflow: the application designer must specify the name, type, and constraint of each attribute than an instance of each class will support, and in return, database relations are instantiated that can persist instances of those objects.

Over a decade ago, the Zope Corporation first offered an alternative, which they designed to better fit the data model of a dynamic language: the Zope Object Database (ZODB), which specializes in the storage of extensible objects rather than classical relations. Today it is the database engine that most often underlies Zope-powered Web applications.

The decision of whether to use the ZODB or an ORM atop a relational database involves important trade-offs, which we will explore in this article, along with a basic guide to the maintenance of a production ZODB instance.

ZODB tips from Jim Fulton

ZODB includes a BTree implementation that allows relation-like structures to be defined and used. Zope "catalogs" can be used much like relational tables. The object used for the root object isn't very scalable. Large collections should use BTrees placed in the root object (or below, of course). Also, the keys of the root object aren't limited to strings.

Who is Jim Fulton?

Jim Fulton is the creator and one of the maintainers of the Zope Object Database. Jim is also one of the creators of the Zope Object Publishing Environment and the CTO at Zope Corporation.

What Jim Fulton has to say about ZODB

ZODB is to relational databases as Python is to statically-type languages such as Java™. For many applications, especially applications involving complex objects, a database like the ZODB is a lot easier to deal with. This is a key reason why ZODB is the most popular back end for Zope applications.

Another important difference between object databases and relational databases is that object databases store already assembled objects. In a relational databases, objects that can't be represented as records of basic data values must be assembled through database joins. This can be very expensive and cumbersome, both conceptually and computationally.

The ZODB takes a minimalist approach to the services it provides. It provides basic persistence and little else. Other services, like security and indexing, must be provided at the application level. This approach has advantages as disadvantages. It allows ZODB developers to focus on core services and makes it easier to innovate, but it leaves some people wanting more. The lack of a database-level security model makes it difficult to fully leverage ZEO storage servers as clients have more or less unfettered access to the database and must therefore be trusted.

It is fairly straightforward to build indexing facilities for ZODB, but there is no query language other than Python. For example, to select people named Smith from a collection of people, we can use a Python list comprehension:

```
        [p for p in people if p.last_name == 'Smith']
```

But this is a linear search. It would be *nice* to have a query language that could introspect and indexing structures. On the other hand, if queries are ad-hoc, it is unlikely that indexes will be available. In an OODB, most non-ad-hoc queries are unnecessary.

Features of ZODB

There are probably two features of the ZODB that are most striking to someone approaching it after experience with relational databases: first, that in the ZODB you store objects in a hierarchy; and, second, that ZODB objects have absolutely no schema and can gain or lose instance attributes on the fly. Let's look at each of these features in turn.

All objects in a ZODB database are part of a hierarchy that begins with the "root object" - a fairly generic container object created automatically by the database engine to contain all further objects in the database. It acts like a Python dictionary that allows objects to be placed at (and later deleted from) names identified by string-typed keys. So the root object can be used directly if the programmer has no greater ambition than to store some objects that are indexed by a simple or delimited text string, similar to the way that old-fashioned Berkeley databases can be used to create inexpensive and persistent associative arrays.

But the ZODB really comes into its own when container objects are placed into the root object which can then, in turn, hold other objects. Like the root object, a container object is designed to maintain a list of keys and associate an object with each one. A container can either support strings as its keys, like the root object itself, or can use integers as an alternate key system instead.

By placing containers inside of containers, a hierarchy of objects can be formed. This ability makes the ZODB very popular for programmers implementing content management systems! Just call your ZODB object containers "Folders" when displaying them for public consumption, and users will happily navigate your object hierarchy as though it were a familiar filesystem. The Plone content-management system makes particularly heavy use of this concept.

Now for the other big difference between the ZODB and a relational database: ZODB objects completely lack any schema specification. An object in a ZODB database is allowed to have any collection of instance attributes. They can be placed upon the object, and later deleted, at will. Furthermore, no type restrictions are imposed, beyond the basic restriction that the ZODB only supports attribute values that are composed of combinations of basic pickle-able Python types like numbers, strings, tuples, lists, and dictionaries.

This behavior fits very well with Python itself, though it contrasts very strongly with the practice in statically-typed languages of specifying the list of attributes that objects of that class can possess when one first designs an object and writes its definition. The attributes of a Python object behave like an associative array to which new keys can be added, and old ones deleted, during runtime.

When a programmer writes something like: `myhost.hostname = 'aix.example.com'`, the programmer is not storing the string in some kind of pre-defined and constrained slot named "hostname" that is an inherent attribute of the "myhost" object. Instead, they are either creating or overwriting the value associated with a "hostname" attribute that only came into existence when it was first assigned to, and which will disappear off of the object if you perform a simple: `del myhost.hostname`.

Of course, seasoned Python programmers are wise and systematic, and generally do not use this freedom to create and delete attributes arbitrarily. Typically, they will use the __init__() initialization method of a class to assign an initial or default value to every instance attribute they ever plan on using. This means that subsequent code in the object's methods can assume the presence of each attribute, and therefore simply use, for example, "self.hostname" without having to worry about whether it has been set yet.

When Python programmers do not go ahead and create a particular object attributes at instantiation, they have to either check for the presence of the attributes with a function like hasattr() before attempting to access it, or use a try...except clause to catch the AttributeError exception that is thrown if they try to access it when it's not there.

But despite all of that, there is significant difference between the casual Python discipline of creating all instance attributes at instantiation, and the practice of creating the sort of type signature available in a statically-typed language. Therefore, Python programmers who want to use a relational database generally find they have imposed upon themselves the extra and unnecessary discipline of providing a schema for each of their types.

Python programmers are not always averse to schemas, which are often crafted in order to drive form generation and constraint checking in a Web framework, or to define interfaces in a dynamic component framework. But these are very specific situations, where a class faces outwards towards other components, or towards users. When using classes that are only visible internally, it can appear quite onerous to many Python programmers to have to provide data definitions simply in order to get persistence working in a relational database. And, of course, once they have done so, they then find that adding a new attribute to their class -- normally a very inexpensive operation -- becomes an exercise in the occasionally arcane world of relational database schema modification.

For all of these reasons, relational databases offer what we might call a noticeable impedance mismatch when we try to use them to persist objects in a dynamic language like Python.

How to use ZODB

To show how to use the ZODB, and to illustrate its properties, we will demonstrate three basic operations: First, we will show how to set up and tear down a connection to the ZODB. Second, we will store and then delete some simple values and data structures directly under keys at the database root. And third, we will persist some actual Python objects and show that their attributes get automatically stored.

We will have to leave for another time the details of how to create container objects with which full hierarchies of objects can be persisted.

Connecting and disconnecting from the ZODB

The standard way of connecting to the ZODB involves creating a series of four objects: a method of storing database data, a "db" wrapper around the storage that gives it actual database behaviors, a "connection" object that starts a particular conversation with that database, and finally a "dbroot" object that gives us access to the base of the object hierarchy contained in the database. All of the following examples will need this same code fragment placed above them in a Python file in order to correctly open and close a ZODB instance:

Using ZODB

```
Â Â Â # myzodb.py

from ZODB import FileStorage, DB
import transaction

class MyZODB(object):
  def __init__(self, path):
    self.storage = FileStorage.FileStorage(path)
    self.db = DB(self.storage)
    self.connection = self.db.open()
    self.dbroot = self.connection.root()

def close(self):
  self.connection.close()
  self.db.close()
  self.storage.close()
```

Note that the "transaction" module was not used in the code fragment above; we imported it because the examples below will be using it.

Storing simple Python data

A Zope database can store all sorts of Python objects. The following script stores several values:

Storing simple Python data

```
# store_simple.py - place some simple data in a ZODB

from myzodb import MyZODB, transaction
db = MyZODB('./Data.fs')
dbroot = db.dbroot
dbroot['a_number'] = 3
dbroot['a_string'] = 'Gift'
dbroot['a_list'] = [1, 2, 3, 5, 7, 12]
dbroot['a_dictionary'] = { 1918: 'Red Sox', 1919: 'Reds' }
dbroot['deeply_nested'] = {
  1918: [ ('Red Sox', 4), ('Cubs', 2) ],
  1919: [ ('Reds', 5), ('White Sox', 3) ],
  }
transaction.commit()
db.close()
```

And then this script will re-open the database and demonstrate that all of the values were stored intact:

Fetching simple data

```
# fetch_simple.py - show what's in the database

from myzodb import MyZODB
db = MyZODB('./Data.fs')
dbroot = db.dbroot
for key in dbroot.keys():
  print key + ':', dbroot[key]
db.close()
```

Note. Our use of the filename "Data.fs" is purely conventional, because many ZODB installations have in fact standardized on the use of that particular file name; but you can use any name you want. When we refer in the rest of this article to your "Data.fs" file, we really mean "whatever file in which you have placed your Zope database."

The ZODB will always be able to see when you set a key to a new value. So a change to the above database like the following will get automatically detected and persisted:

```
dbroot['a_string'] = 'Something Else' transaction.commit() db.close()
```

You need to explicitly tell ZODB about changes to lists or dictionaries because the ZODB cannot see the change being made. This is a define feature of mutability and participation in the persistence framework. The following code will not cause a change that a subsequent run of "fetch_simple.py" will see:

```
        # broken code!
        a_dictionary = dbroot['a_dictionary']
        a_dictionary[1920] = 'Indians'
        transaction.commit()
        db.close()
```

If you are going to be modifying -- rather than completely replacing -- a complex object like this, you need to set the attribute _p_changed on the database root to alert it that it needs to re-store the attributes beneath it:

```
        a_dictionary = dbroot['a_dictionary']
        a_dictionary[1920] = 'Indians'
        dbroot._p_changed = 1
        transaction.commit()
        db.close()
```

If you then re-run "fetch_simple.py", you will see that the change was persisted correctly.

Removing an object is as simple as:

```
        del dbroot['a_number']
        transaction.commit()
        db.close()
```

Note that nothing will happen to the database in any of the above examples if you neglect to call transaction.commit()! Just as in a relational database, only by committing the operations you have been performing do you make them atomically appear in the database.

Persisting objects

Of course, few Python programmers want to work with increasingly complex forests of data structures like the lists, tuples, and dictionaries above. Instead, they want to create full-fledged Python objects whose attributes are then persisted for them. Let us create a small Python file that defines a type we can persist to the database.

To do this, the class will have to inherit from "Persistent." (Note that because Python allows multi-inheritance, the requirement that the class inherits from "Persistent" in no way prevents your own database-ready classes from inheriting from other base classes, as well.)

A model

```
        # mymodel.py - a tiny object model

        from persistent import Persistent

        class Host(Persistent):
          def __init__(self, hostname, ip, interfaces):
            self.hostname = hostname
            self.ip = ip
            self.interfaces = interfaces
```

We can now create several instances of this class and persist them in the ZODB, just like we persisted the simple data structures above:

Storing objects

```
        # store_hosts.py

        from myzodb import MyZODB, transaction
        db = MyZODB('./Data.fs')
        dbroot = db.dbroot

        from mymodel import Host
        host1 = Host('www.example.com', '192.168.7.2', ['eth0', 'eth1'])
        dbroot['www.example.com'] = host1
        host2 = Host('dns.example.com', '192.168.7.4', ['eth0', 'gige0'])
        dbroot['dns.example.com'] = host2

        transaction.commit()
        db.close()
```

The following script will re-open the database and demonstrate that all of the host objects were successfully persisted (by checking the type of each item fetched, it will ignore all of the objects that might still be left in the ZODB from when we ran our first example):

Fetching objects

```
        # fetch_hosts.py - show hosts in the database

        from myzodb import MyZODB
        db = MyZODB('./Data.fs')
        dbroot = db.dbroot

        from mymodel import Host
        for key in dbroot.keys():
          obj = dbroot[key]
          if isinstance(obj, Host):
            print "Host:", obj.name
            print "  IP address:", obj.ip, "  Interfaces:", obj.interfaces

        db.close()
```

Just as the "dbroot" object can detect when place new values at its key indexes, a Persistent object will automatically detect when you set its attributes and save them to the database. So the following code changes the IP address of our first host:

```
        host = dbroot['www.example.com']
        host.ip = '192.168.7.141'
        transaction.commit()
        db.close()
```

But if you store complex data types beneath an object, then the exact same problem comes into play as occurred with complex data types attached to the database root. The following code does not persist its change to the database because the ZODB cannot see that it has occurred:

```
        # broken code!
        host = dbroot['www.example.com']
        host.interfaces.append('eth2')
        transaction.commit()
        db.close()
```

Instead, you have to set the _p_changed attribute like before; but this time, you have to do it on the object itself, *not* on the database root, because objects act as their own root of the attributes under them:

```
        host = dbroot['www.example.com']
        host.interfaces.append('eth2')
        host._p_changed = 1
        transaction.commit()
        db.close()
```

After running this improved code, you should be able to re-run the "fetch_hosts.py" script above and see that the host has indeed gained an interface.

Routine maintenance

A ZODB database instance is easy to maintain. Since it contains no schema that would require design or modification, the only regular maintenance to perform is a periodic packing to prevent it from consuming your entire disk.

Database administrators are already accustomed to the behavior of modern relational databases, whose table files will typically grow on disk without bound unless a SQL "VACUUM" command is performed periodically. For much the same reason -- that is, in order to support the rollback of transactions -- each new change written to a ZODB database is actually appended to the "Data.fs" file, rather than updating information already inside of it. To remove the old information that accumulates as transactions commit, a ZODB administrator must occasionally "pack" its database.

While many relational databases these days offer an auto-vacuuming facility that periodically runs "VACUUM" on each table -- perhaps at set intervals, or maybe after a certain amount of new data has been written -- the ZODB does not currently offer such a facility. Instead, a ZODB administrator will typically establish a cron job to perform the operation regularly. A daily packing is often perfectly appropriate for sites not pushing enormous volumes of new information through their ZODB every hour.

There are two ways of performing a packing. If you are running simple scripts that open "Data.fs" directly, like those demonstrated above, then you will need to create a small script that opens "Data.fs" and then runs the "pack" command on the resulting database object:

```
        db = DB(storage)
        db.pack()
```

Because no two programs can have a "Data.fs" file open at the same time, you cannot run your packing script while any of your other scripts are running.

If, instead of having a script open your "Data.fs" directly, you are using a ZEO server (described in the next section), then packing will be simpler -- and will also not require that your clients disconnect! Simply use the "zeopack" command line tool that comes with the ZODB:

```
        $ zeopack -h localhost -p 8080
```

This will connect to your ZEO server and issue a special order that will induce the server to pack your database. It is usually best to perform this when the load on your database is light; many sites run the command daily, early in the morning before the start of business.

Providing remote access with ZEO

While all of the above example scripts simply opened a local "Data.fs" file directly, most production installations of a ZODB run its database as a server instead. The ZODB server product is named "Zope Enterprise Objects" (ZEO), and it comes packaged with the ZODB code itself. Because only one program can safely have a "Data.fs" file opened at a time, a ZEO server is the only way to support connections from several clients. This is especially critical when a database is supporting several front-end servers that are arranged in a load-balancing configuration.

Even many installations that have only a single database client choose to use a ZEO server anyway, since -- as described in the previous section -- this allows the database to be packed without the client having to disconnect (or perform the pack itself).

While you should read the ZEO documentation, which can be found in the source code for ZODB in doc/ZEO/howto.txt, for more details, one generally creates a configuration file for ZEO that looks something like this:

```
         >zeo]
           address zeo.example.com:8090
           monitor-address zeo.example.com:8091
           </zeo>

         <filestorage 1>
           path /var/tmp/Data.fs
           </filestorage>

         <eventlog>
         <logfile>
             path /var/tmp/zeo.log
             format %(asctime)s %(message)s
             </logfile>
             </eventlog>
```

Once you have written this configuration file, running a zeo instance is as easy as:

```
        $ zeoctl ... start
```

Just like a standard UNIX® "init.d" script, the "zeoctl" command will also accept subcommands like "status" and "stop."

Different ZEO clients have different ways of specifying which ZEO server you want them to talk to. Zope instances, for example, have "zope.conf" files with stanzas that look something like this:

```
        <zodb>
          <filestorage>
            path /srv/myapp/var/Data.fs
          </filestorage>
        </zodb>
```

If you need to connect from one of your own programs, then you could replace the "MyZODB" example class given above with something that connects to ZEO instead:

Zeo client

```
      from ZEO.ClientStorage import ClientStorage
      from ZODB import DB

      class MyRemoteZODB(object):
        def __init__(self, server, port):
          server_and_port = (server, port)
          self.storage = ClientStorage(server_and_port)
          self.db = DB(self.storage)
          self.connection = self.db.open()
          self.dbroot = self.connection.root()

        def close(self):
          self.connection.close()
          self.db.close()
          self.storage.close()

      mydb = MyRemoteZODB('localhost', 8080)
      dbroot = mydb.dbroot
```

The resulting "dbroot" object can be used exactly like the "dbroot" objects illustrated above, and will perform on the remote ZODB instance exactly the same operations that the above scripts were performing on the local "Data.fs" file.

Solutions for replication and redundancy

Whether replication is available for a database often decides whether it can be deployed as a storage infrastructure in the enterprise. No degree of flexibility or convenience for developers can, for a mission-critical application, outweigh the requirement that a database be able to survive the inevitable failure of hardware and servers without suffering downtime.

The traditional redundant ZODB implementation is the "Zope Replication Services" suite, offered commercially from the Zope corporation. Their clustering system supports read-write master server, which then forwards changes to any number of read-only secondary servers among which reader load can be distributed by any standard virtual server front end. When the master fails, the system falls back to read-only mode until one of the secondary servers is promoted to a master.

An open-source solution named "ZEORaid" has recently emerged with which more adventuresome developers might want to experiment. It uses the same techniques as a RAID file server to allow an entire cluster of ZEO databases to operate redundantly, with all servers participating in both reads and writes. See http://pypi.python.org/pypi/gocept.zeoraid/ for more information.

Conclusion

Although object databases can seem somewhat exotic, they are a useful tool for users of dynamic, object-oriented languages. Their flexible stance with respect to object structure is an excellent fit for the corresponding properties of dynamic objects, and their hierarchical structure is a natural complement to the hierarchical nature of content management systems. With the ZODB, all of these features are delivered in a client/server configuration, and with the addition of one of the available clustering solutions, it can be brought to enterprise-level robustness.

Ultimately though, let's get to the punch line. ZODB is cutting-edge stuff, and it is used in several important projects, including Plone, a Python-based, enterprise-quality Content Management System; and Grok, a next-generation, Python Web Application Framework. If you really want to get your hands dirty with an Object Database, go to the Resource section and go through the Grok Tutorial. You will be able to build a Web application that has persistent objects in a trivial amount of time. This should give you some idea of how much power persistent objects bring to Web application development, and you might ask yourself, why didn't I think of this before?

Downloadable resources

Description	Name	Size
Sample Source Code For This Article	zodb_src.zip	1KB

Related topics

- ZODB Download
- Website for Plone, a Python, ZODB-based Content Management System
- Wikipedia Object Database
- Wikipedia Object Relational impedance mismatch
- Wikipedia Object Database Comparisons