ZODB documentation and articles

Release

Zope Developer Community

Aug 30, 2017

Contents

Because ZODB is an object database:

- no separate language for database operations
- very little impact on your code to make objects persistent
- no database mapper that partially hides the database.

  Using an object-relational mapping **is not** like using an object database.
- almost no seam between code and database.

Check out the *Tutorial*!

ZODB runs on Python 2.7 or Python 3.4 and above. It also runs on PyPy.

CHAPTER 1

Transactions

Make programs easier to reason about.

Transactions are atomic Changes made in a transaction are either saved in their entirety or not at all.

This makes error handling a lot easier. If you have an error, you just abort the current transaction. You don't have to worry about undoing previous database changes.

Transactions provide isolation Transactions allow multiple logical threads (threads or processes) to access databases and the database prevents the threads from making conflicting changes.

This allows you to scale your application across multiple threads, processes or machines without having to use low-level locking primitives.

You still have to deal with concurrency on some level. For timestamp-based systems like ZODB, you may have to retry conflicting transactions. With locking-based systems, you have to deal with possible deadlocks.

Transactions affect multiple objects Most NoSQL databases don't have transactions. Their notions of consistency are much weaker, typically applying to single documents. There can be good reasons to use NoSQL databases for their extreme scalability, but otherwise, think hard about giving up the benefits of transactions.

ZODB transaction support:

- ACID transactions with snapshot isolation
- Distributed transaction support using two-phase commit

  This allows transactions to span multiple ZODB databases and to span ZODB and non-ZODB databases.

CHAPTER 2

Other notable ZODB features

Pluggable layered storage ZODB has a pluggable storage architecture. This allows a variety of storage schemes including memory-based, file-based and distributed (client-server) storage. Through storage layering, storage components provide compression, encryption, replication and more.

Database caching with invalidation Every database connection has a cache that is a consistent partial database replica. When accessing database objects, data already in the cache is accessed without any database interactions. When data are modified, invalidations are sent to clients causing cached objects to be invalidated. The next time invalidated objects are accessed they'll be loaded from the database.

This makes caching extremely efficient, but provides some limit to the number of clients. The server has to send an invalidation message to each client for each write.

Easy testing ZODB provides in-memory storage implementations as well as copy-on-write layered "demo storage" implementations that make testing database-related code very easy.

Time travel ZODB storages typically add new records on write and remove old records on "pack" operations. This allows limited time travel, back to the last pack time. This can be very useful for forensic analysis.

Binary large objects, Blobs Many databases have these, but so does ZODB.

In applications, Blobs are files, so they can be treated as files in many ways. This can be especially useful when serving media. If you use AWS, there's a Blob implementation that stores blobs in S3 and caches them on disk.

CHAPTER 3

When should you use ZODB?

You want to focus on your application without writing a lot of database code. Even if find you need to incorporate or switch to another database later, you can use ZODB in the early part of your project to make initial discovery and learning much quicker.

Your application has complex relationships and data structures. In relational databases you have to join tables to model complex data structures and these joins can be tedious and expensive. You can mitigate this to some extent in databases like Postgres by using more powerful data types like arrays and JSON columns, but when relationships extend across rows, you still have to do joins.

In NoSQL databases, you can model complex data structures with documents, but if you have relationships across documents, then you have to do joins and join capabilities in NoSQL databases are typically far less powerful and transactional semantics typically don't cross documents, if they exist at all.

In ZODB, you can make objects as complex as you want and cross object relationships are handled with Python object references.

You access data through object attributes and methods. If your primary object access is search, then other database technologies might be a better fit.

ZODB has no query language other than Python. It's primary support for search is through mapping objects called BTrees. People have build higher-level search APIs on top of ZODB. These work well enough to support some search.

You read data a lot more than you write it. ZODB caches aggressively, and if you're working set fits (or mostly fits) in memory, performance is very good because it rarely has to touch the database server.

If your application is very write heavy (e.g. logging), then you're better off using something else. Sometimes, you can use a database suitable for heavy writes in combination with ZODB.

Need to test logic that uses your database. ZODB has a number of storage implementations, including layered in-memory implementations that make testing very easy.

A database without an in-memory storage option can make testing very complicated.

CHAPTER 4

When should you *not* use ZODB?

- Search is a dominant data access path
- You have high write volume
- Caching is unlikely to benefit you

  This can be the case when write volume is high, or when you tend to access small amounts of data from a working set way too large to fit in memory and when there's no good mechanism for dividing the working set across application servers.

- You need to use non-Python tools to access your database.

  especially tools designed to work with relational databases

CHAPTER 5

How does ZODB scale?

Not as well as many technologies, but some fairly large applications have been built on ZODB.

At Zope Corporation, several hundred newspaper content-management systems and web sites were hosted using a multi-database configuration with most data in a main database and a catalog database. The databases had several hundred gigabytes of ordinary database records plus multiple terabytes of blob data.

CHAPTER 6

ZODB is mature

ZODB is very mature. Development started in 1996 and it has been used in production in thousands of applications for many years.

ZODB is in heavy use in the Pyramid and Plone communities and in many other applications.

CHAPTER 7

Learning more

Tutorial

This tutorial is intended to guide developers with a step-by-step introduction of how to develop an application which stores its data in the ZODB.

Introduction

To save application data in ZODB, you'll generally define classes that subclass `persistent.Persistent`:

```
# account.py

import persistent

class Account(persistent.Persistent):

    def __init__(self):
        self.balance = 0.0

    def deposit(self, amount):
        self.balance += amount

    def cash(self, amount):
        assert amount < self.balance
        self.balance -= amount
```

This code defines a simple class that holds the balance of a bank account and provides two methods to manipulate the balance: deposit and cash.

Subclassing `Persistent` provides a number of features:

- The database will automatically track object changes made by setting attributes[1].

[1] You can manually mark an object as changed by setting its `_p_changed__` attribute to `True`. You might do this if you update a subobject, such as a standard Python `list` or `set`, that doesn't subclass `Persistent`.

- Data will be saved in its own database record.

  You can save data that doesn't subclass `Persistent`, but it will be stored in the database record of whatever persistent object references it.

- Objects will have unique persistent identity.

  Multiple objects can refer to the same persistent object and they'll continue to refer to the same object even after being saved and loaded from the database.

  Non-persistent objects are essentially owned by their containing persistent object and if multiple persistent objects refer to the same non-persistent subobject, they'll (eventually) get their own copies.

Note that we put the class in a named module. Classes aren't stored in the ZODB[2]. They exist on the file system and their names, consisting of their class and module names, are stored in the database. It's sometimes tempting to create persistent classes in scripts or in interactive sessions, but if you do, then their module name will be `'__main__'` and you'll always have to define them that way.

Installation

Before being able to use ZODB we have to install it. A common way to do this is with pip:

```
$ pip install ZODB
```

Creating Databases

When a program wants to use the ZODB it has to establish a connection, like any other database. For the ZODB we need 3 different parts: a storage, a database and finally a connection:

```
import ZODB, ZODB.FileStorage

storage = ZODB.FileStorage.FileStorage('mydata.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root
```

ZODB has a pluggable storage framework. This means there are a variety of storage implementations to meet different needs, from in-memory databases, to databases stored in local files, to databases on remote database servers, and specialized databases for compression, encryption, and so on. In the example above, we created a database that stores its data in a local file, using the `FileStorage` class.

Having a storage, we then use it to instantiate a database, which we then connect to by calling `open()`. A process with multiple threads will often have multiple connections to the same database, with different threads having different connections.

There are a number of convenient shortcuts you can use for some of the commonly used storages:

- You can pass a file name to the `DB` constructor to have it construct a FileStorage for you:

  ```
  db = ZODB.DB('mydata.fs')
  ```

  You can pass None to create an in-memory database:

  ```
  memory_db = ZODB.DB(None)
  ```

[2] Actually, there is semi-experimental support for storing classes in the database, but applications rarely do this.

- If you're only going to use one connection, you can call the `connection` function:

```
connection = ZODB.connection('mydata.fs')
memory_connection = ZODB.connection(None)
```

Storing objects

To store an object in the ZODB we simply attach it to any other object that already lives in the database. Hence, the root object functions as a boot-strapping point. The root object is meant to serve as a namespace for top-level objects in your database. We could store account objects directly on the root object:

```
import account

# Probably a bad idea:
root.account1 = account.Account()
```

But if you're going to store many objects, you'll want to use a collection object[3]:

```
import account, BTrees.OOBTree

root.accounts = BTrees.OOBTree.BTree()
root.accounts['account-1'] = Account()
```

Another common practice is to store a persistent object in the root of the database that provides an application-specific root:

```
root.accounts = AccountManagementApplication()
```

That can facilitate encapsulation of an application that shares a database with other applications. This is a little bit like using modules to avoid namespace colisions in Python programs.

Containers and search

BTrees provide the core scalable containers and indexing facility for ZODB. There are different families of BTrees. The most general are OOBTrees, which have object keys and values. There are specialized BTrees that support integer keys and values. Integers can be stored more efficiently, and compared more quickly than objects and they're often used as application-level object identifiers. It's critical, when using BTrees, to make sure that its keys have a stable ordering.

ZODB doesn't provide a query engine. The primary way to access objects in ZODB is by traversing (accessing attributes or items, or calling methods) other objects. Object traversal is typically much faster than search.

You can use BTrees to build indexes for efficient search, when necessary. If your application is search centric, or if you prefer to approach data access that way, then ZODB might not be the best technology for you. Before you turn your back on the ZODB, it may be worth checking out the up-and-coming Newt DB[6] project, which combines the ZODB with Postgresql for indexing, search and access from non-Python applications.

[3] The root object is a fairy simple persistent object that's stored in a single database record. If you stored many objects in it, its database record would become very large, causing updates to be inefficient and causing memory to be used ineffeciently.

Another reason not to store items directly in the root object is that doing so would make adding a second collection of objects later awkward.

[6] Here is an overview of the Newt DB architecture: http://www.newtdb.org/en/latest/how-it-works.html

Transactions

You now have objects in your root object and in your database. However, they are not permanently stored yet. The ZODB uses transactions and to make your changes permanent, you have to commit the transaction:

```
import transaction

transaction.commit()
```

Now you can stop and start your application and look at the root object again, and you will find the data you saved.

If your application makes changes during a transaction and finds that it does not want to commit those changes, then you can abort the transaction and have the changes rolled back[4] for you:

```
transaction.abort()
```

Transactions are a very powerful way to protect the integrity of a database. Transactions have the property that all of the changes made in a transaction are saved, or none of them are. If in the midst of a program, there's an error after making changes, you can simply abort the transaction (or not commit it) and all of the intermediate changes you make are automatically discarded.

Memory Management

ZODB manages moving objects in and out of memory for you. The unit of storage is the persistent object. When you access attributes of a persistent object, they are loaded from the database automatically, if necessary. If too many objects are in memory, then objects used least recently are evicted[5]. The maximum number of objects or bytes in memory is configurable.

Summary

You have seen how to install ZODB and how to open a database in your application and to start storing objects in it. We also touched the two simple transaction commands: `commit` and `abort`. The reference documentation contains sections with more information on the individual topics.

ZODB programming guide

This guide consists of a collection of topics that should be of interest to most developers. They're provided in order of importance, which is also an order from least to most advanced, but they can be read in any order.

If you haven't yet, you should read the *Tutorial*.

Installing and running ZODB

This topic discusses some boring nitty-gritty details needed to actually run ZODB.

[4] A caveat is that ZODB can only roll back changes to objects that have been stored and committed to the database. Objects not previously committed can't be rolled back because there's no previous state to roll back to.

[5] Objects aren't actually evicted, but their state is released, so they take up much less memory and any objects they referenced can be removed from memory.

Installation

Installation of ZODB is pretty straightforward using Python's packaging system. For example, using pip:

```
pip install ZODB
```

You may need additional optional packages, such as ZEO or RelStorage, depending your deployment choices.

Configuration

You can set up ZODB in your application using either Python, or ZODB's configuration language. For simple database setup, and especially for exploration, the Python APIs are sufficient.

For more complex configurations, you'll probably find ZODB's configuration language easier to use.

To understand database setup, it's important to understand ZODB's architecture. ZODB separates database functionality from storage concerns. When you create a database object, you specify a storage object for it to use, as in:

```
import ZODB, ZODB.FileStorage

storage = ZODB.FileStorage.FileStorage('mydata.fs')
db = ZODB.DB(storage)
```

So when you define a database, you'll also define a storage. In the example above, we define a *file storage* and then use it to define a database.

Sometimes, storages are created through composition. For example, if we want to save space, we could layer a `ZlibStorage`[1] over the file storage:

```
import ZODB, ZODB.FileStorage, zc.zlibstorage

storage = ZODB.FileStorage.FileStorage('mydata.fs')
compressed_storage = zc.zlibstorage.ZlibStorage(storage)
db = ZODB.DB(compressed_storage)
```

ZlibStorage compresses database records[2].

Python configuration

To set up a database with Python, you'll construct a storage using the *storage APIs*, and then pass the storage to the *DB* class to create a database, as shown in the examples in the previous section.

The *DB* class also accepts a string path name as its storage argument to automatically create a file storage. You can also pass `None` as the storage to automatically use a *MappingStorage*, which is convenient when exploring ZODB:

```
db = ZODB.DB(None) # Create an in-memory database.
```

Text configuration

ZODB supports a text-based configuration language. It uses a syntax similar to Apache configuration files. The syntax was chosen to be familiar to site administrators.

[1] zc.zlibstorage is an optional package that you need to install separately.
[2] ZlibStorage uses the `zlib` standard module, which uses the zlib library.

ZODB's text configuration uses ZConfig. You can use ZConfig to create your application's configuration, but it's more common to include ZODB configuration strings in their own files or embedded in simpler configuration files, such as configarser files.

A database configuration string has a `zodb` section wrapping a storage section, as in:

```
<zodb>
  cache-size-bytes 100MB
  <mappingstorage>
  </mappingstorage>
</zodb>
```

In the example above, the *mappingstorage* section defines the storage used by the database.

To create a database from a string, use *ZODB.config.databaseFromString()*:

```
>>> import ZODB.config
>>> db = ZODB.config.databaseFromString(snippet)
```

To load databases from file names or URLs, use *ZODB.config.databaseFromURL()*.

URI-based configuration

Another database configuration option is provided by the zodburi package. See: http://docs.pylonsproject.org/projects/zodburi. It's less powerful than the Python or text configuration options, but allows configuration to be reduced to a single URI and handles most cases.

Using databases: connections

Once you have a database, you need to get a database connection to do much of anything. Connections take care of loading and saving objects and manage object caches. Each connection has its own cache[3].

Getting connections

Amongst[4] the common ways of getting a connection:

db.open() The database *open()* method opens a connection, returning a connection object:

```
>>> conn = db.open()
```

It's up to the application to call *close()* when the application is done using the connection.

If changes are made, the application *commits transactions* to make them permanent.

db.transaction() The database *transaction()* method returns a context manager that can be used with the python with statement to execute a block of code in a transaction:

```
with db.transaction() as connection:
    connection.root.foo = 1
```

[3] ZODB can be very efficient at caching data in memory, especially if your working set is small enough to fit in memory, because the cache is simply an object tree and accessing a cached object typically requires no database interaction. Because each connection has its own cache, connections can be expensive, depending on their cache sizes. For this reason, you'll generally want to limit the number of open connections you have at any one time. Connections are pooled, so opening a connection is inexpensive.

[4] https://www.youtube.com/watch?v=7WJXHY2OXGE

In the example above, we used `as connection` to get the database connection used in the variable `connection`.

some_object._p_jar For code that's already running in the context of an open connection, you can get the current connection as the `_p_jar` attribute of some persistent object that was accessed via the connection.

Getting objects

Once you have a connection, you access objects by traversing the object graph from the root object.

The database root object is a mapping object that holds the top level objects in the database. There should only be a small number of top-level objects (often only one). You can get the root object by calling a connection's `root` attribute:

```
>>> root = conn.root()
>>> root
{'foo': 1}
>>> root['foo']
1
```

For convenience[5], you can also get top-level objects by accessing attributes of the connection root object:

```
>>> conn.root.foo
1
```

Once you have a top-level object, you use its methods, attributes, or operations to access other objects and so on to get the objects you need. Often indexing data structures like BTrees are used to make it possible to search objects in large collections.

Writing persistent objects

In the *Tutorial*, we discussed the basics of implementing persistent objects by subclassing `persistent.Persistent`. This is probably enough for 80% of persistent-object classes you write, but there are some other aspects of writing persistent classes you should be aware of.

Access and modification

Two of the main jobs of the `Persistent` base class are to detect when an object has been accessed and when it has been modified. When an object is accessed, its state may need to be loaded from the database. When an object is modified, the modification needs to be saved if a transaction is committed.

`Persistent` detects object accesses by hooking into object attribute access and update. In the case of object update, there may be other ways of modifying state that we need to make provision for.

Rules of persistence

When implementing persistent objects, be aware that an object's attributes should be :

- immutable (such as strings or integers),
- persistent (subclass Persistent), or

[5] The ability to access top-level objects of the database as root attributes is a recent convenience. Originally, the `root()` method was used to access the root object which was then accessed as a mapping. It's still potentially useful to access top-level objects using the mapping interface if their names aren't valid attribute names.

- You need to take special precautions.

If you modify a non-persistent mutable value of a persistent-object attribute, you need to mark the persistent object as changed yourself by setting `_p_changed` to True:

```
import persistent

class Book(persistent.Persistent):

    def __init__(self, title):
        self.title = title
        self.authors = []

    def add_author(self, author):
        self.authors.append(author)
        self._p_changed = True
```

In this example, `Book` objects have an `authors` object that's a regular Python list, so it's mutable and non-persistent. When we add an author, we append it to the `authors` attribute's value. Because we didn't set an attribute on the book, it's not marked as changed, so we set `_p_changed` ourselves.

Using standard Python lists, dicts, or sets is a common thing to do, so this pattern of setting `_p_changed` is common.

Let's look at some alternatives.

Using tuples for small sequences instead of lists

If objects contain sequences that are small or that don't change often, you can use tuples instead of lists:

```
import persistent

class Book(persistent.Persistent):

    def __init__(self, title):
        self.title = title
        self.authors = ()

    def add_author(self, author):
        self.authors += (author, )
```

Because tuples are immutable, they satisfy the rules of persistence without any special handling.

Using persistent data structures

The `persistent` package provides persistent versions of `list` and `dict`, namely `persistent.list.PersistentList` and `persistent.mapping.PersistentMapping`. We can update our example to use `PersistentList`:

```
import persistent
import persistent.list

class Book(persistent.Persistent):

    def __init__(self, title):
        self.title = title
        self.authors = persistent.list.PersistentList()
```

```
    def add_author(self, author):
        self.authors.append(author)
```

Note that in this example, when we added an author, the book itself didn't change, but the `authors` attribute value did. Because `authors` is a persistent object, it's stored in a separate database record from the book record and is managed by ZODB independent of the management of the book.

In addition to `PersistentList` and `PersistentMapping`, general persistent data structures are provided by the BTrees package, most notably `BTree` and `TreeSet` objects. Unlike `PersistentList` and `PersistentMapping`, `BTree` and `TreeSet` objects are scalable and can easily hold millions of objects, because their data are spread over many subobjects.

It's generally better to use `BTree` objects than `PersistentMapping` objects, because they're scalable and because they handle *conflicts* better. `TreeSet` objects are the only ZODB-provided persistent set implementation. `BTree` and `TreeSets` come in a number of families provided via different modules and differ in their internal implementations:

Module	Key type	Value Type
BTrees.OOBTree	object	object
BTrees.IOBTree	integer	Object
BTrees.OIBTree	object	integer
BTrees.IIBTree	integer	integer
BTrees.IFBTree	integer	float
BTrees.LOBTree	64-bit integer	Object
BTrees.OLBTree	object	64-bit integer
BTrees.LLBTree	64-bit integer	64-bit integer
BTrees.LFBTree	64-bit integer	float

Here's a version of the example that uses a `TreeSet`:

```
import persistent
from BTrees.OOBTree import TreeSet

class Book(persistent.Persistent):

    def __init__(self, title):
        self.title = title
        self.authors = TreeSet()

    def add_author(self, author):
        self.authors.add(author)
```

If you're going to use custom classes as keys in a `BTree` or entries in a `TreeSet`, they must provide a total ordering. The builtin python *str* class is always safe to use as BTree key. You can use zope.keyreference to treat arbitrary persistent objects as totally orderable based on their persistent object identity.

Scalable sequences are a bit more challenging. The zc.blist package provides a scalable list implementation that works well for some sequence use cases.

Properties

If you implement some attributes using Python properties (or other types of descriptors), they are treated just like any other attributes by the persistence machinery. When you set an attribute through a property, the object is considered changed, even if the property didn't actually modify the object state.

Special attributes

There are some attributes that are treated specially.

Attributes with names starting with `_p_` are reserved for use by the persistence machinery and by ZODB. These include (but aren't limited to):

_p_changed The `_p_changed` attribute has the value `None` if the object is a *ghost*, True if it's changed, and False if it's not a ghost and not changed.

_p_oid The object's unique id in the database.

_p_serial The object's revision identifier also know as the object serial number, also known as the object transaction id. It's a timestamp and if not set has the value 0 encoded as string of 8 zero bytes.

_p_jar The database connection the object was accessed through. This is commonly used by database-aware application code to get hold of an object's database connection.

Attributes with names starting with `_v_` are treated as volatile. They aren't saved to the database. They're useful for caching data that can be computed from saved data and shouldn't be saved[1]. They should be treated as though they can disappear between transactions. Setting a volatile attribute doesn't cause an object to be considered to be modified.

An object's `__dict__` attribute is treated specially in that getting it doesn't cause an object's state to be loaded. It may have the value `None` rather than a dictionary for *ghosts*.

Object storage and management

Every persistent object is stored in its own database record. Some storages maintain multiple object revisions, in which case each persistent object is stored in its own set of records. Data for different persistent objects are stored separately.

The database manages each object separately, according to a *life cycle*.

This is important when considering how to distribute data across your objects. If you use lots of small persistent objects, then more objects may need to be loaded or saved and you may incur more memory overhead. On the other hand, if objects are too big, you may load or save more data than would otherwise be needed.

You can't change your mind in subclassing persistent

Currently, you can't change your mind about whether a class is persistent (subclasses `persistent.Persistent`) or not. If you save objects in a database who's classes subclass `persistent.Persistent`, you can't change your mind later and make them non-persistent, and the other way around. This may be a bug or misfeature.

Schema migration

Object requirements and implementations tend to evolve over time. This isn't a problem for objects that are short lived, but persistent objects may have lifetimes that extend for years. There needs to be some way of making sure that state for an older object schema can still be loaded into an object with the new schema.

Adding attributes

Perhaps the commonest schema change is to add attributes. This is usually accomplished easily by adding a default value in a class definition:

[1] The zope.cachedescriptors package provides some descriptors that help implement attributes that cache data.

```
class Book(persistent.Persistent):

    publisher = 'UNKNOWN'

    def __init__(self, title, publisher):
        self.title = title
        self.publisher = publisher
        self.authors = TreeSet()

    def add_author(self, author):
        self.authors.add(author)
```

Removing attributes

Removing attributes generally doesn't require any action, assuming that their presence in older objects doesn't do any harm.

Renaming/moving classes

The easiest way to handle renaming or moving classes is to leave aliases for the old name. For example, if we have a class, `library.Book`, and want to move it to `catalog.Publication`, we can keep a `library` module that contains:

```
from catalog import Publication as Book # XXX deprecated name
```

A downside of this approach is that it clutters code and may even cause us to keep modules solely to hold aliases. (zope.deferredimport can help with this by making these aliases a little more efficient and by generating deprecation warnings.)

Migration scripts

If the simple approaches above aren't enough, then migration scripts can be used. How these scripts are written is usually application dependent, as the application usually determines where objects of a given type reside in the database. (There are also some low-level interfaces for iterating over all of the objects of a database, but these are usually impractical for large databases.)

An improvement to running migration scripts manually is to use a generational framework like zope.generations. With a generational framework, each migration is assigned a migration number and the number is recorded in the database as each migration is run. This is useful because remembering what migrations are needed is automated.

Upgrading multiple clients without down time

Production applications typically have multiple clients for availability and load balancing. This means an active application may be committing transactions using multiple software and schema versions. In this situation, you may need to plan schema migrations in multiple steps:

1. Upgrade software on all clients to a version that works with the old and new version of the schema and that writes data using the old schema.
2. Upgrade software on all clients to a version that works with the old and new version of the schema and that writes data using the new schema.

3. Migrate objects written with the old schema to the new schema.
4. Remove support for the old schema from the software.

Object life cycle states and special attributes (advanced)

Persistent objects typically transition through a collection of states. Most of the time, you don't need to think too much about this.

Unsaved When an object is created, it's said to be in an *unsaved* state until it's associated with a database.

Added When an unsaved object is added to a database, but hasn't been saved by committing a transaction, it's in the *added* state.

Note that most objects are added implicitly by being set as subobjects (attribute values or items) of objects already in the database.

Saved When an object is added and saved through a transaction commit, the object is in the *saved* state.

Changed When a saved object is updated, it enters the *changed* state to indicate that there are changes that need to be committed. It remains in this state until either:

- The current transaction is committed, and the object transitions to the saved state, or
- The current transaction is aborted, and the object transitions to the ghost state.

Ghost An object in the *ghost* state is an empty shell. It has no state. When it's accessed, its state will be loaded automatically, and it will enter the saved state. A saved object can become a ghost if it hasn't been accessed in a while and the database releases its state to make room for other objects. A changed object can also become a ghost if the transaction it's modified in is aborted.

An object that's loaded from the database is loaded as a ghost. This typically happens when the object is a subobject of another object who's state is loaded.

We can interrogate and control an object's state, although somewhat indirectly. To do this, we'll look at some special persistent-object attributes, described in *Special attributes*, above.

Let's look at some state transitions with an example. First, we create an unsaved book:

```
>>> book = Book("ZODB")
>>> from ZODB.utils import z64
>>> book._p_changed, bool(book._p_oid)
(False, False)
```

We can tell that it's unsaved because it doesn't have an object id, `_p_oid`.

If we add it to a database:

```
>>> import ZODB
>>> connection = ZODB.connection(None)
>>> connection.add(book)
>>> book._p_changed, bool(book._p_oid), book._p_serial == z64
(False, True, True)
```

We know it's added because it has an oid, but its serial (object revision timestamp), `_p_serial`, is the special zero value, and it's value for `_p_changed` is False.

If we commit the transaction that added it:

```
>>> import transaction
>>> transaction.commit()
>>> book._p_changed, bool(book._p_oid), book._p_serial == z64
(False, True, False)
```

We see that the object is in the saved state because it has an object id and serial, and is unchanged.

Now if we modify the object, it enters the changed state:

```
>>> book.title = "ZODB Explained"
>>> book._p_changed, bool(book._p_oid), book._p_serial == z64
(True, True, False)
```

If we abort the transaction, the object becomes a ghost:

```
>>> transaction.abort()
>>> book._p_changed, bool(book._p_oid)
(None, True)
```

We can see it's a ghost because `_p_changed` is None. (`_p_serial` isn't meaningful for ghosts.)

If we access the object, it will be loaded into the saved state, which is indicated by a false `_p_changed` and an object id and non-zero serial.

```
>>> book.title
'ZODB'
>>> book._p_changed, bool(book._p_oid), book._p_serial == z64
(False, True, False)
```

Note that accessing `_p_` attributes didn't cause the object's state to be loaded.

We've already seen how modifying `_p_changed` can cause an object to be marked as modified. We can also use it to make an object into a ghost:

```
>>> book._p_changed = None
>>> book._p_changed, bool(book._p_oid)
(None, True)
```

Things you can do, but need to carefully consider (advanced)

While you can do anything with a persistent subclass that you can with a normal subclass, certain things have additional implications for persistent objects. These often show up as performance issues, or the result may become hard to maintain.

Implement `__eq__` and `__hash__`

When you store an entry into a Python `dict` (or the persistent variant `PersistentMapping`, or a `set` or `frozenset`), the key's `__eq__` and `__hash__` methods are used to determine where to store the value. Later they are used to look it up via `in` or `__getitem__`.

When that `dict` is later loaded from the database, the internal storage is rebuilt from scratch. This means that every key has its `__hash__` method called at least once, and may have its `__eq__` method called many times.

By default, every object, including persistent objects, inherits an implementation of `__eq__` and `__hash__` from `object`. These default implementations are based on the object's *identity*, that is, its unique identifier within the current Python process. Calling them, therefore, is very fast, even on *ghosts*, and doesn't cause a ghost to load its state.

If you override __eq__ and __hash__ in a custom persistent subclass, however, when you use instances of that class as a key in a dict, then the instance will have to be unghosted before it can be put in the dictionary. If you're building a large dictionary with many such keys that are ghosts, you may find that loading all the object states takes a considerable amount of time. If you were to store that dictionary in the database and load it later, *all* the keys will have to be unghosted at the same time before the dictionary can be accessed, again, possibly taking a long time.

For example, a class that defines __eq__ and __hash__ like this:

```
class BookEq(persistent.Persistent):

    def __init__(self, title):
        self.title = title
        self.authors = ()

    def add_author(self, author):
        self.authors += (author, )

    def __eq__(self, other):
        return self.title == other.title and self.authors == other.authors

    def __hash__(self):
        return hash((self.title, self.authors))
```

is going to be much slower to use as a key in a persistent dictionary, or in a new dictionary when the key is a ghost, than the class that inherits identity-based __eq__ and __hash__.

There are some alternatives:

- Avoiding the use of persistent objects as keys in dictionaries or entries in sets sidesteps the issue.
- If your application can tolerate identity based comparisons, simply don't implement the two methods. This means that objects will be compared only by identity, but because persistent objects are persistent, the same object will have the same identity in each connection, so that often works out.

  It is safe to remove __eq__ and __hash__ methods from a class even if you already have dictionaries in a database using instances of those classes as keys.
- Make your classes orderable and use them as keys in a BTree or entries in a TreeSet instead of a dictionary or set. Even though your custom comparison methods will have to unghost the objects, the nature of a BTree means that only a small number of objects will have to be loaded in most cases.
- Any persistent object can be wrapped in a zope.keyreferenece to make it orderable and hashable based on persistent identity. This can be an alternative for some dictionaries if you can't alter the class definition but can accept identity comparisons in some dictionaries or sets. You must remember to wrap all keys, though.

Implement __getstate__ and __setstate__

When an object is saved in a database, its __getstate__ method is called without arguments to get the object's state. The default implementation simply returns a copy of an object's instance dictionary. (It's a little more complicated for objects with slots.)

An object's state is loaded by loading the state from the database and passing it to the object's __setstate__ method. The default implementation expects a dictionary, which it uses to populate the object's instance dictionary.

Early on, we thought that overriding these methods would be useful for tasks like providing more efficient state representations or for *schema migration*, but we found that the result was to make object implementations brittle and/or complex and the benefit usually wasn't worth it.

Implement `__getattr__`, `__getattribute__`, or `__setattribute__`

This is something extremely clever people might attempt, but it's probably never worth the bother. It's possible, but it requires such deep understanding of persistence and internals that we're not even going to document it. :)

Links

persistent.Persistent provides additional documentation on the `Persistent` base class.

The zc.blist package provides a scalable sequence implementation for many use cases.

The zope.cachedescriptors package provides descriptor implementations that facilitate implementing caching attributes, especially _v_ volatile attributes.

The zope.deferredimport package provides lazy import and support for deprecating import location, which is helpful when moving classes, especially persistent classes.

The zope.generations package provides a framework for managing schema-migration scripts.

Transactions and concurrency

Contents

Transactions are a core feature of ZODB. Much has been written about transactions, and we won't go into much detail here. Transactions provide two core benefits:

Atomicity When a transaction executes, it succeeds or fails completely. If some data are updated and then an error occurs, causing the transaction to fail, the updates are rolled back automatically. The application using the transactional system doesn't have to undo partial changes. This takes a significant burden from developers and increases the reliability of applications.

Concurrency Transactions provide a way of managing concurrent updates to data. Different programs operate on the data independently, without having to use low-level techniques to moderate their access. Coordination and synchronization happen via transactions.

Using transactions

All activity in ZODB happens in the context of database connections and transactions. Here's a simple example:

```
import ZODB, transaction
db = ZODB.DB(None) # Use a mapping storage
conn = db.open()

conn.root.x = 1
transaction.commit()
```

In the example above, we used `transaction.commit()` to commit a transaction, making the change to `conn.root` permanent. This is the most common way to use ZODB, at least historically.

If we decide we don't want to commit a transaction, we can use `abort`:

```
conn.root.x = 2
transaction.abort() # conn.root.x goes back to 1
```

In this example, because we aborted the transaction, the value of `conn.root.x` was rolled back to 1.

There are a number of things going on here that deserve some explanation. When using transactions, there are three kinds of objects involved:

Transaction Transactions represent units of work. Each transaction has a beginning and an end. Transactions provide the *ITransaction* interface.

Transaction manager Transaction managers create transactions and provide APIs to start and end transactions. The transactions managed are always sequential. There is always exactly one active transaction associated with a transaction manager at any point in time. Transaction managers provide the *ITransactionManager* interface.

Data manager Data managers manage data associated with transactions. ZODB connections are data managers. The details of how they interact with transactions aren't important here.

Explicit transaction managers

ZODB connections have transaction managers associated with them when they're opened. When we call the database *open()* method without an argument, a thread-local transaction manager is used. Each thread has its own transaction manager. When we called `transaction.commit()` above we were calling commit on the thread-local transaction manager.

Because we used a thread-local transaction manager, all of the work in the transaction needs to happen in the same thread. Similarly, only one transaction can be active in a thread.

If we want to run multiple simultaneous transactions in a single thread, or if we want to spread the work of a transaction over multiple threads[5], then we can create transaction managers ourselves and pass them to *open()*:

```
my_transaction_manager = transaction.TransactionManager()
conn = db.open(my_transaction_manager)
conn.root.x = 2
my_transaction_manager.commit()
```

In this example, to commit our work, we called `commit()` on the transaction manager we created and passed to *open()*.

[5] While it's possible to spread transaction work over multiple threads, **it's not a good idea**. See *Concurrency, threads and processes*

Context managers

In the examples above, the transaction beginnings were implicit. Transactions were effectively[6] created when the transaction managers were created and when previous transactions were committed. We can create transactions explicitly using *begin()*:

```
my_transaction_manager.begin()
```

A more modern[7] way to manage transaction boundaries is to use context managers and the Python `with` statement. Transaction managers are context managers, so we can use them with the `with` statement directly:

```
with my_transaction_manager as trans:
    trans.note(u"incrementing x")
    conn.root.x += 1
```

When used as a context manager, a transaction manager explicitly begins a new transaction, executes the code block and commits the transaction if there isn't an error and aborts it if there is an error.

We used `as trans` above to get the transaction.

Databases provide the *transaction()* method to execute a code block as a transaction:

```
with db.transaction() as conn2:
    conn2.root.x += 1
```

This opens a connection, assignes it its own context manager, and executes the nested code in a transaction. We used `as conn2` to get the connection. The transaction boundaries are defined by the `with` statement.

Getting a connection's transaction manager

In the previous example, you may have wondered how one might get the current transaction. Every connection has an associated transaction manager, which is available as the `transaction_manager` attribute. So, for example, if we wanted to set a transaction note:

```
with db.transaction() as conn2:
    conn2.transaction_manager.get().note(u"incrementing x again")
    conn2.root.x += 1
```

Here, we used the *get()* method to get the current transaction.

Connection isolation

In the last few examples, we used a connection opened using *transaction()*. This was distinct from and used a different transaction manager than the original connection. If we looked at the original connection, `conn`, we'd see that it has the same value for `x` that we set earlier:

```
>>> conn.root.x
3
```

This is because it's still in the same transaction that was begun when a change was last committed against it. If we want to see changes, we have to begin a new transaction:

[6] Transactions are implicitly created when needed, such as when data are first modified.
[7] ZODB and the transaction package predate context managers and the Python `with` statement.

```
>>> trans = my_transaction_manager.begin()
>>> conn.root.x
5
```

ZODB uses a timestamp-based commit protocol that provides snapshot isolation. Whenever we look at ZODB data, we see its state as of the time the transaction began.

Conflict errors

As mentioned in the previous section, each connection sees and operates on a view of the database as of the transaction start time. If two connections modify the same object at the same time, one of the connections will get a conflict error when it tries to commit:

```
with db.transaction() as conn2:
   conn2.root.x += 1

conn.root.x = 9
my_transaction_manager.commit() # will raise a conflict error
```

If we executed this code, we'd get a `ConflictError` exception on the last line. After a conflict error is raised, we'd need to abort the transaction, or begin a new one, at which point we'd see the data as written by the other connection:

```
>>> my_transaction_manager.abort()
>>> conn.root.x
6
```

The timestamp-based approach used by ZODB is referred to as an *optimistic* approach, because it works best if there are no conflicts.

The best way to avoid conflicts is to design your application so that multiple connections don't update the same object at the same time. This isn't always easy.

Sometimes you may need to queue some operations that update shared data structures, like indexes, so the updates can be made by a dedicated thread or process, without making simultaneous updates.

Retrying transactions

The most common way to deal with conflict errors is to catch them and retry transactions. To do this manually involves code that looks something like this:

```
max_attempts = 3
attempts = 0
while True:
    try:
        with transaction.manager:
            ... code that updates a database
    except transaction.interfaces.TransientError:
        attempts += 1
        if attempts == max_attempts:
            raise
    else:
        break
```

In the example above, we used `transaction.manager` to refer to the thread-local transaction manager, which we then used used with the `with` statement. When a conflict error occurs, the transaction must be aborted before retrying the update. Using the transaction manager as a context manager in the `with` statement takes care of this for us.

The example above is rather tedious. There are a number of tools to automate transaction retry. The transaction package provides a context-manager-based mechanism for retrying transactions:

```
for attempt in transaction.manager.attempts():
    with attempt:
        ... code that updates a database
```

Which is shorter and simpler[1].

For Python web frameworks, there are WSGI[2] middle-ware components, such as repoze.tm2 that align transaction boundaries with HTTP requests and retry transactions when there are transient errors.

For applications like queue workers or cron jobs, conflicts can sometimes be allowed to fail, letting other queue workers or subsequent cron-job runs retry the work.

Conflict resolution

ZODB provides a conflict-resolution framework for merging conflicting changes. When conflicts occur, conflict resolution is used, when possible, to resolve the conflicts without raising a ConflictError to the application.

Commonly used objects that implement conflict resolution are buckets and `Length` objects provided by the BTree package.

The main data structures provided by BTrees, BTrees and TreeSets, spread their data over multiple objects. The leaf-level objects, called *buckets*, allow distinct keys to be updated without causing conflicts[3].

`Length` objects are conflict-free counters that merge changes by simply accumulating changes.

> **Caution:** Conflict resolution weakens consistency. Resist the temptation to try to implement conflict resolution yourself. In the future, ZODB will provide greater control over conflict resolution, including the option of disabling it.
>
> It's generally best to avoid conflicts in the first place, if possible.

ZODB and atomicity

ZODB provides atomic transactions. When using ZODB, it's important to align work with transactions. Once a transaction is committed, it can't be rolled back[4] automatically. For applications, this implies that work that should be atomic shouldn't be split over multiple transactions. This may seem somewhat obvious, but the rule can be broken in non-obvious ways. For example a Web API that splits logical operations over multiple web requests, as is often done in REST APIs, violates this rule.

[1] But also a bit obscure. The Python context-manager mechanism isn't a great fit for the transaction-retry use case.
[2] Web Server Gateway Interface
[3] Conflicts can still occur when buckets split due to added objects causing them to exceed their maximum size.
[4] Transactions can't be rolled back, but they may be undone in some cases, especially if subsequent transactions haven't modified the same objects.

Partial transaction error recovery using savepoints

A transaction can be split into multiple steps that can be rolled back individually. This is done by creating savepoints. Changes in a savepoint can be rolled back without rolling back an entire transaction:

```
import ZODB
db = ZODB.DB(None) # using a mapping storage
with db.transaction() as conn:
    conn.root.x = 1
    conn.root.y = 0
    savepoint = conn.transaction_manager.savepoint()
    conn.root.y = 2
    savepoint.rollback()

with db.transaction() as conn:
    print([conn.root.x, conn.root.y]) # prints 1 0
```

If we executed this code, it would print 1 and 0, because while the initial changes were committed, the changes in the savepoint were rolled back.

A secondary benefit of savepoints is that they save any changes made before the savepoint to a file, so that memory of changed objects can be freed if they aren't used later in the transaction.

Concurrency, threads and processes

ZODB supports concurrency through transactions. Multiple programs[8] can operate independently in separate transactions. They synchronize at transaction boundaries.

The most common way to run ZODB is with each program running in its own thread. Usually the thread-local transaction manager is used.

You can use multiple threads per transaction and you can run multiple transactions in a single thread. To do this, you need to instantiate and use your own transaction manager, as described in *Explicit transaction managers*. To run multiple transaction managers simultaneously in a thread, you need to use a separate transaction manager for each transaction.

To spread a transaction over multiple threads, you need to keep in mind that database connections, transaction managers and transactions are **not thread-safe**. You have to prevent simultaneous access from multiple threads. For this reason, **using multiple threads with a single transaction is not recommended**, but it is possible with care.

Using multiple processes

Using multiple Python processes is a good way to scale an application horizontally, especially given Python's global interpreter lock.

Some things to keep in mind when utilizing multiple processes:

- If using the `multiprocessing` module, you can't[9] share databases or connections between processes. When you launch a subprocess, you'll need to re-instantiate your storage and database.
- You'll need to use a storage such as ZEO, RelStorage, or NEO, that supports multiple processes. None of the included storages do.

[8] We're using *program* here in a fairly general sense, meaning some logic that we want to run to perform some function, as opposed to an operating system program.

[9] at least not now.

Reference Documentation

ZODB APIs

Contents

ZODB module functions

DB(*storage*, **args*, ***kw*)
Create a database. See `ZODB.DB`.

ZODB.**connection**(**args*, ***kw*)
Create a database `connection`.

A database is created using the given arguments and opened to create the returned connection. The database will be closed when the connection is closed. This is a convenience function to avoid managing a separate database object.

Databases

class ZODB.**DB**(*storage*, *pool_size=7*, *pool_timeout=2147483648*, *cache_size=400*, *cache_size_bytes=0*, *historical_pool_size=3*, *historical_cache_size=1000*, *historical_cache_size_bytes=0*, *historical_timeout=300*, *database_name='unnamed'*, *databases=None*, *xrefs=True*, *large_record_size=16777216*, ***storage_args*)
The Object Database

The DB class coordinates the activities of multiple database Connection instances. Most of the work is done by the Connections created via the open method.

The DB instance manages a pool of connections. If a connection is closed, it is returned to the pool and its object cache is preserved. A subsequent call to open() will reuse the connection. There is no hard limit on the pool size. If more than *pool_size* connections are opened, a warning is logged, and if more than twice that many, a critical problem is logged.

The database provides a few methods intended for application code – open, close, undo, and pack – and a large collection of methods for inspecting the database and its connections' caches.

__init__(*storage*, *pool_size=7*, *pool_timeout=2147483648*, *cache_size=400*, *cache_size_bytes=0*, *historical_pool_size=3*, *historical_cache_size=1000*, *historical_cache_size_bytes=0*, *historical_timeout=300*, *database_name='unnamed'*, *databases=None*, *xrefs=True*, *large_record_size=16777216*, ***storage_args*)
Create an object database.

Parameters

- **storage** – the storage used by the database, such as a *FileStorage*. This can be a string path name to use a constructed *FileStorage* storage or None to use a constructed *MappingStorage*.
- **pool_size** (*int*) – expected maximum number of open connections. Warnings are logged when this is exceeded and critical messages are logged if twice the pool size is exceeded.
- **pool_timeout** (*seconds*) – Maximum age of inactive connections When a connection has remained unused in a connection pool for more than pool_timeout seconds, it will be discarded and it's resources released.
- **cache_size** (*objects*) – target maximum number of non-ghost objects in each connection object cache.
- **cache_size_bytes** (*int*) – target total memory usage of non-ghost objects in each connection object cache.
- **historical_pool_size** (*int*) – expected maximum number of total historical connections
- **historical_cache_size** (*objects*) – target maximum number of non-ghost objects in each historical connection object cache.
- **historical_cache_size_bytes** (*int*) – target total memory usage of non-ghost objects in each historical connection object cache.
- **historical_timeout** (*seconds*) – Maximum age of inactive historical connections. When a connection has remained unused in a historical connection pool for more than pool_timeout seconds, it will be discarded and it's resources released.
- **database_name** (*str*) – The name of this database in a multi-database configuration. The name is used when constructing cross-database references ans when accessing database connections fron other databases.
- **databases** (*dict*) – dictionary of database name to databases in a multi-database configuration. The new database will add itself to this dictionary. The dictionary is used when getting connections in other databases.
- **xrefs** (*boolean*) – Flag indicating whether cross-database references are allowed from this database to other databases in a multi-database configuration.
- **large_record_size** (*int*) – When object records are saved that are larger than this, a warning is issued, suggesting that blobs should be used instead.
- **storage_args** – Extra keywork arguments passed to a storage constructor if a path name or None is passed as the storage argument.

cacheDetail()
Return object counts by class accross all connections.

cacheDetailSize()
Return non-ghost counts sizes for all connections.

cacheExtremeDetail()
Return information about all of the objects in the object caches.

Information includes a connection number, class, object id, reference count and state. The reference count returned excludes references help by ZODB itself.

cacheMinimize()
Minimize cache sizes for all connections

cacheSize()
Return the total count of non-ghost objects in all object caches

close()
Close the database and its underlying storage.

It is important to close the database, because the storage may flush in-memory data structures to disk when it is closed. Leaving the storage open with the process exits can cause the next open to be slow.

What effect does closing the database have on existing connections? Technically, they remain open, but their storage is closed, so they stop behaving usefully. Perhaps close() should also close all the Connections.

connectionDebugInfo()
Get debugging information about connections

This is especially useful to debug connections that seem to be leaking or open too long. Information includes connection info, the connection before setting, and, if a connection is open, the time it was opened. The info is the result of calling *getDebugInfo()* on the connection, and the connection's cache size.

getCacheSize()
Get the configured cache size (objects).

getCacheSizeBytes()
Get the configured cache size in bytes.

getHistoricalCacheSize()
Get the configured historical cache size (objects).

getHistoricalCacheSizeBytes()
Get the configured historical cache size in bytes.

getHistoricalPoolSize()
Get the configured historical pool size

getHistoricalTimeout()
Get the configured historical pool timeout

getName()
Get the storage name

getPoolSize()
Get the configured pool size

getSize()
Get the approximate database size, in bytes

history(*oid*, *size=1*)
Get revision history information for an object.

See *ZODB.interfaces.IStorage.history()*.

lastTransaction()
Get the storage last transaction id.

objectCount()
Get the approximate object count

open(*transaction_manager=None*, *at=None*, *before=None*)
Return a database Connection for use by application code.

Note that the connection pool is managed as a stack, to increase the likelihood that the connection's stack will include useful objects.

Parameters

- *transaction_manager*: transaction manager to use. None means use the default transaction manager.
- *at*: a datetime.datetime or 8 character transaction id of the time to open the database with a read-only connection. Passing both *at* and *before* raises a ValueError, and passing neither opens a standard writable transaction of the newest state. A timezone-naive datetime.datetime is treated as a UTC value.
- *before*: like *at*, but opens the readonly state before the tid or datetime.

pack (*t=None*, *days=0*)
Pack the storage, deleting unused object revisions.

A pack is always performed relative to a particular time, by default the current time. All object revisions that are not reachable as of the pack time are deleted from the storage.

The cost of this operation varies by storage, but it is usually an expensive operation.

There are two optional arguments that can be used to set the pack time: t, pack time in seconds since the epcoh, and days, the number of days to subtract from t or from the current time if t is not specified.

setCacheSize (*size*)
Reconfigure the cache size (non-ghost object count)

setCacheSizeBytes (*size*)
Reconfigure the cache total size in bytes

setHistoricalCacheSize (*size*)
Reconfigure the historical cache size (non-ghost object count)

setHistoricalCacheSizeBytes (*size*)
Reconfigure the historical cache total size in bytes

setHistoricalPoolSize (*size*)
Reconfigure the connection historical pool size

setHistoricalTimeout (*timeout*)
Reconfigure the connection historical pool timeout

setPoolSize (*size*)
Reconfigure the connection pool size

storage = storage object
Database storage, implementing *IStorage*

supportsUndo ()
Return whether the database supports undo.

transaction (*note=None*)
Execute a block of code as a transaction.

If a note is given, it will be added to the transaction's description.

The `transaction` method returns a context manager that can be used with the `with` statement.

undo (*id*, *txn=None*)
Undo a transaction identified by id.

A transaction can be undone if all of the objects involved in the transaction were not modified subsequently, if any modifications can be resolved by conflict resolution, or if subsequent changes resulted in the same object state.

The value of id should be generated by calling undoLog() or undoInfo(). The value of id is not the same as a transaction id used by other methods; it is unique to undo().

Parameters

- *id*: a transaction identifier
- *txn*: transaction context to use for undo(). By default, uses the current transaction.

undoInfo(**args*, ***kw*)
Return a sequence of descriptions for transactions.

See *ZODB.interfaces.IStorageUndoable.undoInfo()*.

undoLog(**args*, ***kw*)
Return a sequence of descriptions for transactions.

See *ZODB.interfaces.IStorageUndoable.undoLog()*.

undoMultiple(*ids*, *txn=None*)
Undo multiple transactions identified by ids.

A transaction can be undone if all of the objects involved in the transaction were not modified subsequently, if any modifications can be resolved by conflict resolution, or if subsequent changes resulted in the same object state.

The values in ids should be generated by calling undoLog() or undoInfo(). The value of ids are not the same as a transaction ids used by other methods; they are unique to undo().

Parameters

- *ids*: a sequence of storage-specific transaction identifiers
- *txn*: transaction context to use for undo(). By default, uses the current transaction.

Database text configuration

Databases are configured with `zodb` sections:

```
<zodb>
  cache-size-bytes 100MB
  <mappingstorage>
  </mappingstorage>
</zodb>
```

A `zodb` section must have a storage sub-section specifying a storage and any of the following options:

allow-implicit-cross-references (*boolean*) If set to false, implicit cross references (the only kind currently possible) are disallowed.

cache-size (*integer*, default: 5000) Target size, in number of objects, of each connection's object cache.

cache-size-bytes (*byte-size*, default: 0) Target size, in total estimated size for objects, of each connection's object cache. "0" means no limit.

database-name (*string*) When multi-databases are in use, this is the name given to this database in the collection. The name must be unique across all databases in the collection. The collection must also be given a mapping

from its databases' names to their databases, but that cannot be specified in a ZODB config file. Applications using multi-databases typical supply a way to configure the mapping in their own config files, using the "databases" parameter of a DB constructor.

historical-cache-size (*integer*, default: 1000) Target size, in number of objects, of each historical connection's object cache.

historical-cache-size-bytes (*byte-size*, default: 0) Target size, in total estimated size of objects, of each historical connection's object cache.

historical-pool-size (*integer*, default: 3) The expected maximum total number of historical connections simultaneously open.

historical-timeout (*time-interval*, default: 5m) The minimum interval that an unused historical connection should be kept.

large-record-size (*byte-size*, default: 16MB) When object records are saved that are larger than this, a warning is issued, suggesting that blobs should be used instead.

pool-size (*integer*, default: 7) The expected maximum number of simultaneously open connections. There is no hard limit (as many connections as are requested will be opened, until system resources are exhausted). Exceeding pool-size connections causes a warning message to be logged, and exceeding twice pool-size connections causes a critical message to be logged.

pool-timeout (*time-interval*) The minimum interval that an unused (non-historical) connection should be kept.

For a multi-database configuration, use multiple `zodb` sections and give the sections names:

```
<zodb first>
  cache-size-bytes 100MB
  <mappingstorage>
  </mappingstorage>
</zodb>

<zodb second>
  <mappingstorage>
  </mappingstorage>
</zodb>
```

When the configuration is loaded, a single database will be returned, but all of the databases will be available through the returned database's `databases` attribute.

Connections

class ZODB.Connection.**Connection**(*db*, *cache_size=400*, *before=None*, *cache_size_bytes=0*)
Connection to ZODB for loading and storing objects.

Connections manage object state in collaboration with transaction managers. They're created by calling the *open()* method on *database* objects.

add(*obj*)
Add a new object 'obj' to the database and assign it an oid.

cacheGC()
Reduce cache size to target size.

cacheMinimize()
Deactivate all unmodified objects in the cache.

close(*primary=True*)
Close the Connection.

db()
Returns a handle to the database this connection belongs to.

get(*oid*)
Return the persistent object with oid 'oid'.

getDebugInfo()
Returns a tuple with different items for debugging the connection.

get_connection(*database_name*)
Return a Connection for the named database.

isReadOnly()
Returns True if this connection is read only.

oldstate(*obj*, *tid*)
Return copy of 'obj' that was written by transaction 'tid'.

onCloseCallback(*f*)
Register a callable, f, to be called by close().

root
Return the database root object.

setDebugInfo(**args*)
Add the given items to the debug information of this connection.

sync()
Manually update the view on the database.

transaction_manager = current transaction manager
Transaction manager associated with the connection when it was opened.

TimeStamp (transaction ids)

class ZODB.TimeStamp.**TimeStamp**(*year*, *month*, *day*, *hour*, *minute*, *seconds*)
Create a time-stamp object. Time stamps facilitate the computation of transaction ids, which are based on times. The arguments are integers, except for seconds, which may be a floating-point number. Time stamps have microsecond precision. Time stamps are implicitly UTC based.

Time stamps are orderable and hashable.

day()
Return the time stamp's day.

hour()
Return the time stamp's hour.

laterThan(*other*)
Return a timestamp instance which is later than 'other'.

If self already qualifies, return self.

Otherwise, return a new instance one moment later than 'other'.

minute()
Return the time stamp's minute.

month()
Return the time stamp's month.

raw()
Get an 8-byte representation of the time stamp for use in APIs that require a time stamp.

second()
Return the time stamp's second.

timeTime()
Return the time stamp as seconds since the epoc, as used by the `time` module.

year()
Return the time stamp's year.

Loading configuration

Open database and storage from a configuration.

`ZODB.config.`**databaseFromString**(*s*)
Create a database from a database-configuration string.

The string must contain one or more *zodb* sections.

The database defined by the first section is returned.

If *more than one zodb section is provided*, a multi-database configuration will be created and all of the databases will be available in the returned database's `databases` attribute.

`ZODB.config.`**databaseFromFile**(*f*)
Create a database from a file object that provides configuration.

See *databaseFromString()*.

`ZODB.config.`**databaseFromURL**(*url*)
Load a database from URL (or file name) that provides configuration.

See *databaseFromString()*.

`ZODB.config.`**storageFromString**(*s*)
Create a storage from a storage-configuration string.

`ZODB.config.`**storageFromFile**(*f*)
Create a storage from a file object providing storage-configuration.

`ZODB.config.`**storageFromURL**(*url*)
Create a storage from a URL (or file name) providing storage-configuration.

Storage APIs

Contents

Storage interfaces

There are various storage implementations that implement standard storage interfaces. They differ primarily in their constructors.

Application code rarely calls storage methods, and those it calls are generally called indirectly through databases. There are interface-defined methods that are called internally by ZODB. These aren't shown below.

IStorage

interface `ZODB.interfaces.`**`IStorage`**

A storage is responsible for storing and retrieving data of objects.

Consistency and locking

When transactions are committed, a storage assigns monotonically increasing transaction identifiers (tids) to the transactions and to the object versions written by the transactions. ZODB relies on this to decide if data in object caches are up to date and to implement multi-version concurrency control.

There are methods in IStorage and in derived interfaces that provide information about the current revisions (tids) for objects or for the database as a whole. It is critical for the proper working of ZODB that the resulting tids are increasing with respect to the object identifier given or to the databases. That is, if there are 2 results for an object or for the database, R1 and R2, such that R1 is returned before R2, then the tid returned by R2 must be greater than or equal to the tid returned by R1. (When thinking about results for the database, think of these as results for all objects in the database.)

This implies some sort of locking strategy. The key method is tcp_finish, which causes new tids to be generated and also, through the callback passed to it, returns new current tids for the objects stored in a transaction and for the database as a whole.

The IStorage methods affected are lastTransaction, load, store, and tpc_finish. Derived interfaces may introduce additional methods.

`__len__``()`

The approximate number of objects in the storage

This is used soley for informational purposes.

close()
Close the storage.

Finalize the storage, releasing any external resources. The storage should not be used after this method is called.

Note that databases close their storages when they're closed, so this method isn't generally called from application code.

getName()
The name of the storage

The format and interpretation of this name is storage dependent. It could be a file name, a database name, etc..

This is used soley for informational purposes.

getSize()
An approximate size of the database, in bytes.

This is used soley for informational purposes.

history(*oid*, *size=1*)
Return a sequence of history information dictionaries.

Up to size objects (including no objects) may be returned.

The information provides a log of the changes made to the object. Data are reported in reverse chronological order.

Each dictionary has the following keys:

time UTC seconds since the epoch (as in time.time) that the object revision was committed.

tid The transaction identifier of the transaction that committed the version.

serial An alias for tid, which expected by older clients.

user_name The bytes user identifier, if any (or an empty string) of the user on whos behalf the revision was committed.

description The bytes transaction description for the transaction that committed the revision.

size The size of the revision data record.

If the transaction had extension items, then these items are also included if they don't conflict with the keys above.

isReadOnly()
Test whether a storage allows committing new transactions

For a given storage instance, this method always returns the same value. Read-only-ness is a static property of a storage.

lastTransaction()
Return the id of the last committed transaction.

If no transactions have been committed, return a string of 8 null (0) characters.

pack(*pack_time*, *referencesf*)
Pack the storage

It is up to the storage to interpret this call, however, the general idea is that the storage free space by:

- discarding object revisions that were old and not current as of the given pack time.

- garbage collecting objects that aren't reachable from the root object via revisions remaining after discarding revisions that were not current as of the pack time.

The pack time is given as a UTC time in seconds since the epoch.

The second argument is a function that should be used to extract object references from database records. This is needed to determine which objects are referenced from object revisions.

sortKey()
Sort key used to order distributed transactions

When a transaction involved multiple storages, 2-phase commit operations are applied in sort-key order. This must be unique among storages used in a transaction. Obviously, the storage can't assure this, but it should construct the sort key so it has a reasonable chance of being unique.

The result must be a string.

IStorageIteration

interface ZODB.interfaces.**IStorageIteration**
API for iterating over the contents of a storage.

iterator(*start=None*, *stop=None*)
Return an IStorageTransactionInformation iterator.

If the start argument is not None, then iteration will start with the first transaction whose identifier is greater than or equal to start.

If the stop argument is not None, then iteration will end with the last transaction whose identifier is less than or equal to stop.

The iterator provides access to the data as available at the time when the iterator was retrieved.

IStorageUndoable

interface ZODB.interfaces.**IStorageUndoable**
A storage supporting transactional undo.

undoInfo(*first=0*, *last=-20*, *specification=None*)
Return a sequence of descriptions for undoable transactions.

This is like *undoLog()*, except for the *specification* argument. If given, *specification* is a dictionary, and *undoInfo()* synthesizes a *filter* function *f* for *undoLog()* such that *f(desc)* returns true for a transaction description mapping *desc* if and only if *desc* maps each key in *specification* to the same value *specification* maps that key to. In other words, only extensions (or supersets) of *specification* match.

ZEO note: *undoInfo()* passes the *specification* argument from a ZEO client to its ZEO server (while a ZEO client ignores any *filter* argument passed to *undoLog()*).

undoLog(*first*, *last*, *filter=None*)
Return a sequence of descriptions for undoable transactions.

Application code should call undoLog() on a DB instance instead of on the storage directly.

A transaction description is a mapping with at least these keys:

"time": The time, as float seconds since the epoch, when the transaction committed.

"user_name": The bytes value of the *.user* attribute on that transaction.

"description": The bytes value of the *.description* attribute on that transaction.

"id'" A bytes uniquely identifying the transaction to the storage. If it's desired to undo this transaction, this is the *transaction_id* to pass to *undo()*.

In addition, if any name+value pairs were added to the transaction by *setExtendedInfo()*, those may be added to the transaction description mapping too (for example, FileStorage's *undoLog()* does this).

filter is a callable, taking one argument. A transaction description mapping is passed to *filter* for each potentially undoable transaction. The sequence returned by *undoLog()* excludes descriptions for which *filter* returns a false value. By default, *filter* always returns a true value.

ZEO note: Arbitrary callables cannot be passed from a ZEO client to a ZEO server, and a ZEO client's implementation of *undoLog()* ignores any *filter* argument that may be passed. ZEO clients should use the related *undoInfo()* method instead (if they want to do filtering).

Now picture a list containing descriptions of all undoable transactions that pass the filter, most recent transaction first (at index 0). The *first* and *last* arguments specify the slice of this (conceptual) list to be returned:

***first*: This is the index of the first transaction description** in the slice. It must be >= 0.

***last*: If >= 0, first:last acts like a Python slice, selecting** the descriptions at indices *first*, first+1, ..., up to but not including index *last*. At most last-first descriptions are in the slice, and *last* should be at least as large as *first* in this case. If *last* is less than 0, then abs(last) is taken to be the maximum number of descriptions in the slice (which still begins at index *first*). When *last* < 0, the same effect could be gotten by passing the positive first-last for *last* instead.

IStorageCurrentRecordIteration

interface `ZODB.interfaces.`**`IStorageCurrentRecordIteration`**

`record_iternext`(*next=None*)

Iterate over the records in a storage

Use like this:

```
>>> next = None
>>> while 1:
...     oid, tid, data, next = storage.record_iternext(next)
...     # do things with oid, tid, and data
...     if next is None:
...         break
```

IBlobStorage

interface `ZODB.interfaces.`**`IBlobStorage`**

A storage supporting BLOBs.

`temporaryDirectory`()

Return a directory that should be used for uncommitted blob data.

If Blobs use this, then commits can be performed with a simple rename.

IStorageRecordInformation

interface ZODB.interfaces.**IStorageRecordInformation**
Provide information about a single storage record

data = <zope.interface.interface.Attribute object>
The data record, bytes

data_txn = <zope.interface.interface.Attribute object>
The previous transaction id, bytes

oid = <zope.interface.interface.Attribute object>
The object id, bytes

tid = <zope.interface.interface.Attribute object>
The transaction id, bytes

IStorageTransactionInformation

interface ZODB.interfaces.**IStorageTransactionInformation**
Provide information about a storage transaction.

Can be iterated over to retrieve the records modified in the transaction.

Note that this may contain a status field used by FileStorage to support packing. At some point, this will go away when FileStorage has a better pack algoritm.

__iter__()
Iterate over the transaction's records given as IStorageRecordInformation objects.

tid = <zope.interface.interface.Attribute object>
Transaction id

Included storages

FileStorage

class ZODB.FileStorage.FileStorage.**FileStorage**(*file_name*, *create=False*, *read_only=False*, *stop=None*, *quota=None*, *pack_gc=True*, *pack_keep_old=True*, *packer=None*, *blob_dir=None*)
Storage that saves data in a file

__init__(*file_name*, *create=False*, *read_only=False*, *stop=None*, *quota=None*, *pack_gc=True*, *pack_keep_old=True*, *packer=None*, *blob_dir=None*)
Create a file storage

Parameters

- **file_name** (*str*) – Path to store data file
- **create** (*bool*) – Flag indicating whether a file should be created even if it already exists.
- **read_only** (*bool*) – Flag indicating whether the file is read only. Only one process is able to open the file non-read-only.
- **stop** (*bytes*) – Time-travel transaction id When the file is opened, data will be read up to the given transaction id. Transaction ids correspond to times and you can compute transaction ids for a given time using *TimeStamp*.

- **quota** (*int*) – File-size quota
- **pack_gc** (*bool*) – Flag indicating whether garbage collection should be performed when packing.
- **pack_keep_old** (*bool*) – flag indicating whether old data files should be retained after packing as a `.old` file.
- **packer** (*callable*) – An alternative *packer*.
- **blob_dir** (*str*) – A blob-directory path name. Blobs will be supported if this option is provided.

A file storage stores data in a single file that behaves like a traditional transaction log. New data records are appended to the end of the file. Periodically, the file is packed to free up space. When this is done, current records as of the pack time or later are copied to a new file, which replaces the old file.

FileStorages keep in-memory indexes mapping object oids to the location of their current records in the file. Back pointers to previous records allow access to non-current records from the current records.

In addition to the data file, some ancillary files are created. These can be lost without affecting data integrity, however losing the index file may cause extremely slow startup. Each has a name that's a concatenation of the original file and a suffix. The files are listed below by suffix:

.index Snapshot of the in-memory index. This are created on shutdown, packing, and after rebuilding an index when one was not found. For large databases, creating a file-storage object without an index file can take very long because it's necessary to scan the data file to build the index.

.lock A lock file preventing multiple processes from opening a file storage on non-read-only mode.

.tmp A file used to store data being committed in the first phase of 2-phase commit

.index_tmp A temporary file used when saving the in-memory index to avoid overwriting an existing index until a new index has been fully saved.

.pack A temporary file written while packing containing current records as of and after the pack time.

.old The previous database file after a pack.

When the database is packed, current records as of the pack time and later are written to the `.pack` file. At the end of packing, the `.old` file is removed, if it exists, and the data file is renamed to the `.old` file and finally the `.pack` file is rewritten to the data file.

interface `ZODB.FileStorage.interfaces.`**IFileStoragePacker**

__call__(*storage*, *referencesf*, *stop*, *gc*)
Pack the file storage into a new file

Parameters

- **storage** (`FileStorage`) – The storage object to be packed
- **referencesf** (*callable*) – A function that extracts object references from a pickle bytes string. This is usually `ZODB.serialize.referencesf`.
- **stop** (*bytes*) – A transaction id representing the time at which to stop packing.
- **gc** (*bool*) – A flag indicating whether garbage collection should be performed.

The new file will have the same name as the old file with `.pack` appended. (The packer can get the old file name via storage._file.name.) If blobs are supported, if the storages blob_dir attribute is not None or empty, then a .removed file must be created in the blob directory. This file contains records of the form:

```
(oid+serial).encode('hex')+'\n'
```

or, of the form:

```
oid.encode('hex')+'\n'
```

If packing is unnecessary, or would not change the file, then no pack or removed files are created None is returned, otherwise a tuple is returned with:

- the size of the packed file, and
- the packed index

If and only if packing was necessary (non-None) and there was no error, then the commit lock must be acquired. In addition, it is up to FileStorage to:

- Rename the .pack file, and
- process the blob_dir/.removed file by removing the blobs corresponding to the file records.

FileStorage text configuration

File storages are configured using the `filestorage` section:

```
<filestorage>
  path Data.fs
</filestorage>
```

which accepts the following options:

blob-dir (*existing-dirpath*) If supplied, the file storage will provide blob support and this is the name of a directory to hold blob data. The directory will be created if it doesn't exist. If no value (or an empty value) is provided, then no blob support will be provided. (You can still use a BlobStorage to provide blob support.)

create (*boolean*) Flag that indicates whether the storage should be truncated if it already exists.

pack-gc (*boolean*, default: true) If false, then no garbage collection will be performed when packing. This can make packing go much faster and can avoid problems when objects are referenced only from other databases.

pack-keep-old (*boolean*, default: true) If true, a copy of the database before packing is kept in a ".old" file.

packer (*string*) The dotted name (dotted module name and object name) of a packer object. This is used to provide an alternative pack implementation.

path (*existing-dirpath*, required) Path name to the main storage file. The names for supplemental files, including index and lock files, will be computed from this.

quota (*byte-size*) Maximum allowed size of the storage file. Operations which would cause the size of the storage to exceed the quota will result in a ZODB.FileStorage.FileStorageQuotaError being raised.

read-only (*boolean*) If true, only reads may be executed against the storage. Note that the "pack" operation is not considered a write operation and is still allowed on a read-only filestorage.

MappingStorage

class ZODB.MappingStorage.**MappingStorage**(*name='MappingStorage'*)
In-memory storage implementation

Note that this implementation is somewhat naive and inefficient with regard to locking. Its implementation is primarily meant to be a simple illustration of storage implementation. It's also useful for testing and exploration where scalability and efficiency are unimportant.

__init__(*name='MappingStorage'*)
Create a mapping storage

The name parameter is used by the *getName()* and *sortKey()* methods.

MappingStorage text configuration

File storages are configured using the `mappingstorage` section:

```
<mappingstorage>
</mappingstorage>
```

Options:

name (*string*, default: Mapping Storage) The storage name, used by the *getName()* and *sortKey()* methods.

DemoStorage

class ZODB.DemoStorage.**DemoStorage**(*name=None*, *base=None*, *changes=None*, *close_base_on_close=None*, *close_changes_on_close=None*)
A storage that stores changes against a read-only base database

This storage was originally meant to support distribution of application demonstrations with populated read-only databases (on CDROM) and writable in-memory databases.

Demo storages are extemely convenient for testing where setup of a base database can be shared by many tests.

Demo storages are also handy for staging appplications where a read-only snapshot of a production database (often accomplished using a beforestorage) is combined with a changes database implemented with a *FileStorage*.

__init__(*name=None*, *base=None*, *changes=None*, *close_base_on_close=None*, *close_changes_on_close=None*)
Create a demo storage

Parameters

- **name** (*str*) – The storage name used by the *getName()* and *sortKey()* methods.
- **base** (*object*) – base storage
- **changes** (*object*) – changes storage
- **close_base_on_close** (*bool*) – A Flag indicating whether the base database should be closed when the demo storage is closed.
- **close_changes_on_close** (*bool*) – A Flag indicating whether the changes database should be closed when the demo storage is closed.

If a base database isn't provided, a *MappingStorage* will be constructed and used.

If `close_base_on_close` isn't specified, it will be `True` if a base database was provided and `False` otherwise.

If a changes database isn't provided, a *MappingStorage* will be constructed and used and blob support will be provided using a temporary blob directory.

If `close_changes_on_close` isn't specified, it will be `True` if a changes database was provided and `False` otherwise.

pop()
Close the changes database and return the base.

push(*changes=None*)
Create a new demo storage using the storage as a base.

The given changes are used as the changes for the returned storage and `False` is passed as `close_base_on_close`.

DemoStorage text configuration

Demo storages are configured using the `demostorage` section:

```
<demostorage>
  <filestorage base>
    path base.fs
  </filestorage>
  <mappingstorage changes>
    name Changes
  </mappingstorage>
</demostorage>
```

`demostorage` sections can contain up to 2 storage subsections, named `base` and `changes`, specifying the demo storage's base and changes storages. See *ZODB.DemoStorage.DemoStorage.__init__()* for more on the base and changes storages.

Options:

name (*string*) The storage name, used by the *getName()* and *sortKey()* methods.

Noteworthy non-included storages

A number of important ZODB storages are distributed separately.

Base storages

Unlike the included storages, all the implementations listed in this section allow multiple processes to share the same database.

NEO NEO can spread data among several computers for load-balancing and multi-master replication. It also supports asynchronous replication to off-site NEO databases for further disaster resistance without affecting local operation latency.

For more information, see https://lab.nexedi.com/nexedi/neoppod.

RelStorage RelStorage stores data in relational databases. This is especially useful when you have requirements or existing infrastructure for storing data in relational databases.

For more information, see http://relstorage.readthedocs.io/en/latest/.

ZEO ZEO is a client-server database implementation for ZODB. To use ZEO, you run a ZEO server, and use ZEO clients in your application.

For more information, see https://github.com/zopefoundation/ZEO.

Optional layers

ZRS ZRS provides replication from one database to another. It's most commonly used with ZEO. With ZRS, you create a ZRS primary database around a *FileStorage* and in a separate process, you create a ZRS secondary storage around any *storage*. As transactions are committed on the primary, they're copied asynchronously to secondaries.

For more information, see https://github.com/zc/zrs.

zlibstorage zlibstorage compresses database records using the compression algorithm used by gzip.

For more information, see https://pypi.python.org/pypi/zc.zlibstorage.

beforestorage beforestorage provides a point-in-time view of a database that might be changing. This can be useful to provide a non-changing view of a production database for use with a *DemoStorage*.

For more information, see https://pypi.python.org/pypi/zc.beforestorage.

cipher.encryptingstorage cipher.encryptingstorage provided compression and encryption of database records.

For more information, see https://pypi.python.org/pypi/cipher.encryptingstorage/.

Transactions

Transaction support is provided by the transaction package[1], which is installed automatically when you install ZODB. There are two important APIs provided by the transaction package, `ITransactionManager` and `ITransaction`, described below.

ITransactionManager

interface `transaction.interfaces.`**`ITransactionManager`**

An object that manages a sequence of transactions.

Applications use transaction managers to establish transaction boundaries.

`abort``()`

Abort the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

`begin``()`

Explicitly begin and return a new transaction.

If an existing transaction is in progress and the transaction manager not in explicit mode, the previous transaction will be aborted. If an existing transaction is in progress and the transaction manager is in explicit mode, an `AlreadyInTransaction` exception will be raised..

The `newTransaction` method of registered synchronizers is called, passing the new transaction object.

Note that when not in explicit mode, transactions may be started implicitly without calling `begin`. In that case, `newTransaction` isn't called because the transaction manager doesn't know when to call it. The transaction is likely to have begun long before the transaction manager is involved. (Conceivably the `commit` and `abort` methods could call `begin`, but they don't.)

`commit``()`

Commit the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

[1] The :mod:transaction package is a general purpose package for managing distributed transactions with a two-phase commit protocol. It can and occasionally is used with packages other than ZODB.

doom()
Doom the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

get()
Get the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

isDoomed()
Returns True if the current transaction is doomed, otherwise False.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

savepoint(*optimistic=False*)
Create a savepoint from the current transaction.

If the optimistic argument is true, then data managers that don't support savepoints can be used, but an error will be raised if the savepoint is rolled back.

An ISavepoint object is returned.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

ITransaction

interface `transaction.interfaces.`**ITransaction**
Object representing a running transaction.

Objects with this interface may represent different transactions during their lifetime (.begin() can be called to start a new transaction using the same instance, although that example is deprecated and will go away in ZODB 3.6).

abort()
Abort the transaction.

This is called from the application. This can only be called before the two-phase commit protocol has been started.

addAfterCommitHook(*hook*, *args=()*, *kws=None*)
Register a hook to call after a transaction commit attempt.

The specified hook function will be called after the transaction commit succeeds or aborts. The first argument passed to the hook is a Boolean value, true if the commit succeeded, or false if the commit aborted. *args* specifies additional positional, and *kws* keyword, arguments to pass to the hook. *args* is a sequence of positional arguments to be passed, defaulting to an empty tuple (only the true/false success argument is passed). *kws* is a dictionary of keyword argument names and values to be passed, or the default None (no keyword arguments are passed).

Multiple hooks can be registered and will be called in the order they were registered (first registered, first called). This method can also be called from a hook: an executing hook can register more hooks. Applications should take care to avoid creating infinite loops by recursively registering hooks.

Hooks are called only for a top-level commit. A savepoint creation does not call any hooks. Calling a hook "consumes" its registration: hook registrations do not persist across transactions. If it's desired to call the same hook on every transaction commit, then addAfterCommitHook() must be called with that hook during every transaction; in such a case consider registering a synchronizer object via a TransactionManager's registerSynch() method instead.

addBeforeCommitHook(*hook*, *args=()*, *kws=None*)
Register a hook to call before the transaction is committed.

The specified hook function will be called after the transaction's commit method has been called, but before the commit process has been started. The hook will be passed the specified positional (*args*) and keyword (*kws*) arguments. *args* is a sequence of positional arguments to be passed, defaulting to an empty tuple (no positional arguments are passed). *kws* is a dictionary of keyword argument names and values to be passed, or the default None (no keyword arguments are passed).

Multiple hooks can be registered and will be called in the order they were registered (first registered, first called). This method can also be called from a hook: an executing hook can register more hooks. Applications should take care to avoid creating infinite loops by recursively registering hooks.

Hooks are called only for a top-level commit. A savepoint creation does not call any hooks. If the transaction is aborted, hooks are not called, and are discarded. Calling a hook "consumes" its registration too: hook registrations do not persist across transactions. If it's desired to call the same hook on every transaction commit, then addBeforeCommitHook() must be called with that hook during every transaction; in such a case consider registering a synchronizer object via a TransactionManager's registerSynch() method instead.

commit()
Finalize the transaction.

This executes the two-phase commit algorithm for all IDataManager objects associated with the transaction.

description = <zope.interface.interface.Attribute object>
A textual description of the transaction.

The value is text (unicode). Method note() is the intended way to set the value. Storages record the description, as meta-data, when a transaction commits.

A storage may impose a limit on the size of the description; behavior is undefined if such a limit is exceeded (for example, a storage may raise an exception, or truncate the value).

doom()
Doom the transaction.

Dooms the current transaction. This will cause DoomedTransactionException to be raised on any attempt to commit the transaction.

Otherwise the transaction will behave as if it was active.

getAfterCommitHooks()
Return iterable producing the registered addAfterCommit hooks.

A triple (hook, args, kws) is produced for each registered hook. The hooks are produced in the order in which they would be invoked by a top-level transaction commit.

getBeforeCommitHooks()
Return iterable producing the registered addBeforeCommit hooks.

A triple (hook, args, kws) is produced for each registered hook. The hooks are produced in the order in which they would be invoked by a top-level transaction commit.

note(*text*)
Add text (unicode) to the transaction description.

This modifies the *.description* attribute; see its docs for more detail. First surrounding whitespace is stripped from *text*. If *.description* is currently an empty string, then the stripped text becomes its value, else two newlines and the stripped text are appended to *.description*.

savepoint (*optimistic=False*)
Create a savepoint.

If the optimistic argument is true, then data managers that don't support savepoints can be used, but an error will be raised if the savepoint is rolled back.

An ISavepoint object is returned.

setExtendedInfo (*name*, *value*)
Add extension data to the transaction.

name is the text (unicode) name of the extension property to set

value must be picklable and json serializable (not an instance).

Multiple calls may be made to set multiple extension properties, provided the names are distinct.

Storages record the extension data, as meta-data, when a transaction commits.

A storage may impose a limit on the size of extension data; behavior is undefined if such a limit is exceeded (for example, a storage may raise an exception, or remove *<name, value>* pairs).

user = <zope.interface.interface.Attribute object>
A user name associated with the transaction.

The format of the user name is defined by the application. The value is text (unicode). Storages record the user value, as meta-data, when a transaction commits.

A storage may impose a limit on the size of the value; behavior is undefined if such a limit is exceeded (for example, a storage may raise an exception, or truncate the value).

ZODB articles

Contents

An overview of the ZODB (by Laurence Rowe)

ZODB in comparison to relational databases, transactions, scalability and best practice. Originally delivered to the Plone Conference 2007, Naples.

Comparison to other database types

Relational Databases are great at handling large quantities of homogenous data. If you're building a ledger system a Relational Database is a great fit. But Relational Databases only support hierarchical data structures to a limited degree. Using foreign-key relationships must refer to a single table, so only a single type can be contained.

Hierarchical databases (such as LDAP or a filesystem) are much more suitable for modelling the flexible containment hierarchies required for content management applications. But most of these systems do not support transactional semantics. ORMs such as SQLAlchemy. make working with Relational Databases in an object orientated manner much more pleasant. But they don't overcome the restrictions inherent in a relational model.

The **ZODB** is an (almost) transparent python object persistence system, heavily influenced by Smalltalk. As an Object-Orientated Database it gives you the flexibility to build a data model fit your application. For the most part you don't have to worry about persistency - you only work with python objects and it just happens in the background.

Of course this power comes at a price. While changing the methods your classes provide is not a problem, changing attributes can necessitate writing a migration script, as you would with a relational schema change. With ZODB obejcts though explicit schema migrations are not enforced, which can bite you later.

Transactions

The ZODB has a transactional support at its core. Transactions provide concurrency control and atomicity. Transactions are executed as if they have exclusive access to the data, so as an application developer you don't have to worry about threading. Of course there is nothing to prevent two simultaneous conflicting requests, So checks are made at transaction commit time to ensure consistency.

Since Zope 2.8 ZODB has implemented **Multi Version Concurrency Control**. This means no more ReadConflictErrors, each transaction is guaranteed to be able to load any object as it was when the transaction begun.

You may still see (Write) **ConflictErrors**. These can be minimised using data structures that support conflict resolution, primarily B-Trees in the BTrees library. These scalable data structures are used in Large Plone Folders and many parts of Zope. One downside is that they don't support user definable ordering.

The hot points for ConflictErrors are the catalogue indexes. Some of the indexes do not support conflict resolution and you will see ConflictErrors under write-intensive loads. On solution is to defer catalogue updates using QueueCatalog (PloneQueueCatalog), which allows indexing operations to be serialized using a seperate ZEO client. This can bring big performance benefits as request retries are reduced, but the downside is that index updates are no longer reflected immediately in the application. Another alternative is to offload text indexing to a dedicated search engine using collective.solr.

This brings us to **Atomicity**, the other key feature of ZODB transactions. A transaction will either succeed or fail, your data is never left in an inconsistent state if an error occurs. This makes Zope a forgiving system to work with.

You must though be careful with interactions with external systems. If a ConflictError occurs Zope will attempt to replay a transaction up to three times. Interactions with an external system should be made through a Data Manager that participates in the transaction. If you're talking to a database use a Zope DA or a SQLAlchemy wrapper like zope.sqlalchemy.

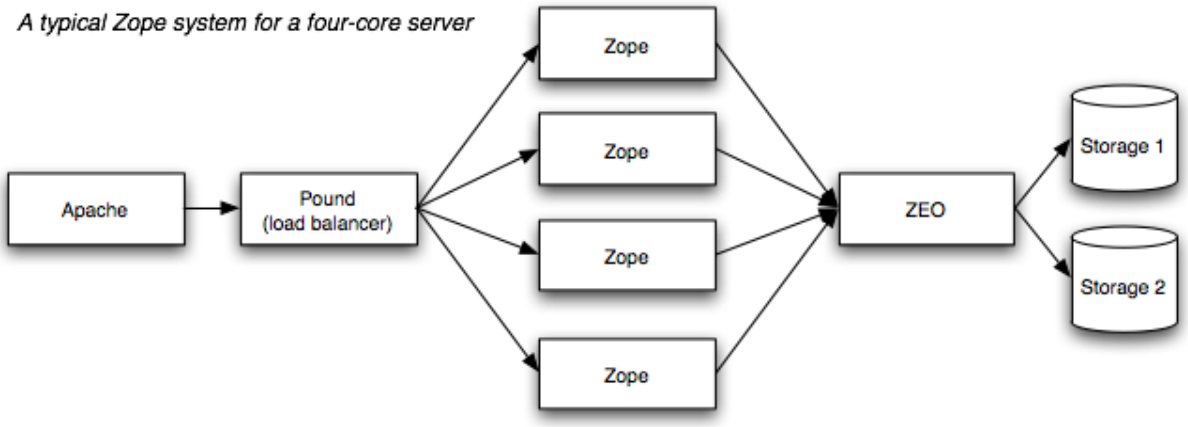
Unfortunately the default MailHost implementation used by Plone is not transaction aware. With it you can see duplicate emails sent. If this is a problem use TransactionalMailHost.

Scalability Python is limited to a single CPU by the Global Interpreter Lock, but that's ok, ZEO lets us run multiple Zope Application servers sharing a single database. You should run one Zope client for each processor on your server. ZEO also lets you connect a debug session to your database at the same time as your Zope web server, invaluable for debugging.

ZEO tends to be IO bound, so the GIL is not an issue.

ZODB also supports **partitioning**, allowing you to spread data over multiple storages. However you should be careful about cross database references (especially when copying and pasting between two databases) as they can be problematic.

Another common reason to use partitioning is because the ZODB in memory cache settings are made per database. Separating the catalogue into another storage lets you set a higher target cache size for catalogue objects than for your content objects. As much of the Plone interface is catalogue driven this can have a significant performance benefit, especially on a large site.



Storage Options

FileStorage is the default. Everything in one big Data.fs file, which is essentially a transaction log. Use this unless you have a very good reason not to.

DirectoryStorage (site) stores one file per object revision. Does not require the Data.fs.index to be rebuilt on an unclean shutdown (which can take a significant time for a large database). Small number of users.

RelStorage (pypi) stores pickles in a relational database. PostgreSQL, MySQL and Oracle are supported and no ZEO server is required. You benefit from the faster network layers of these database adapters. However, conflict resolution is moved to the application server, which can be bad for worst case performance when you have high network latency.

BDBStorage, OracleStorage, PGStorage and APE have now fallen by the wayside.

Other features

Savepoints (previously sub-transactions) allow fine grained error control and objects to be garbage collected during a transaction, saving memory.

Versions are deprecated (and will be removed in ZODB 3.9). The application layer is responsible for versioning, e.g. CMFEditions / ZopeVersionControl.

Undo, don't rely on it! If your object is indexed it may prove impossible to undo the transaction (independently) if a later transaction has changed the same index. Undo is only performed on a single database, so if you have separated out your catalogue it will get out of sync. Fine for undoing in portal_skins/custom though.

BLOBs are new in ZODB 3.8 / Zope 2.11, bringing efficient large file support. Great for document management applications.

Packing removes old revisions of objects. Similar to Routine Vacuuming in PostgreSQL.

Some best practice

Don't write on read. Your Data.fs should not grow on a read. Beware of setDefault and avoid inplace migration.

Keep your code on the filesystem. Too much stuff in the custom folder will just lead to pain further down the track. Though this can be very convenient for getting things done when they are needed yesterday...

Use scalable data structures such as BTrees. Keep your content objects simple, add functionality with adapters and views.

Introduction to the ZODB (by Michel Pelletier)

In this article, we cover the very basics of the Zope Object Database (ZODB) for Python programmers. This short article documents almost everything you need to know about using this powerful object database in Python. In a later article, I will cover some of the more advanced features of ZODB for Python programmers.

ZODB is a database for Python objects that comes with Zope. If you've ever worked with a relational database, like PostgreSQL, MySQL, or Oracle, than you should be familiar with the role of a database. It's a long term or short term storage for your application data.

For many tasks, relational databases are clearly a good solution, but sometimes relational databases don't fit well with your object model. If you have lots of different kinds of interconnected objects with complex relationships, and changing schemas then ZODB might be worth giving a try.

A major feature of ZODB is transparency. You do not need to write any code to explicitly read or write your objects to or from a database. You just put your *persistent* objects into a container that works just like a Python dictionary. Everything inside this dictionary is saved in the database. This dictionary is said to be the "root" of the database. It's like a magic bag; any Python object that you put inside it becomes persistent.

Actually there are a few restrictions on what you can store in the ZODB. You can store any objects that can be "pickled" into a standard, cross-platform serial format. Objects like lists, dictionaries, and numbers can be pickled. Objects like files, sockets, and Python code objects, cannot be stored in the database because they cannot be pickled. For more information on "pickling", see the Python pickle module documentation at http://www.python.org/doc/current/lib/module-pickle.html

A Simple Example

The first thing you need to do to start working with ZODB is to create a "root object". This process involves first opening a connection to a "storage", which is the actual back-end that stores your data.

ZODB supports many pluggable storage back-ends, but for the purposes of this article I'm going to show you how to use the 'FileStorage' back-end storage, which stores your object data in a file. Other storages include storing objects in relational databases, Berkeley databases, and a client to server storage that stores objects on a remote storage server.

To set up a ZODB, you must first install it. ZODB comes with Zope, so the easiest way to install ZODB is to install Zope and use the ZODB that comes with your Zope installation. For those of you who don't want all of Zope, but just ZODB, see the instructions for downloading StandaloneZODB from the ZODB web page.

StandaloneZODB can be installed into your system's Python libraries using the standard 'distutils' Python module.

After installing ZODB, you can start to experiment with it right from the Python command line interpreter. For example, try the following python code in your interpreter:

```
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage('mydatabase.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()
```

Here, you create storage and use the 'mydatabse.fs' file to store the object information. Then, you create a database that uses that storage.

Next, the database needs to be "opened" by calling the 'open()' method. This will return a connection object to the database. The connection object then gives you access to the 'root' of the database with the 'root()' method.

The 'root' object is the dictionary that holds all of your persistent objects. For example, you can store a simple list of strings in the root object:

```
>>> root['employees'] = ['Mary', 'Jo', 'Bob']
```

Now, you have changed the persistent database by adding a new object, but this change is so far only temporary. In order to make the change permanent, you must commit the current transaction:

```
>>> import transaction
>>> transaction.commit()
```

Transactions group of lots of changes in one atomic operation. In a later article, I'll show you how this is a very powerful feature. For now, you can think of committing transactions as "checkpoints" where you save the changes you've made to your objects so far. Later on, I'll show you how to abort those changes, and how to undo them after they are committed.

Now let's find out if our data was actually saved. First close the database connection:

```
>>> connection.close()
```

Then quit Python. Now start the Python interpreter up again, and connect to the database you just created:

```
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage('mydatabase.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()
```

Now, let's see what's in the root:

```
>>> root.items()
[('employees', ['Mary', 'Jo', 'Bob'])]
```

There's your list. If you had used a relational database, you would have had to issue a SQL query to save even a simple Python list like the above example. You would have also needed some code to convert a SQL query back into the list when you wanted to use it again. You don't have to do any of this work when using ZODB. Using ZODB is almost completely transparent, in fact, ZODB based programs often look suspiciously simple!

Keep in mind that ZODB's persistent dictionary is just the tip of the persistent iceberg. Persistent objects can have attributes that are themselves persistent. In other words, even though you may have only one or two "top level" persistent objects as values in the persistent dictionary, you can still have thousands of sub-objects below them. This is, in fact, how Zope does it. In Zope, there is only *one* top level object that is the root "application" object for all other objects in Zope.

Detecting Changes

One thing that makes ZODB so easy to use is that it doesn't require you to keep track of your changes. All you have to do is to make changes to persistent objects and then commit a transaction. Anything that has changed will be stored in the database.

There is one exception to this rule when it comes to simple mutable Python types like lists and dictionaries. If you change a list or dictionary that is already stored in the database, then the change will *not* take effect. Consider this example:

```
>>> root['employees'].append('Bill')
>>> transaction.commit()
```

You would expect this to work, but it doesn't. The reason for this is that ZODB cannot detect that the 'employees' list changed. The 'employees' list is a mutable object that does not notify ZODB when it changes.

There are a couple of very simple ways around this problem. The simplest is to re-assign the changed object:

```
>>> employees = root['employees']
>>> employees.append('Bill')
>>> root['employees'] = employees
>>> transaction.commit()
```

Here, you move the employees list to a local variable, change the list, and then *reassign* the list back into the database and commit the transaction. This reassignment notifies the database that the list changed and needs to be saved to the database.

Later in this article, we'll show you another technique for notifying the ZODB that your objects have changed. Also, in a later article, we'll show you how to use simple, ZODB-aware list and dictionary classes that come pre-packaged with ZODB for your convenience.

Persistent Classes

The easiest way to create mutable objects that notify the ZODB of changes is to create a persistent class. Persistent classes let you store your own kinds of objects in the database. For example, consider a class that represents a employee:

```
import ZODB
from Persistence import Persistent

class Employee(Persistent):

    def setName(self, name):
        self.name = name
```

To create a persistent class, simply subclass from 'Persistent.Persistent'. Because of some special magic that ZODB does, you must first import ZODB before you can import Persistent. The 'Persistent' module is actually *created* when you import 'ZODB'.

Now, you can put Employee objects in your database:

```
>>> employees=[]
>>> for name in ['Mary', 'Joe', 'Bob']:
...     employee = Employee()
...     employee.setName(name)
...     employees.append(employee)
>>> root['employees']=employees
>>> transaction.commit()
```

Don't forget to call 'commit()', so that the changes you have made so far are committed to the database, and a new transaction is begun.

Now you can change your employees and they will be saved in the database. For example you can change Bob's name to "Robert":

```
>>> bob=root['employees'][2]
>>> bob.setName('Robert')
>>> transaction.commit()
```

You can even change attributes of persistent instaces without calling methods:

```
>>> bob=root['employees'][2]
>>> bob._coffee_prefs=('Cream', 'Sugar')
>>> transaction.commit()
```

It doesn't matter whether you change an attribute directly, or whether it's changed by a method. As you can tell, all of the normal Python language rules still work as you'd expect.

Mutable Attributes

Earlier you saw how ZODB can't detect changes to normal mutable objects like Python lists. This issue still affects you when using persistent instances. This is because persistent instances can have attributes which are normal mutable objects. For example, consider this class:

```
class Employee(Persistent):

    def __init__(self):
        self.tasks = []

    def setName(self, name):
        self.name = name

    def addTask(self, task):
        self.task.append(task)
```

When you call 'addTask', the ZODB won't know that the mutable attribute 'self.tasks' has changed. As you saw earlier, you can reassign 'self.tasks' after you change it to get around this problem. However, when you're using persistent instances, you have another choice. You can signal the ZODB that your instance has changed with the '_p_changed' attribute:

```
class Employee(Persistent):
    ...

    def addTask(self, task):
        self.task.append(task)
        self._p_changed = 1
```

To signal that this object has change, set the '_p_changed' attribute to 1. You only need to signal ZODB once, even if you change many mutable attributes.

The '_p_changed' flag leads us to one of the few rules of you must follow when creating persistent classes: your instances *cannot* have attributes that begin with '_p_', those names are reserved for use by the ZODB.

A Complete Example

Here's a complete example program. It builds on the employee examples used so far:

```
from ZODB import DB
from ZODB.FileStorage import FileStorage
from ZODB.PersistentMapping import PersistentMapping
from Persistence import Persistent
import transaction

class Employee(Persistent):
    """An employee"""
```

```
    def __init__(self, name, manager=None):
        self.name=name
        self.manager=manager

# setup the database
storage=FileStorage("employees.fs")
db=DB(storage)
connection=db.open()
root=connection.root()

# get the employees mapping, creating an empty mapping if
# necessary
if not root.has_key("employees"):
    root["employees"] = {}
employees=root["employees"]


def listEmployees():
    if len(employees.values())==0:
        print "There are no employees."
        print
        return
    for employee in employees.values():
        print "Name: %s" % employee.name
        if employee.manager is not None:
            print "Manager's name: %s" % employee.manager.name
        print

def addEmployee(name, manager_name=None):
    if employees.has_key(name):
        print "There is already an employee with this name."
        return
    if manager_name:
        try:
            manager=employees[manager_name]
        except KeyError:
            print
            print "No such manager"
            print
            return
        employees[name]=Employee(name, manager)
    else:
        employees[name]=Employee(name)

    root['employees'] = employees  # reassign to change
    transaction.commit()
    print "Employee %s added." % name
    print


if __name__=="__main__":
    while 1:
        choice=raw_input("Press 'L' to list employees, 'A' to add"
                         "an employee, or 'Q' to quit:")
        choice=choice.lower()
        if choice=="l":
            listEmployees()
        elif choice=="a":
```

```
            name=raw_input("Employee name:")
            manager_name=raw_input("Manager name:")
            addEmployee(name, manager_name)
        elif choice=="q":
            break

    # close database
    connection.close()
```

This program demonstrates a couple interesting things. First, this program shows how persistent objects can refer to each other. The 'self.manager' attribute of 'Employee' instances can refer to other 'Employee' instances. Unlike a relational database, there is no need to use indirection such as object ids when referring from one persistent object to another. You can just use normal Python references. In fact, you can even use circular references.

A final trick used by this program is to look for a persistent object and create it if it is not present. This allows you to just run this program without having to run a setup script to build the database first. If there is not database present, the program will create one and initialize it.

Conclusion

ZODB is a very simple, transparent object database for Python that is a freely available component of the Zope application server. As these examples illustrate, only a few lines of code are needed to start storing Python objects in ZODB, with no need to write SQL queries. In the next article on ZODB, we'll show you some more advanced techniques for using ZODB, like using ZODB's distributed object protocol to distribute your persistent objects across many machines.

ZODB Resources

- Andrew Kuchling's "ZODB pages" (archived)
- Zope.org "ZODB Wiki"
- Jim Fulton's "Introduction to the Zope Object Database"

Advanced ZODB for Python Programmers

In the first article in this series, "ZODB for Python Programmers":ZODB1 I covered some of the simpler aspects of Python object persistence. In this article, I'll go over some of the more advanced features of ZODB.

In addition to simple persistence, ZODB offers some very useful extras for the advanced Python application. Specificly, we'll cover the following advanced features in this article:

- Persistent-Aware Types – ZODB comes with some special, "persistent-aware" data types for storing data in a ZODB. The most useful of these is the "BTree", which is a fast, efficient storage object for lots of data.
- Voalitile Data – Not all your data is meant to be stored in the database, ZODB let's you have volatile data on your objects that does not get saved.
- Pluggable Storages – ZODB offers you the ability to use many different storage back-ends to store your object data, including files, relational databases and a special client-server storage that stores objects on a remote server.
- Conflict Resolution – When many threads try to write to the same object at the same time, you can get conflicts. ZODB offers a conflict resolution protocol that allows you to mitigate most conflicting writes to your data.
- Transactions – When you want your changes to be "all or nothing" transactions come to the rescue.

Persistent-Aware Types

You can also get around the mutable attribute problem discussed in the first article by using special types that are "persistent aware". ZODB comes with the following persistent aware mutable object types:

- PersistentList – This type works just like a list, except that changing it does not require setting _p_changed or explicitly re-assigning the attribute.
- PersistentMapping – A persistent aware dictionary, much like PersistentList.
- BTree – A dictionary-like object that can hold large collections of objects in an ordered, fast, efficient way.

BTrees offer a very powerful facility to the Python programmer:

- BTrees can hold a large collection of information in an efficient way; more objects than your computer has enough memory to hold at one time.
- BTrees are integrated into the persistence machinery to work effectively with ZODB's object cache. Recently, or heavily used objects are kept in a memory cache for speed.
- BTrees can be searched very quickly, because they are stored in an fast, balanced tree data structure.
- BTrees come in three flavors, OOBTrees, IOBTrees, OIBTrees, and IIBTrees. The last three are optimized for integer keys, values, and key-value pairs, respectively. This means that, for example, an IOBTree is meant to map an integer to an object, and is optimized for having integers keys.

Using BTrees

Suppose you track the movement of all your employees with heat-seeking cameras hidden in the ceiling tiles. Since your employees tend to frequently congregate against you, all of the tracking information could end up to be a lot of data, possibly thousands of coordinates per day per employee. Further, you want to key the coordinate on the time that it was taken, so that you can only look at where your employees were during certain times:

```
from BTrees import IOBTree
from time import time

class Employee(Persistent):

    def __init__(self):
        self.movements = IOBTree()

    def fix(self, coords):
        "get a fix on the employee"
        self.movements[int(time())] = coords

    def trackToday(self):
        "return all the movements of the
        employee in the last 24 hours"
        current_time = int(time())
        return self.movements.items(current_time - 86400,
                                    current_time)
```

In this example, the 'fix' method is called every time one of your cameras sees that employee. This information is then stored in a BTree, with the current 'time()' as the key and the 'coordinates' as the value.

Because BTrees store their information is a ordered structure, they can be quickly searched for a range of key values. The 'trackToday' method uses this feature to return a sequence of coordinates from 24 hours hence to the present.

This example shows how BTrees can be quickly searched for a range of values from a minimum to a maximum, and how you can use this technique to oppress your workforce. BTrees have a very rich API, including doing unions and intersections of result sets.

Not All Objects are Persistent

You don't have to make all of your objects persistent. Non-persistent objects are often useful to represent either "canned" behavior (classes that define methods but no state), or objects that are useful only as a "cache" that can be thrown away when your persistent object is deactivated (removed from memory when not used).

ZODB provides you with the ability to have *volatile* attributes. Volatile attributes are attributes of persistent objects that are never saved in the database, even if they are capable of being persistent. Volatile attributes begin with '_v_' are good for keeping cached information around for optimization. ZODB also provides you with access to special pickling hooks that allow you to set volatile information when an object is activated.

Imagine you had a class that stored a complex image that you needed to calculate. This calculation is expensive. Instead of calculating the image every time you called a method, it would be better to calculate it *once* and then cache the result in a volatile attribute:

```
def image(self):
    "a large and complex image of the terrain"
    if hasattr(self, '_v_image'):
        return self._v_image
    image=expensive_calculation()
    self._v_image=image
    return image
```

Here, calling 'image' the first time the object is activated will cause the method to do the expensive calculation. After the first call, the image will be cached in a volatile attribute. If the object is removed from memory, the '_v_image' attribute is not saved, so the cached image is thrown away, only to be recalculated the next time you call 'image'.

ZODB and Concurrency

Different, threads, processes, and computers on a network can open connections to a single ZODB object database. Each of these different processes keeps its own copy of the objects that it uses in memory.

The problem with allowing concurrent access is that conflicts can occur. If different threads try to commit changes to the same objects at the same time, one of the threads will raise a ConflictError. If you want, you can write your application to either resolve or retry conflicts a reasonable number of times.

Zope will retry a conflicting ZODB operation three times. This is usually pretty reasonable behavior. Because conflicts only happen when two threads write to the same object, retrying a conflict means that one thread will win the conflict and write itself, and the other thread will retry a few seconds later.

Pluggable Storages

Different processes and computers can connection to the same database using a special kind of storage called a 'ClientStorage'. A 'ClientStorage' connects to a 'StorageServer' over a network.

In the very beginning, you created a connection to the database by first creating a storage. This was of the type 'FileStorage'. Zope comes with several different back end storage objects, but one of the most interesting is the 'ClientStorage' from the Zope Enterprise Objects product (ZEO).

The 'ClientStorage' storage makes a TCP/IP connection to a 'StorageServer' (also provided with ZEO). This allows many different processes on one or machines to work with the same object database and, hence, the same objects. Each

process gets a cached "copy" of a particular object for speed. All of the 'ClientStorages' connected to a 'StorageServer' speak a special object transport and cache invalidation protocol to keep all of your computers synchronized.

Opening a 'ClientStorage' connection is simple. The following code creates a database connection and gets the root object for a 'StorageServer' listening on "localhost:12345":

```
from ZODB import DB
from ZEO import ClientStorage
storage = ClientStorage.ClientStorage('localhost', 12345)
db = DB( storage )
connection = db.open()
root = connection.root()
```

In the rare event that two processes (or threads) modify the same object at the same time, ZODB provides you with the ability to retry or resolve these conflicts yourself.

Resolving Conflicts

If a conflict happens, you have two choices. The first choice is that you live with the error and you try again. Statistically, conflicts are going to happen, but only in situations where objects are "hot-spots". Most problems like this can be "designed away"; if you can redesign your application so that the changes get spread around to many different objects then you can usually get rid of the hot spot.

Your second choice is to try and *resolve* the conflict. In many situations, this can be done. For example, consider the following persistent object:

```
class Counter(Persistent):

    self.count = 0

    def hit(self):
        self.count = self.count + 1
```

This is a simple counter. If you hit this counter with a lot of requests though, it will cause conflict errors as different threads try to change the count attribute simultaneously.

But resolving the conflict between conflicting threads in this case is easy. Both threads want to increment the self.count attribute by a value, so the resolution is to increment the attribute by the sum of the two values and make both commits happy.

To resolve a conflict, a class should define an '_p_resolveConflict' method. This method takes three arguments:

- 'oldState' – The state of the object that the changes made by the current transaction were based on. The method is permitted to modify this value.
- 'savedState' – The state of the object that is currently stored in the database. This state was written after 'oldState' and reflects changes made by a transaction that committed before the current transaction. The method is permitted to modify this value.
- 'newState' – The state after changes made by the current transaction. The method is *not* permitted to modify this value. This method should compute a new state by merging changes reflected in 'savedState' and 'newState', relative to 'oldState'.

The method should return the state of the object after resolving the differences.

Here is an example of a '_p_resolveConflict' in the 'Counter' class:

```
class Counter(Persistent):
```

```
    self.count = 0

    def hit(self):
        self.count = self.count + 1

    def _p_resolveConflict(self, oldState, savedState, newState):

        # Figure out how each state is different:
        savedDiff= savedState['count'] - oldState['count']
        newDiff= newState['count']- oldState['count']

        # Apply both sets of changes to old state:
        return oldState['count'] + savedDiff + newDiff
```

In the above example, '_p_resolveConflict' resolves the difference between the two conflicting transactions.

Transactions and Subtransactions

Transactions are a very powerful concept in databases. Transactions let you make many changes to your information as if they were all one big change. Imagine software that did online banking and allowed you to transfer money from one account to another. You would do this by deducting the amount of the transfer from one account, and adding that amount onto the other.

If an error happened while you were adding the money to the receiving account (say, the bank's computers were unavailable), then you would want to abort the transaction so that the state of the accounts went back to the way they were before you changed anything.

To abort a transaction, you need to call the 'abort' method of the transactions object:

```
>>> import transaction
>>> transaction.abort()

This will throw away all the currently changed objects and start a
new, empty transaction.
```

Subtransactions, sometimes called "inner transactions", are transactions that happen inside another transaction. Subtransactions can be commited and aborted like regular "outer" transactions. Subtransactions mostly provide you with an optimization technique.

Subtransactions can be commited and aborted. Commiting or aborting a subtransaction does not commit or abort its outer transaction, just the subtransaction. This lets you use many, fine-grained transactions within one big transaction.

Why is this important? Well, in order for a transaction to be "rolled back" the changes in the transaction must be stored in memory until commit time. By commiting a subtransaction, you are telling Zope that "I'm pretty sure what I've done so far is permenant, you can store this subtransaction somewhere other than in memory". For very, very large transactions, this can be a big memory win for you.

If you abort an outer transaction, then all of its inner subtransactions will also be aborted and not saved. If you abort an inner subtransaction, then only the changes made during that subtransaction are aborted, and the outer transaction is *not* aborted and more changes can be made and commited, including more subtransactions.

You can commit or abort a subtransaction by calling either commit() or abort() with an argument of 1:

```
transaction.commit(1) # or
transaction.abort(1)
```

Subtransactions offer you a nice way to "batch" all of your "all or none" actions into smaller "all or none" actions while still keeping the outer level "all or none" transaction intact. As a bonus, they also give you much better memory resource performance.

Conclusion

ZODB offers many advanced features to help you develop simple, but powerful python programs. In this article, you used some of the more advanced features of ZODB to handle different application needs, like storing information in large sets, using the database concurrently, and maintaining transactional integrity. For more information on ZODB, join the discussion list at zodb-dev@zope.org where you can find out more about this powerful component of Zope.

Very old ZODB programming guide

> This guide is based heavily on the work of A. M. Kuchling who wrote the original guide back in 2002 and which was published under the GNU Free Documentation License, Version 1.1. See the appendix entitled "GNU Free Documentation License" for more information.

Introduction

This guide explains how to write Python programs that use the Z Object Database (ZODB) and Zope Enterprise Objects (ZEO). The latest version of the guide is always available at http://www.zope.org/Wikis/ZODB/guide/index.html.

What is the ZODB?

The ZODB is a persistence system for Python objects. Persistent programming languages provide facilities that automatically write objects to disk and read them in again when they're required by a running program. By installing the ZODB, you add such facilities to Python.

It's certainly possible to build your own system for making Python objects persistent. The usual starting points are the `pickle` module, for converting objects into a string representation, and various database modules, such as the `gdbm` or `bsddb` modules, that provide ways to write strings to disk and read them back. It's straightforward to combine the `pickle` module and a database module to store and retrieve objects, and in fact the `shelve` module, included in Python's standard library, does this.

The downside is that the programmer has to explicitly manage objects, reading an object when it's needed and writing it out to disk when the object is no longer required. The ZODB manages objects for you, keeping them in a cache, writing them out to disk when they are modified, and dropping them from the cache if they haven't been used in a while.

OODBs vs. Relational DBs

Another way to look at it is that the ZODB is a Python-specific object-oriented database (OODB). Commercial object databases for C++ or Java often require that you jump through some hoops, such as using a special preprocessor or avoiding certain data types. As we'll see, the ZODB has some hoops of its own to jump through, but in comparison the naturalness of the ZODB is astonishing.

Relational databases (RDBs) are far more common than OODBs. Relational databases store information in tables; a table consists of any number of rows, each row containing several columns of information. (Rows are more formally called relations, which is where the term "relational database" originates.)

Let's look at a concrete example. The example comes from my day job working for the MEMS Exchange, in a greatly simplified version. The job is to track process runs, which are lists of manufacturing steps to be performed in a semiconductor fab. A run is owned by a particular user, and has a name and assigned ID number. Runs consist of a number of operations; an operation is a single step to be performed, such as depositing something on a wafer or etching something off it.

Operations may have parameters, which are additional information required to perform an operation. For example, if you're depositing something on a wafer, you need to know two things: 1) what you're depositing, and 2) how much should be deposited. You might deposit 100 microns of silicon oxide, or 1 micron of copper.

Mapping these structures to a relational database is straightforward:

```
CREATE TABLE runs (
  int      run_id,
  varchar  owner,
  varchar  title,
  int      acct_num,
  primary key(run_id)
);

CREATE TABLE operations (
  int      run_id,
  int      step_num,
  varchar  process_id,
  PRIMARY KEY(run_id, step_num),
  FOREIGN KEY(run_id) REFERENCES runs(run_id),
);

CREATE TABLE parameters (
  int      run_id,
  int      step_num,
  varchar  param_name,
  varchar  param_value,
  PRIMARY KEY(run_id, step_num, param_name)
  FOREIGN KEY(run_id, step_num)
     REFERENCES operations(run_id, step_num),
);
```

In Python, you would write three classes named `Run`, `Operation`, and `Parameter`. I won't present code for defining these classes, since that code is uninteresting at this point. Each class would contain a single method to begin with, an `__init__()` method that assigns default values, such as 0 or `None`, to each attribute of the class.

It's not difficult to write Python code that will create a `Run` instance and populate it with the data from the relational tables; with a little more effort, you can build a straightforward tool, usually called an object- relational mapper, to do this automatically. (See http://www.amk.ca/python/unmaintained/ordb.html for a quick hack at a Python object-relational mapper, and http://www.python.org/workshops/1997-10/proceedings/shprentz.html for Joel Shprentz's more successful implementation of the same idea; Unlike mine, Shprentz's system has been used for actual work.)

However, it is difficult to make an object-relational mapper reasonably quick; a simple-minded implementation like mine is quite slow because it has to do several queries to access all of an object's data. Higher performance object-relational mappers cache objects to improve performance, only performing SQL queries when they actually need to.

That helps if you want to access run number 123 all of a sudden. But what if you want to find all runs where a step has a parameter named 'thickness' with a value of 2.0? In the relational version, you have two unappealing choices:

1. Write a specialized SQL query for this case: `SELECT run_id FROM operations WHERE param_name = 'thickness' AND param_value = 2.0`

   If such queries are common, you can end up with lots of specialized queries. When the database tables get rearranged, all these queries will need to be modified.

2. An object-relational mapper doesn't help much. Scanning through the runs means that the the mapper will perform the required SQL queries to read run #1, and then a simple Python loop can check whether any of its steps have the parameter you're looking for. Repeat for run #2, 3, and so forth. This does a vast number of SQL queries, and therefore is incredibly slow.

An object database such as ZODB simply stores internal pointers from object to object, so reading in a single object is much faster than doing a bunch of SQL queries and assembling the results. Scanning all runs, therefore, is still inefficient, but not grossly inefficient.

What is ZEO?

The ZODB comes with a few different classes that implement the `Storage` interface. Such classes handle the job of writing out Python objects to a physical storage medium, which can be a disk file (the `FileStorage` class), a BerkeleyDB file (`BDBFullStorage`), a relational database (`DCOracleStorage`), or some other medium. ZEO adds `ClientStorage`, a new `Storage` that doesn't write to physical media but just forwards all requests across a network to a server. The server, which is running an instance of the `StorageServer` class, simply acts as a front-end for some physical `Storage` class. It's a fairly simple idea, but as we'll see later on in this document, it opens up many possibilities.

About this guide

The primary author of this guide works on a project which uses the ZODB and ZEO as its primary storage technology. We use the ZODB to store process runs and operations, a catalog of available processes, user information, accounting information, and other data. Part of the goal of writing this document is to make our experience more widely available. A few times we've spent hours or even days trying to figure out a problem, and this guide is an attempt to gather up the knowledge we've gained so that others don't have to make the same mistakes we did while learning.

The author's ZODB project is described in a paper available here, http://www.amk.ca/python/writing/mx-architecture/

This document will always be a work in progress. If you wish to suggest clarifications or additional topics, please send your comments to the ZODB-dev mailing list.

Acknowledgements

Andrew Kuchling wrote the original version of this guide, which provided some of the first ZODB documentation for Python programmers. His initial version has been updated over time by Jeremy Hylton and Tim Peters.

I'd like to thank the people who've pointed out inaccuracies and bugs, offered suggestions on the text, or proposed new topics that should be covered: Jeff Bauer, Willem Broekema, Thomas Guettler, Chris McDonough, George Runyan.

ZODB Programming

Installing ZODB

ZODB is packaged using the standard distutils tools.

Requirements

You will need Python 2.3 or higher. Since the code is packaged using distutils, it is simply a matter of untarring or unzipping the release package, and then running `python setup.py install`.

You'll need a C compiler to build the packages, because there are various C extension modules. Binary installers are provided for Windows users.

Installing the Packages

Download the ZODB tarball containing all the packages for both ZODB and ZEO from http://www.zope.org/Products/ZODB3.3. See the `README.txt` file in the top level of the release directory for details on building, testing, and installing.

You can find information about ZODB and the most current releases in the ZODB Wiki at http://www.zope.org/Wikis/ZODB.

How ZODB Works

The ZODB is conceptually simple. Python classes subclass a `persistent.Persistent` class to become ZODB-aware. Instances of persistent objects are brought in from a permanent storage medium, such as a disk file, when the program needs them, and remain cached in RAM. The ZODB traps modifications to objects, so that when a statement such as `obj.size = 1` is executed, the modified object is marked as "dirty." On request, any dirty objects are written out to permanent storage; this is called committing a transaction. Transactions can also be aborted or rolled back, which results in any changes being discarded, dirty objects reverting to their initial state before the transaction began.

The term "transaction" has a specific technical meaning in computer science. It's extremely important that the contents of a database don't get corrupted by software or hardware crashes, and most database software offers protection against such corruption by supporting four useful properties, Atomicity, Consistency, Isolation, and Durability. In computer science jargon these four terms are collectively dubbed the ACID properties, forming an acronym from their names.

The ZODB provides all of the ACID properties. Definitions of the ACID properties are:

Atomicity means that any changes to data made during a transaction are all-or-nothing. Either all the changes are applied, or none of them are. If a program makes a bunch of modifications and then crashes, the database won't be partially modified, potentially leaving the data in an inconsistent state; instead all the changes will be forgotten. That's bad, but it's better than having a partially- applied modification put the database into an inconsistent state.

Consistency means that each transaction executes a valid transformation of the database state. Some databases, but not ZODB, provide a variety of consistency checks in the database or language; for example, a relational database constraint columns to be of particular types and can enforce relations across tables. Viewed more generally, atomicity and isolation make it possible for applications to provide consistency.

Isolation means that two programs or threads running in two different transactions cannot see each other's changes until they commit their transactions.

Durability means that once a transaction has been committed, a subsequent crash will not cause any data to be lost or corrupted.

Opening a ZODB

There are 3 main interfaces supplied by the ZODB: `Storage`, *DB*, and `Connection` classes. The *DB* and `Connection` interfaces both have single implementations, but there are several different classes that implement the `Storage` interface.

- `Storage` classes are the lowest layer, and handle storing and retrieving objects from some form of long-term storage. A few different types of Storage have been written, such as `FileStorage`, which uses regular disk files, and `BDBFullStorage`, which uses Sleepycat Software's BerkeleyDB database. You could write a new

Storage that stored objects in a relational database, for example, if that would better suit your application. Two example storages, `DemoStorage` and `MappingStorage`, are available to use as models if you want to write a new Storage.

- The *DB* class sits on top of a storage, and mediates the interaction between several connections. One *DB* instance is created per process.
- Finally, the `Connection` class caches objects, and moves them into and out of object storage. A multi-threaded program should open a separate `Connection` instance for each thread. Different threads can then modify objects and commit their modifications independently.

Preparing to use a ZODB requires 3 steps: you have to open the `Storage`, then create a *DB* instance that uses the `Storage`, and then get a `Connection` from the `DB instance`. All this is only a few lines of code:

```
from ZODB import FileStorage, DB

storage = FileStorage.FileStorage('/tmp/test-filestorage.fs')
db = DB(storage)
conn = db.open()
```

Note that you can use a completely different data storage mechanism by changing the first line that opens a `Storage`; the above example uses a `FileStorage`. In section *ZEO*, "How ZEO Works", you'll see how ZEO uses this flexibility to good effect.

Using a ZODB Configuration File

ZODB also supports configuration files written in the ZConfig format. A configuration file can be used to separate the configuration logic from the application logic. The storages classes and the *DB* class support a variety of keyword arguments; all these options can be specified in a config file.

The configuration file is simple. The example in the previous section could use the following example:

```
<zodb>
  <filestorage>
  path /tmp/test-filestorage.fs
  </filestorage>
</zodb>
```

The *ZODB.config* module includes several functions for opening database and storages from configuration files.

```
import ZODB.config

db = ZODB.config.databaseFromURL('/tmp/test.conf')
conn = db.open()
```

The ZConfig documentation, included in the ZODB3 release, explains the format in detail. Each configuration file is described by a schema, by convention stored in a `component.xml` file. ZODB, ZEO, zLOG, and zdaemon all have schemas.

Writing a Persistent Class

Making a Python class persistent is quite simple; it simply needs to subclass from the `Persistent` class, as shown in this example:

```
from persistent import Persistent

class User(Persistent):
    pass
```

The `Persistent` base class is a new-style class implemented in C.

For simplicity, in the examples the `User` class will simply be used as a holder for a bunch of attributes. Normally the class would define various methods that add functionality, but that has no impact on the ZODB's treatment of the class.

The ZODB uses persistence by reachability; starting from a set of root objects, all the attributes of those objects are made persistent, whether they're simple Python data types or class instances. There's no method to explicitly store objects in a ZODB database; simply assign them as an attribute of an object, or store them in a mapping, that's already in the database. This chain of containment must eventually reach back to the root object of the database.

As an example, we'll create a simple database of users that allows retrieving a `User` object given the user's ID. First, we retrieve the primary root object of the ZODB using the `root()` method of the `Connection` instance. The root object behaves like a Python dictionary, so you can just add a new key/value pair for your application's root object. We'll insert an `OOBTree` object that will contain all the `User` objects. (The `BTree` module is also included as part of Zope.)

```
dbroot = conn.root()

# Ensure that a 'userdb' key is present
# in the root
if not dbroot.has_key('userdb'):
    from BTrees.OOBTree import OOBTree
    dbroot['userdb'] = OOBTree()

userdb = dbroot['userdb']
```

Inserting a new user is simple: create the `User` object, fill it with data, insert it into the `BTree` instance, and commit this transaction.

```
# Create new User instance
import transaction

newuser = User()

# Add whatever attributes you want to track
newuser.id = 'amk'
newuser.first_name = 'Andrew' ; newuser.last_name = 'Kuchling'
...

# Add object to the BTree, keyed on the ID
userdb[newuser.id] = newuser

# Commit the change
transaction.commit()
```

The `transaction` module defines a few top-level functions for working with transactions. `commit()` writes any modified objects to disk, making the changes permanent. `abort()` rolls back any changes that have been made, restoring the original state of the objects. If you're familiar with database transactional semantics, this is all what you'd expect. `get()` returns a `Transaction` object that has additional methods like `note()`, to add a note to the transaction metadata.

More precisely, the `transaction` module exposes an instance of the `ThreadTransactionManager` transac-

tion manager class as `transaction.manager`, and the `transaction` functions `get()` and `begin()` redirect to the same-named methods of `transaction.manager`. The `commit()` and `abort()` functions apply the methods of the same names to the `Transaction` object returned by `transaction.manager.get()`. This is for convenience. It's also possible to create your own transaction manager instances, and to tell `DB.open()` to use your transaction manager instead.

Because the integration with Python is so complete, it's a lot like having transactional semantics for your program's variables, and you can experiment with transactions at the Python interpreter's prompt:

```
>>> newuser
<User instance at 81b1f40>
>>> newuser.first_name           # Print initial value
'Andrew'
>>> newuser.first_name = 'Bob'   # Change first name
>>> newuser.first_name           # Verify the change
'Bob'
>>> transaction.abort()          # Abort transaction
>>> newuser.first_name           # The value has changed back
'Andrew'
```

Rules for Writing Persistent Classes

Practically all persistent languages impose some restrictions on programming style, warning against constructs they can't handle or adding subtle semantic changes, and the ZODB is no exception. Happily, the ZODB's restrictions are fairly simple to understand, and in practice it isn't too painful to work around them.

The summary of rules is as follows:

- If you modify a mutable object that's the value of an object's attribute, the ZODB can't catch that, and won't mark the object as dirty. The solution is to either set the dirty bit yourself when you modify mutable objects, or use a wrapper for Python's lists and dictionaries (*PersistentList*, *PersistentMapping*) that will set the dirty bit properly.
- Recent versions of the ZODB allow writing a class with `__setattr__()` , `__getattr__()`, or `__delattr__()` methods. (Older versions didn't support this at all.) If you write such a `__setattr__()` or `__delattr__()` method, its code has to set the dirty bit manually.
- A persistent class should not have a `__del__()` method. The database moves objects freely between memory and storage. If an object has not been used in a while, it may be released and its contents loaded from storage the next time it is used. Since the Python interpreter is unaware of persistence, it would call `__del__()` each time the object was freed.

Let's look at each of these rules in detail.

Modifying Mutable Objects

The ZODB uses various Python hooks to catch attribute accesses, and can trap most of the ways of modifying an object, but not all of them. If you modify a `User` object by assigning to one of its attributes, as in `userobj.first_name = 'Andrew'`, the ZODB will mark the object as having been changed, and it'll be written out on the following `commit()`.

The most common idiom that *isn't* caught by the ZODB is mutating a list or dictionary. If `User` objects have a attribute named `friends` containing a list, calling `userobj.friends.append(otherUser)` doesn't mark `userobj` as modified; from the ZODB's point of view, `userobj.friends` was only read, and its value, which happened to be an ordinary Python list, was returned. The ZODB isn't aware that the object returned was subsequently modified.

This is one of the few quirks you'll have to remember when using the ZODB; if you modify a mutable attribute of an object in place, you have to manually mark the object as having been modified by setting its dirty bit to true. This is done by setting the `_p_changed` attribute of the object to true:

```
userobj.friends.append(otherUser)
userobj._p_changed = True
```

You can hide the implementation detail of having to mark objects as dirty by designing your class's API to not use direct attribute access; instead, you can use the Java-style approach of accessor methods for everything, and then set the dirty bit within the accessor method. For example, you might forbid accessing the `friends` attribute directly, and add a `get_friend_list()` accessor and an `add_friend()` modifier method to the class. `add_friend()` would then look like this:

```
def add_friend(self, friend):
    self.friends.append(otherUser)
    self._p_changed = True
```

Alternatively, you could use a ZODB-aware list or mapping type that handles the dirty bit for you. The ZODB comes with a *PersistentMapping* class, and I've contributed a *PersistentList* class that's included in my ZODB distribution, and may make it into a future upstream release of Zope.

`__getattr__()`, `__delattr__()`, and `__setattr__()`

ZODB allows persistent classes to have hook methods like `__getattr__()` and `__setattr__()`. There are four special methods that control attribute access; the rules for each are a little different.

The `__getattr__()` method works pretty much the same for persistent classes as it does for other classes. No special handling is needed. If an object is a ghost, then it will be activated before `__getattr__()` is called.

The other methods are more delicate. They will override the hooks provided by `Persistent`, so user code must call special methods to invoke those hooks anyway.

The `__getattribute__()` method will be called for all attribute access; it overrides the attribute access support inherited from `Persistent`. A user-defined `__getattribute__()` must always give the `Persistent` base class a chance to handle special attribute, as well as `__dict__` or `__class__`. The user code should call `_p_getattr()`, passing the name of the attribute as the only argument. If it returns True, the user code should call `Persistent`'s `__getattribute__()` to get the value. If not, the custom user code can run.

A `__setattr__()` hook will also override the `Persistent` `__setattr__()` hook. User code must treat it much like `__getattribute__()`. The user-defined code must call `_p_setattr()` first to all `Persistent` to handle special attributes; `_p_setattr()` takes the attribute name and value. If it returns True, `Persistent` handled the attribute. If not, the user code can run. If the user code modifies the object's state, it must assigned to `_p_changed`.

A `__delattr__()` hooks must be implemented the same was as a the last two hooks. The user code must call `_p_delattr()`, passing the name of the attribute as an argument. If the call returns True, `Persistent` handled the attribute; if not, the user code can run.

`__del__()` methods

A `__del__()` method is invoked just before the memory occupied by an unreferenced Python object is freed. Because ZODB may materialize, and dematerialize, a given persistent object in memory any number of times, there isn't a meaningful relationship between when a persistent object's `__del__()` method gets invoked and any natural aspect of a persistent object's life cycle. For example, it is emphatically not the case that a persistent object's `__del__()`

method gets invoked only when the object is no longer referenced by other objects in the database. `__del__()` is only concerned with reachability from objects in memory.

Worse, a `__del__()` method can interfere with the persistence machinery's goals. For example, some number of persistent objects reside in a `Connection`'s memory cache. At various times, to reduce memory burden, objects that haven't been referenced recently are removed from the cache. If a persistent object with a `__del__()` method is so removed, and the cache was holding the last memory reference to the object, the object's `__del__()` method will be invoked. If the `__del__()` method then references any attribute of the object, ZODB needs to load the object from the database again, in order to satisfy the attribute reference. This puts the object back into the cache again: such an object is effectively immortal, occupying space in the memory cache forever, as every attempt to remove it from cache puts it back into the cache. In ZODB versions prior to 3.2.2, this could even cause the cache reduction code to fall into an infinite loop. The infinite loop no longer occurs, but such objects continue to live in the memory cache forever.

Because `__del__()` methods don't make good sense for persistent objects, and can create problems, persistent classes should not define `__del__()` methods.

Writing Persistent Classes

Now that we've looked at the basics of programming using the ZODB, we'll turn to some more subtle tasks that are likely to come up for anyone using the ZODB in a production system.

Changing Instance Attributes

Ideally, before making a class persistent you would get its interface right the first time, so that no attributes would ever need to be added, removed, or have their interpretation change over time. It's a worthy goal, but also an impractical one unless you're gifted with perfect knowledge of the future. Such unnatural foresight can't be required of any person, so you therefore have to be prepared to handle such structural changes gracefully. In object-oriented database terminology, this is a schema update. The ZODB doesn't have an actual schema specification, but you're changing the software's expectations of the data contained by an object, so you're implicitly changing the schema.

One way to handle such a change is to write a one-time conversion program that will loop over every single object in the database and update them to match the new schema. This can be easy if your network of object references is quite structured, making it easy to find all the instances of the class being modified. For example, if all `User` objects can be found inside a single dictionary or BTree, then it would be a simple matter to loop over every `User` instance with a `for` statement. This is more difficult if your object graph is less structured; if `User` objects can be found as attributes of any number of different class instances, then there's no longer any easy way to find them all, short of writing a generalized object traversal function that would walk over every single object in a ZODB, checking each one to see if it's an instance of `User`.

Some OODBs support a feature called extents, which allow quickly finding all the instances of a given class, no matter where they are in the object graph; unfortunately the ZODB doesn't offer extents as a feature.

ZEO

How ZEO Works

The ZODB, as I've described it so far, can only be used within a single Python process (though perhaps with multiple threads). ZEO, Zope Enterprise Objects, extends the ZODB machinery to provide access to objects over a network. The name "Zope Enterprise Objects" is a bit misleading; ZEO can be used to store Python objects and access them in a distributed fashion without Zope ever entering the picture. The combination of ZEO and ZODB is essentially a Python- specific object database.

ZEO consists of about 12,000 lines of Python code, excluding tests. The code is relatively small because it contains only code for a TCP/IP server, and for a new type of Storage, `ClientStorage`. `ClientStorage` simply makes remote procedure calls to the server, which then passes them on a regular `Storage` class such as `FileStorage`. The following diagram lays out the system:

XXX insert diagram here later

Any number of processes can create a `ClientStorage` instance, and any number of threads in each process can be using that instance. `ClientStorage` aggressively caches objects locally, so in order to avoid using stale data the ZEO server sends an invalidation message to all the connected `ClientStorage` instances on every write operation. The invalidation message contains the object ID for each object that's been modified, letting the `ClientStorage` instances delete the old data for the given object from their caches.

This design decision has some consequences you should be aware of. First, while ZEO isn't tied to Zope, it was first written for use with Zope, which stores HTML, images, and program code in the database. As a result, reads from the database are *far* more frequent than writes, and ZEO is therefore better suited for read-intensive applications. If every `ClientStorage` is writing to the database all the time, this will result in a storm of invalidate messages being sent, and this might take up more processing time than the actual database operations themselves. These messages are small and sent in batches, so there would need to be a lot of writes before it became a problem.

On the other hand, for applications that have few writes in comparison to the number of read accesses, this aggressive caching can be a major win. Consider a Slashdot-like discussion forum that divides the load among several Web servers. If news items and postings are represented by objects and accessed through ZEO, then the most heavily accessed objects – the most recent or most popular postings – will very quickly wind up in the caches of the `ClientStorage` instances on the front-end servers. The back-end ZEO server will do relatively little work, only being called upon to return the occasional older posting that's requested, and to send the occasional invalidate message when a new posting is added. The ZEO server isn't going to be contacted for every single request, so its workload will remain manageable.

Installing ZEO

This section covers how to install the ZEO package, and how to configure and run a ZEO Storage Server on a machine.

Requirements

The ZEO server software is included in ZODB3. As with the rest of ZODB3, you'll need Python 2.3 or higher.

Running a server

The runzeo.py script in the ZEO directory can be used to start a server. Run it with the -h option to see the various values. If you're just experimenting, a good choise is to use `python ZEO/runzeo.py -a /tmp/zeosocket -f /tmp/test.fs` to run ZEO with a Unix domain socket and a `FileStorage`.

Testing the ZEO Installation

Once a ZEO server is up and running, using it is just like using ZODB with a more conventional disk-based storage; no new programming details are introduced by using a remote server. The only difference is that programs must create a `ClientStorage` instance instead of a `FileStorage` instance. From that point onward, ZODB-based code is happily unaware that objects are being retrieved from a ZEO server, and not from the local disk.

As an example, and to test whether ZEO is working correctly, try running the following lines of code, which will connect to the server, add some bits of data to the root of the ZODB, and commits the transaction:

```
from ZEO import ClientStorage
from ZODB import DB
import transaction

# Change next line to connect to your ZEO server
addr = 'kronos.example.com', 1975
storage = ClientStorage.ClientStorage(addr)
db = DB(storage)
conn = db.open()
root = conn.root()

# Store some things in the root
root['list'] = ['a', 'b', 1.0, 3]
root['dict'] = {'a':1, 'b':4}

# Commit the transaction
transaction.commit()
```

If this code runs properly, then your ZEO server is working correctly.

You can also use a configuration file.

```
<zodb>
    <zeoclient>
    server localhost:9100
    </zeoclient>
</zodb>
```

One nice feature of the configuration file is that you don't need to specify imports for a specific storage. That makes the code a little shorter and allows you to change storages without changing the code.

```
import ZODB.config

db = ZODB.config.databaseFromURL('/tmp/zeo.conf')
```

ZEO Programming Notes

ZEO is written using `asyncore`, from the Python standard library. It assumes that some part of the user application is running an `asyncore` mainloop. For example, Zope run the loop in a separate thread and ZEO uses that. If your application does not have a mainloop, ZEO will not process incoming invalidation messages until you make some call into ZEO. The `Connection.sync()` method can be used to process pending invalidation messages. You can call it when you want to make sure the `Connection` has the most recent version of every object, but you don't have any other work for ZEO to do.

Sample Application: chatter.py

For an example application, we'll build a little chat application. What's interesting is that none of the application's code deals with network programming at all; instead, an object will hold chat messages, and be magically shared between all the clients through ZEO. I won't present the complete script here; you can download it from `chatter.py`. Only the interesting portions of the code will be covered here.

The basic data structure is the `ChatSession` object, which provides an `add_message()` method that adds a message, and a `new_messages()` method that returns a list of new messages that have accumulated since the last call to `new_messages()`. Internally, `ChatSession` maintains a B-tree that uses the time as the key, and stores the message as the corresponding value.

The constructor for `ChatSession` is pretty simple; it simply creates an attribute containing a B-tree:

```
class ChatSession(Persistent):
    def __init__(self, name):
        self.name = name
        # Internal attribute: _messages holds all the chat messages.
        self._messages = BTrees.OOBTree.OOBTree()
```

`add_message()` has to add a message to the `_messages` B-tree. A complication is that it's possible that some other client is trying to add a message at the same time; when this happens, the client that commits first wins, and the second client will get a `ConflictError` exception when it tries to commit. For this application, `ConflictError` isn't serious but simply means that the operation has to be retried; other applications might treat it as a fatal error. The code uses `try...except...else` inside a `while` loop, breaking out of the loop when the commit works without raising an exception.

```
def add_message(self, message):
    """Add a message to the channel.
    message -- text of the message to be added
    """

    while 1:
        try:
            now = time.time()
            self._messages[now] = message
            get_transaction().commit()
        except ConflictError:
            # Conflict occurred; this process should abort,
            # wait for a little bit, then try again.
            transaction.abort()
            time.sleep(.2)
        else:
            # No ConflictError exception raised, so break
            # out of the enclosing while loop.
            break
    # end while
```

`new_messages()` introduces the use of *volatile* attributes. Attributes of a persistent object that begin with `_v_` are considered volatile and are never stored in the database. `new_messages()` needs to store the last time the method was called, but if the time was stored as a regular attribute, its value would be committed to the database and shared with all the other clients. `new_messages()` would then return the new messages accumulated since any other client called `new_messages()`, which isn't what we want.

```
def new_messages(self):
    "Return new messages."

    # self._v_last_time is the time of the most recent message
    # returned to the user of this class.
    if not hasattr(self, '_v_last_time'):
        self._v_last_time = 0

    new = []
    T = self._v_last_time

    for T2, message in self._messages.items():
        if T2 > T:
            new.append(message)
            self._v_last_time = T2
```

```
    return new
```

This application is interesting because it uses ZEO to easily share a data structure; ZEO and ZODB are being used for their networking ability, not primarily for their data storage ability. I can foresee many interesting applications using ZEO in this way:

- With a Tkinter front-end, and a cleverer, more scalable data structure, you could build a shared whiteboard using the same technique.
- A shared chessboard object would make writing a networked chess game easy.
- You could create a Python class containing a CD's title and track information. To make a CD database, a read-only ZEO server could be opened to the world, or an HTTP or XML-RPC interface could be written on top of the ZODB.
- A program like Quicken could use a ZODB on the local disk to store its data. This avoids the need to write and maintain specialized I/O code that reads in your objects and writes them out; instead you can concentrate on the problem domain, writing objects that represent cheques, stock portfolios, or whatever.

Transactions and Versioning

Committing and Aborting

Changes made during a transaction don't appear in the database until the transaction commits. This is done by calling the `commit()` method of the current `Transaction` object, where the latter is obtained from the `get()` method of the current transaction manager. If the default thread transaction manager is being used, then `transaction.commit()` suffices.

Similarly, a transaction can be explicitly aborted (all changes within the transaction thrown away) by invoking the `abort()` method of the current `Transaction` object, or simply `transaction.abort()` if using the default thread transaction manager.

Prior to ZODB 3.3, if a commit failed (meaning the `commit()` call raised an exception), the transaction was implicitly aborted and a new transaction was implicitly started. This could be very surprising if the exception was suppressed, and especially if the failing commit was one in a sequence of subtransaction commits.

So, starting with ZODB 3.3, if a commit fails, all further attempts to commit, join, or register with the transaction raise `ZODB.POSException.TransactionFailedError`. You must explicitly start a new transaction then, either by calling the `abort()` method of the current transaction, or by calling the `begin()` method of the current transaction's transaction manager.

Subtransactions

Subtransactions can be created within a transaction. Each subtransaction can be individually committed and aborted, but the changes within a subtransaction are not truly committed until the containing transaction is committed.

The primary purpose of subtransactions is to decrease the memory usage of transactions that touch a very large number of objects. Consider a transaction during which 200,000 objects are modified. All the objects that are modified in a single transaction have to remain in memory until the transaction is committed, because the ZODB can't discard them from the object cache. This can potentially make the memory usage quite large. With subtransactions, a commit can be be performed at intervals, say, every 10,000 objects. Those 10,000 objects are then written to permanent storage and can be purged from the cache to free more space.

To commit a subtransaction instead of a full transaction, pass a true value to the `commit()` or `abort()` method of the `Transaction` object.

```
# Commit a subtransaction
transaction.commit(True)

# Abort a subtransaction
transaction.abort(True)
```

A new subtransaction is automatically started upon successful committing or aborting the previous subtransaction.

Undoing Changes

Some types of `Storage` support undoing a transaction even after it's been committed. You can tell if this is the case by calling the `supportsUndo()` method of the *DB* instance, which returns true if the underlying storage supports undo. Alternatively you can call the `supportsUndo()` method on the underlying storage instance.

If a database supports undo, then the `undoLog(start, end[, func])()` method on the *DB* instance returns the log of past transactions, returning transactions between the times *start* and *end*, measured in seconds from the epoch. If present, *func* is a function that acts as a filter on the transactions to be returned; it's passed a dictionary representing each transaction, and only transactions for which *func* returns true will be included in the list of transactions returned to the caller of `undoLog()`. The dictionary contains keys for various properties of the transaction. The most important keys are `id`, for the transaction ID, and `time`, for the time at which the transaction was committed.

```
>>> print storage.undoLog(0, sys.maxint)
[{'description': '',
  'id': 'AzpGEGqU/0QAAAAAAAAGMA',
  'time': 981126744.98,
  'user_name': ''},
 {'description': '',
  'id': 'AzpGC/hUOKoAAAAAAAAFDQ',
  'time': 981126478.202,
  'user_name': ''}
  ...
```

To store a description and a user name on a commit, get the current transaction and call the `note(text)()` method to store a description, and the `setUser(user_name)()` method to store the user name. While `setUser()` overwrites the current user name and replaces it with the new value, the `note()` method always adds the text to the transaction's description, so it can be called several times to log several different changes made in the course of a single transaction.

```
transaction.get().setUser('amk')
transaction.get().note('Change ownership')
```

To undo a transaction, call the `DB.undo(id)()` method, passing it the ID of the transaction to undo. If the transaction can't be undone, a `ZODB.POSException.UndoError` exception will be raised, with the message "non-undoable transaction". Usually this will happen because later transactions modified the objects affected by the transaction you're trying to undo.

After you call `undo()` you must commit the transaction for the undo to actually be applied.[1] There is one glitch in the undo process. The thread that calls undo may not see the changes to the object until it calls `Connection.sync()` or commits another transaction.

[1] There are actually two different ways a storage can implement the undo feature. Most of the storages that ship with ZODB use the transactional form of undo described in the main text. Some storages may use a non-transactional undo makes changes visible immediately.

Versions

Warning: Versions should be avoided. They're going to be deprecated, replaced by better approaches to long-running transactions.

While many subtransactions can be contained within a single regular transaction, it's also possible to contain many regular transactions within a long-running transaction, called a version in ZODB terminology. Inside a version, any number of transactions can be created and committed or rolled back, but the changes within a version are not made visible to other connections to the same ZODB.

Not all storages support versions, but you can test for versioning ability by calling `supportsVersions()` method of the *DB* instance, which returns true if the underlying storage supports versioning.

A version can be selected when creating the `Connection` instance using the `DB.open([*version*])()` method. The *version* argument must be a string that will be used as the name of the version.

```
vers_conn = db.open(version='Working version')
```

Transactions can then be committed and aborted using this versioned connection. Other connections that don't specify a version, or provide a different version name, will not see changes committed within the version named `Working version`. To commit or abort a version, which will either make the changes visible to all clients or roll them back, call the `DB.commitVersion()` or `DB.abortVersion()` methods. XXX what are the source and dest arguments for?

The ZODB makes no attempt to reconcile changes between different versions. Instead, the first version which modifies an object will gain a lock on that object. Attempting to modify the object from a different version or from an unversioned connection will cause a `ZODB.POSException.VersionLockError` to be raised:

```
from ZODB.POSException import VersionLockError

try:
    transaction.commit()
except VersionLockError, (obj_id, version):
    print ('Cannot commit; object %s '
           'locked by version %s' % (obj_id, version))
```

The exception provides the ID of the locked object, and the name of the version having a lock on it.

Multithreaded ZODB Programs

ZODB databases can be accessed from multithreaded Python programs. The `Storage` and *DB* instances can be shared among several threads, as long as individual `Connection` instances are created for each thread.

Related Modules

The ZODB package includes a number of related modules that provide useful data types such as BTrees.

persistent.mapping.PersistentMapping

The *PersistentMapping* class is a wrapper for mapping objects that will set the dirty bit when the mapping is modified by setting or deleting a key.

PersistentMapping(*container = {}*)
Create a *PersistentMapping* object that wraps the mapping object *container*. If you don't specify a value for *container*, a regular Python dictionary is used.

PersistentMapping objects support all the same methods as Python dictionaries do.

persistent.list.PersistentList

The *PersistentList* class is a wrapper for mutable sequence objects, much as *PersistentMapping* is a wrapper for mappings.

PersistentList(*initlist = []*)
Create a *PersistentList* object that wraps the mutable sequence object *initlist*. If you don't specify a value for *initlist*, a regular Python list is used.

PersistentList objects support all the same methods as Python lists do.

BTrees Package

When programming with the ZODB, Python dictionaries aren't always what you need. The most important case is where you want to store a very large mapping. When a Python dictionary is accessed in a ZODB, the whole dictionary has to be unpickled and brought into memory. If you're storing something very large, such as a 100,000-entry user database, unpickling such a large object will be slow. BTrees are a balanced tree data structure that behave like a mapping but distribute keys throughout a number of tree nodes. The nodes are stored in sorted order (this has important consequences – see below). Nodes are then only unpickled and brought into memory as they're accessed, so the entire tree doesn't have to occupy memory (unless you really are touching every single key).

The BTrees package provides a large collection of related data structures. There are variants of the data structures specialized to integers, which are faster and use less memory. There are five modules that handle the different variants. The first two letters of the module name specify the types of the keys and values in mappings – O for any object, I for 32-bit signed integer, and (new in ZODB 3.4) F for 32-bit C float. For example, the `BTrees.IOBTree` module provides a mapping with integer keys and arbitrary objects as values.

The four data structures provide by each module are a BTree, a Bucket, a TreeSet, and a Set. The BTree and Bucket types are mappings and support all the usual mapping methods, e.g. `update()` and `keys()`. The TreeSet and Set types are similar to mappings but they have no values; they support the methods that make sense for a mapping with no keys, e.g. `keys()` but not `items()`. The Bucket and Set types are the individual building blocks for BTrees and TreeSets, respectively. A Bucket or Set can be used when you are sure that it will have few elements. If the data structure will grow large, you should use a BTree or TreeSet. Like Python lists, Buckets and Sets are allocated in one contiguous piece, and insertions and deletions can take time proportional to the number of existing elements. Also like Python lists, a Bucket or Set is a single object, and is pickled and unpickled in its entirety. BTrees and TreeSets are multi-level tree structures with much better (logarithmic) worst- case time bounds, and the tree structure is built out of multiple objects, which ZODB can load individually as needed.

The five modules are named `OOBTree`, `IOBTree`, `OIBTree`, `IIBTree`, and (new in ZODB 3.4) `IFBTree`. The two letter prefixes are repeated in the data types names. The `BTrees.OOBTree` module defines the following types: `OOBTree`, `OOBucket`, `OOSet`, and `OOTreeSet`. Similarly, the other four modules each define their own variants of those four types.

The `keys()`, `values()`, and `items()` methods on BTree and TreeSet types do not materialize a list with all of the data. Instead, they return lazy sequences that fetch data from the BTree as needed. They also support optional arguments to specify the minimum and maximum values to return, often called "range searching". Because all these types are stored in sorted order, range searching is very efficient.

The `keys()`, `values()`, and `items()` methods on Bucket and Set types do return lists with all the data. Starting in ZODB 3.3, there are also `iterkeys()`, `itervalues()`, and `iteritems()` methods that return iterators (in the Python 2.2 sense). Those methods also apply to BTree and TreeSet objects.

A BTree object supports all the methods you would expect of a mapping, with a few extensions that exploit the fact that the keys are sorted. The example below demonstrates how some of the methods work. The extra methods are `minKey()` and `maxKey()`, which find the minimum and maximum key value subject to an optional bound argument, and `byValue()`, which should probably be ignored (it's hard to explain exactly what it does, and as a result it's almost never used – best to consider it deprecated). The various methods for enumerating keys, values and items also accept minimum and maximum key arguments ("range search"), and (new in ZODB 3.3) optional Boolean arguments to control whether a range search is inclusive or exclusive of the range's endpoints.

```
>>> from BTrees.OOBTree import OOBTree
>>> t = OOBTree()
>>> t.update({1: "red", 2: "green", 3: "blue", 4: "spades"})
>>> len(t)
4
>>> t[2]
'green'
>>> s = t.keys() # this is a "lazy" sequence object
>>> s
<OOBTreeItems object at 0x0088AD20>
>>> len(s)  # it acts like a Python list
4
>>> s[-2]
3
>>> list(s) # materialize the full list
[1, 2, 3, 4]
>>> list(t.values())
['red', 'green', 'blue', 'spades']
>>> list(t.values(1, 2)) # values at keys in 1 to 2 inclusive
['red', 'green']
>>> list(t.values(2))    # values at keys >= 2
['green', 'blue', 'spades']
>>> list(t.values(min=1, max=4))  # keyword args new in ZODB 3.3
['red', 'green', 'blue', 'spades']
>>> list(t.values(min=1, max=4, excludemin=True, excludemax=True))
['green', 'blue']
>>> t.minKey()     # smallest key
1
>>> t.minKey(1.5)  # smallest key >= 1.5
2
>>> for k in t.keys():
...     print k,
1 2 3 4
>>> for k in t:    # new in ZODB 3.3
...     print k,
1 2 3 4
>>> for pair in t.iteritems():  # new in ZODB 3.3
...     print pair,
...
(1, 'red') (2, 'green') (3, 'blue') (4, 'spades')
>>> t.has_key(4)  # returns a true value, but exactly what undefined
2
>>> t.has_key(5)
0
>>> 4 in t  # new in ZODB 3.3
True
```

```
>>> 5 in t  # new in ZODB 3.3
False
>>>
```

Each of the modules also defines some functions that operate on BTrees – `difference()`, `union()`, and `intersection()`. The `difference()` function returns a Bucket, while the other two methods return a Set. If the keys are integers, then the module also defines `multiunion()`. If the values are integers or floats, then the module also defines `weightedIntersection()` and `weightedUnion()`. The function doc strings describe each function briefly.

`BTrees/Interfaces.py` defines the operations, and is the official documentation. Note that the interfaces don't define the concrete types returned by most operations, and you shouldn't rely on the concrete types that happen to be returned: stick to operations guaranteed by the interface. In particular, note that the interfaces don't specify anything about comparison behavior, and so nothing about it is guaranteed. In ZODB 3.3, for example, two BTrees happen to use Python's default object comparison, which amounts to comparing the (arbitrary but fixed) memory addresses of the BTrees. This may or may not be true in future releases. If the interfaces don't specify a behavior, then whether that behavior appears to work, and exactly happens if it does appear to work, are undefined and should not be relied on.

Total Ordering and Persistence

The BTree-based data structures differ from Python dicts in several fundamental ways. One of the most important is that while dicts require that keys support hash codes and equality comparison, the BTree-based structures don't use hash codes and require a total ordering on keys.

Total ordering means three things:

1. Reflexive. For each *x*, `x == x` is true.
2. Trichotomy. For each *x* and *y*, exactly one of `x < y`, `x == y`, and `x > y` is true.
3. Transitivity. Whenever `x <= y` and `y <= z`, it's also true that `x <= z`.

The default comparison functions for most objects that come with Python satisfy these rules, with some crucial cautions explained later. Complex numbers are an example of an object whose default comparison function does not satisfy these rules: complex numbers only support == and != comparisons, and raise an exception if you try to compare them in any other way. They don't satisfy the trichotomy rule, and must not be used as keys in BTree-based data structures (although note that complex numbers can be used as keys in Python dicts, which do not require a total ordering).

Examples of objects that are wholly safe to use as keys in BTree-based structures include ints, longs, floats, 8-bit strings, Unicode strings, and tuples composed (possibly recursively) of objects of wholly safe types.

It's important to realize that even if two types satisfy the rules on their own, mixing objects of those types may not. For example, 8-bit strings and Unicode strings both supply total orderings, but mixing the two loses trichotomy; e.g., `'x' < chr(255)` and `u'x' == 'x'`, but trying to compare `chr(255)` to `u'x'` raises an exception. Partly for this reason (another is given later), it can be dangerous to use keys with multiple types in a single BTree-based structure. Don't try to do that, and you don't have to worry about it.

Another potential problem is mutability: when a key is inserted in a BTree- based structure, it must retain the same order relative to the other keys over time. This is easy to run afoul of if you use mutable objects as keys. For example, lists supply a total ordering, and then

```
>>> L1, L2, L3 = [1], [2], [3]
>>> from BTrees.OOBTree import OOSet
>>> s = OOSet((L2, L3, L1))  # this is fine, so far
>>> list(s.keys())           # note that the lists are in sorted order
[[1], [2], [3]]
>>> s.has_key([3])           # and [3] is in the set
```

```
1
>>> L2[0] = 5                # horrible -- the set is insane now
>>> s.has_key([3])           # for example, it's insane this way
0
>>> s
OOSet([[1], [5], [3]])
>>>
```

Key lookup relies on that the keys remain in sorted order (an efficient form of binary search is used). By mutating key L2 after inserting it, we destroyed the invariant that the OOSet is sorted. As a result, all future operations on this set are unpredictable.

A subtler variant of this problem arises due to persistence: by default, Python does several kinds of comparison by comparing the memory addresses of two objects. Because Python never moves an object in memory, this does supply a usable (albeit arbitrary) total ordering across the life of a program run (an object's memory address doesn't change). But if objects compared in this way are used as keys of a BTree-based structure that's stored in a database, when the objects are loaded from the database again they will almost certainly wind up at different memory addresses. There's no guarantee then that if key K1 had a memory address smaller than the memory address of key K2 at the time K1 and K2 were inserted in a BTree, K1's address will also be smaller than K2's when that BTree is loaded from a database later. The result will be an insane BTree, where various operations do and don't work as expected, seemingly at random.

Now each of the types identified above as "wholly safe to use" never compares two instances of that type by memory address, so there's nothing to worry about here if you use keys of those types. The most common mistake is to use keys that are instances of a user-defined class that doesn't supply its own `__cmp__()` method. Python compares such instances by memory address. This is fine if such instances are used as keys in temporary BTree-based structures used only in a single program run. It can be disastrous if that BTree-based structure is stored to a database, though.

```
>>> class C:
...     pass
...
>>> a, b = C(), C()
>>> print a < b   # this may print 0 if you try it
1
>>> del a, b
>>> a, b = C(), C()
>>> print a < b   # and this may print 0 or 1
0
>>>
```

That example illustrates that comparison of instances of classes that don't define `__cmp__()` yields arbitrary results (but consistent results within a single program run).

Another problem occurs with instances of classes that do define `__cmp__()`, but define it incorrectly. It's possible but rare for a custom `__cmp__()` implementation to violate one of the three required formal properties directly. It's more common for it to "fall back" to address-based comparison by mistake. For example:

```
class Mine:
    def __cmp__(self, other):
        if other.__class__ is Mine:
            return cmp(self.data, other.data)
        else:
            return cmp(self.data, other)
```

It's quite possible there that the `else` clause allows a result to be computed based on memory address. The bug won't show up until a BTree-based structure uses objects of class `Mine` as keys, and also objects of other types as keys, and the structure is loaded from a database, and a sequence of comparisons happens to execute the `else` clause in a case

where the relative order of object memory addresses happened to change.

This is as difficult to track down as it sounds, so best to stay far away from the possibility.

You'll stay out of trouble by follwing these rules, violating them only with great care:

1. Use objects of simple immutable types as keys in BTree-based data structures.
2. Within a single BTree-based data structure, use objects of a single type as keys. Don't use multiple key types in a single structure.
3. If you want to use class instances as keys, and there's any possibility that the structure may be stored in a database, it's crucial that the class define a `__cmp__()` method, and that the method is carefully implemented.

   Any part of a comparison implementation that relies (explicitly or implicitly) on an address-based comparison result will eventually cause serious failure.
4. Do not use `Persistent` objects as keys, or objects of a subclass of `Persistent`.

That last item may be surprising. It stems from details of how conflict resolution is implemented: the states passed to conflict resolution do not materialize persistent subobjects (if a persistent object P is a key in a BTree, then P is a subobject of the bucket containing P). Instead, if an object O references a persistent subobject P directly, and O is involved in a conflict, the states passed to conflict resolution contain an instance of an internal `PersistentReference` stub class everywhere O references P. Two `PersistentReference` instances compare equal if and only if they "represent" the same persistent object; when they're not equal, they compare by memory address, and, as explained before, memory-based comparison must never happen in a sane persistent BTree. Note that it doesn't help in this case if your `Persistent` subclass defines a sane `__cmp__()` method: conflict resolution doesn't know about your class, and so also doesn't know about its `__cmp__()` method. It only sees instances of the internal `PersistentReference` stub class.

Iteration and Mutation

As with a Python dictionary or list, you should not mutate a BTree-based data structure while iterating over it, except that it's fine to replace the value associated with an existing key while iterating. You won't create internal damage in the structure if you try to remove, or add new keys, while iterating, but the results are undefined and unpredictable. A weak attempt is made to raise `RuntimeError` if the size of a BTree-based structure changes while iterating, but it doesn't catch most such cases, and is also unreliable. Example:

```
>>> from BTrees.IIBTree import *
>>> s = IISet(range(10))
>>> list(s)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in s:  # the output is undefined
...     print i,
...     s.remove(i)
0 2 4 6 8
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: the bucket being iterated changed size
>>> list(s)      # this output is also undefined
[1, 3, 5, 7, 9]
>>>
```

Also as with Python dictionaries and lists, the safe and predictable way to mutate a BTree-based structure while iterating over it is to iterate over a copy of the keys. Example:

```
>>> from BTrees.IIBTree import *
>>> s = IISet(range(10))
>>> for i in list(s.keys()):  # this is well defined
```

```
...     print i,
...     s.remove(i)
0 1 2 3 4 5 6 7 8 9
>>> list(s)
[]
>>>
```

BTree Diagnostic Tools

A BTree (or TreeSet) is a complex data structure, really a graph of variable- size nodes, connected in multiple ways via three distinct kinds of C pointers. There are some tools available to help check internal consistency of a BTree as a whole.

Most generally useful is the `BTrees.check` module. The `check.check()` function examines a BTree (or Bucket, Set, or TreeSet) for value-based consistency, such as that the keys are in strictly increasing order. See the function docstring for details. The `check.display()` function displays the internal structure of a BTree.

BTrees and TreeSets also have a `_check()` method. This verifies that the (possibly many) internal pointers in a BTree or TreeSet are mutually consistent, and raises `AssertionError` if they're not.

If a `check.check()` or `_check()` call fails, it may point to a bug in the implementation of BTrees or conflict resolution, or may point to database corruption.

Repairing a damaged BTree is usually best done by making a copy of it. For example, if *self.data* is bound to a corrupted IOBTree,

```
self.data = IOBTree(self.data)
```

usually suffices. If object identity needs to be preserved,

```
acopy = IOBTree(self.data)
self.data.clear()
self.data.update(acopy)
```

does the same, but leaves *self.data* bound to the same object.

Resources

Introduction to the Zope Object Database, by Jim Fulton: — Goes into much greater detail, explaining advanced uses of the ZODB and how it's actually implemented. A definitive reference, and highly recommended. — http://www.python.org/workshops/2000-01/proceedings/papers/fulton/zodb3.html

Persistent Programing with ZODB, by Jeremy Hylton and Barry Warsaw: — Slides for a tutorial presented at the 10th Python conference. Covers much of the same ground as this guide, with more details in some areas and less in others. — http://www.zope.org/Members/bwarsaw/ipc10-slides

GNU Free Documentation License

Version 1.1, March 2000 —

Copyright 2000 Free Software Foundation, Inc. — 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA — Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front- Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back- Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This Licence

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and

conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back- Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Using zc.zodbdgc (fix PosKeyError's)

This article was written by Hanno Schlichting

The zc.zodbdgc library contains two useful features. On the one hand it supports advanced ZODB packing and garbage collection approaches and on the other hand it includes the ability to create a database of all persistent references.

The second feature allows us to debug and repair PosKeyErrors by finding the persistent object(s) that point to the lost object.

Note: This documentation applies to ZODB 3.9 and later. Earlier versions of the ZODB are not supported, as they lack the fast storage iteration API's required by `zc.zodbdgc`.

Note: Unless you're using multi-databases, this documentation does not apply to RelStorage which has the same features built-in, but accessible in different ways. Look at the options for the `zodbpack` script. The `--prepack` option creates a table containing the same information as we are creating in the reference database.

If you *are* using multi-databases, be aware that RelStorage 2.0 is needed to perform packing and garbage collection with `zc.zodbdgc`, and those features only work in history-free databases.

It's important to realize that there is currently no way to perform garbage collection in a history-preserving multi-database RelStorage.

Setup

We'll assume you are familiar with a buildout setup. A typical config might look like this:

```
[buildout]
parts =
  zeo
  zeopy
```

```
  zeo-conf
  zodbdgc
  refdb-conf

[zeo]
recipe = plone.recipe.zeoserver
zeo-address = 127.0.0.1:8100
blob-storage = ${buildout:directory}/var/blobstorage
pack-gc = false
pack-keep-old = false

[zeopy]
recipe = zc.recipe.egg
eggs =
    ZODB3
    zc.zodbdgc
interpreter = zeopy
scripts = zeopy

[zeo-conf]
recipe = collective.recipe.template
input = inline:
  <zodb main>
    <zeoclient>
      blob-dir ${buildout:directory}/var/blobstorage
      shared-blob-dir yes
      server ${zeo:zeo-address}
      storage 1
      name zeostorage
      var ${buildout:directory}/var
    </zeoclient>
  </zodb>
output = ${buildout:directory}/etc/zeo.conf

[zodbdgc]
recipe = zc.recipe.egg
eggs = zc.zodbdgc

[refdb-conf]
recipe = collective.recipe.template
input = inline:
  <zodb main>
    <filestorage 1>
      path ${buildout:directory}/var/refdb.fs
    </filestorage>
  </zodb>
output = ${buildout:directory}/etc/refdb.conf
```

Garbage collection

We configured the ZEO server to skip garbage collection as part of the normal pack in the above config (*pack-gc = false*). Instead we use explicit garbage collection via a different job:

```
bin/multi-zodb-gc etc/zeo.conf
```

On larger databases garbage collection can take a couple hours. We can run this only once a week or even less frequent.

All explicitly deleted objects will still be packed away by the normal pack, so the database doesn't grow out-of-bound. We can also run the analysis against a database copy, taking away load from the live database and only write the resulting deletions to the production database.

Packing

We can do regular packing every day while the ZEO server is running, via:

```
bin/zeopack
```

Packing without garbage collection is much faster.

Reference analysis and POSKeyErrors

If our database has any POSKeyErrors, we can find and repair those.

Either we already have the oids of lost objects, or we can check the entire database for any errors. To check everything we run the following command:

```
$ bin/multi-zodb-check-refs etc/zeo.conf
```

This can take about 15 to 30 minutes on moderately sized databases of up to 10gb, dependent on disk speed. We'll write down the reported errors, as we'll need them later on to analyze them.

If there are any lost objects, we can create a reference database to make it easier to debug and find those lost objects:

```
$ bin/multi-zodb-check-refs -r var/refdb.fs etc/zeo.conf
```

This is significantly slower and can take several hours to complete. Once this is complete we can open the generated database via our interpreter:

```
$ bin/zeopy

>>> import ZODB.config
>>> db = ZODB.config.databaseFromFile(open('./etc/refdb.conf'))
>>> conn = db.open()
>>> refs = conn.root()['references']
```

If we've gotten this error report:

```
!!! main 13184375 ?
POSKeyError: 0xc92d77
```

We can look up the persistent oid it was referenced from via:

```
>>> parent = list(refs['main'][13184375])
>>> parent
[13178389]
```

We can also get the hex representation:

```
>>> from ZODB.utils import p64
>>> p64(parent[0])
'\x00\x00\x00\x00\x00\xc9\x16\x15'
```

With this information, we should get back to our actual database and look up this object. We'll leave the ref db open, as we might need to recursively look up some more objects, until we get one we can identify and work on.

We could load the parent. In a debug prompt we could do something like:

```
>>> app._p_jar.get('\x00\x00\x00\x00\x00\xc9\x16\x15')
2010-04-28 14:28:28 ERROR ZODB.Connection Couldn't load state for 0xc91615
Traceback (most recent call last):
...
ZODB.POSException.POSKeyError: 0xc92d77
```

Gah, this gives us the POSKeyError of course. But we can load the actual data of the parent, to get an idea of what this is:

```
>>> app._p_jar.db()._storage.load('\x00\x00\x00\x00\x00\xc9\x16\x15', '')
('cBTrees.IOBTree
IOBucket
q\x01.((J$KT\x02ccopy_reg
_reconstructor
q\x02(cfive.intid.keyreference
KeyReferenceToPersistent
...
```

Now we can be real evil and create a new fake object in place of the missing one:

```
>>> import transaction
>>> transaction.begin()
```

The persistent oid that was reported missing was `13184375`:

```
>>> from ZODB.utils import p64
>>> p64(13184375)
'\x00\x00\x00\x00\x00\xc9-w'

>>> from persistent import Persistent
>>> a = Persistent()
>>> a._p_oid = '\x00\x00\x00\x00\x00\xc9-w'
```

We cannot use the `add` method of the connection, as this would assign the object a new persistent oid. So we replicate its internals here:

```
>>> a._p_jar = app._p_jar
>>> app._p_jar._register(a)
>>> app._p_jar._added[a._p_oid] = a

>>> transaction.commit()
```

Both getting the object as well as its parent will work now:

```
>>> app._p_jar.get('\x00\x00\x00\x00\x00\xc9-w')
<persistent.Persistent object at 0xa3e348c>

>>> app._p_jar.get('\x00\x00\x00\x00\x00\xc9\x16\x15')
BTrees.IOBTree.IOBucket([(39078692, <five.intid.keyreference...
```

Once we are finished we should be nice and close all databases:

```
>>> conn.close()
>>> db.close()
```

Depending on the class of object that went missing, we might need to use a different persistent class, like a persistent mapping or a BTree bucket.

In general it's best to remove the parent object and thus our fake object from the database and rebuild the data structure again via the proper application level API's.

Other ZODB Resources

- IBM developerWorks Example-driven ZODB
- How To Love ZODB and Forget RDBMS
- Very old ZODB wiki
- The ZODB Book (in progress)

CHAPTER 8

Downloads

ZODB is distributed through the Python Package Index.

You can install the ZODB using pip command:

```
$ pip install ZODB
```

CHAPTER 9

Community and contributing

Discussion occurs on the ZODB mailing list. (And for the transaction system on the transaction list)

Bug reporting and feature requests are submitted through github issue trackers for various ZODB components:

- ZODB
- persistent
- transaction
- BTrees
- ZEO (client-server framework)

If you'd like to contribute then we'll gladly accept work on documentation, helping out other developers and users at the mailing list, submitting bugs, creating proposals and writing code.

ZODB is a project managed by the Zope Foundation so you can get write access for contributing directly - check out the foundation's Zope Developer Information.

Python Module Index

Z

Index

Symbols

A

B

C

D

F

G

U

Y

Z