ZODB documentation and articles

Zope Developer Community

Apr 20, 2021

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.
- Relationships between objects are handled very naturally, supporting complex object graphs without joins.

Check out the *Tutorial*!

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

CHAPTER 1

Learning more

1.1 Introduction

1.1.1 Transactions

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.

1.1.2 Other notable ZODB features

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.

Applications don't have to invalidate cache entries. The database invalidates cache entries automatically.

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.

Easy testing Because application code rarely has database logic, it can usually be unit tested without a database.

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

Garbage collection Removal of unused objects is automatic, so application developers don't have to worry about referential integrity.

Binary large objects, Blobs ZODB blobs are database-managed files. 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.

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.

1.1.3 When should you use ZODB?

You want to focus on your application without writing a lot of database code. ZODB provides highly transparent persistence.

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 your 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.

1.1.4 When should you *not* use ZODB?

- You have very high write volume.

  ZODB can commit thousands of transactions per second with suitable storage configuration and without conflicting changes.

  Internal search indexes can lead to lots of conflicts, and can therefore limit write capacity. If you need high write volume and search beyond mapping access, consider using external indexes.

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

  especially tools designed to work with relational databases

Newt DB addresses these issues to a significant degree. See http://newtdb.org.

1.1.5 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.

1.1.6 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.

1.2 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.

1.2.1 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):
```

(continues on next page)

(continued from previous page)

```
        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].
- 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.

1.2.2 Installation

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

```
$ pip install ZODB
```

1.2.3 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.

[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`.

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

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)
  ```

- 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)
  ```

1.2.4 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.

1.2.5 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.

[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.

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.

1.2.6 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.

1.2.7 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.

1.2.8 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.

1.3 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*.

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

[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.

1.3.1 Installing and running ZODB

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

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.
```

[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.

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.

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:

[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

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

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.

1.3.2 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.

[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.

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
- 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.

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*.

Volatile Attributes

Attributes with names starting with `_v_` are volatile, they are never serialized and not saved to the database. They are useful for caching data that can be computed from other data[#cache]_.

Volatile attributes are local to a specific active object in memory and thus to a specific connection. If an object is removed from the connection cache the volatile attribute is lost.

Setting a volatile attribute does not cause an object to be considered to be modified.

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:

```
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:

```python
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.

1.3.3 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()`.

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.

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

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:

```
>>> 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.

[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.

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.

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.

[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.

[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.

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.

1.4 ZODB articles

1.4.1 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.

[9] at least not now.

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.

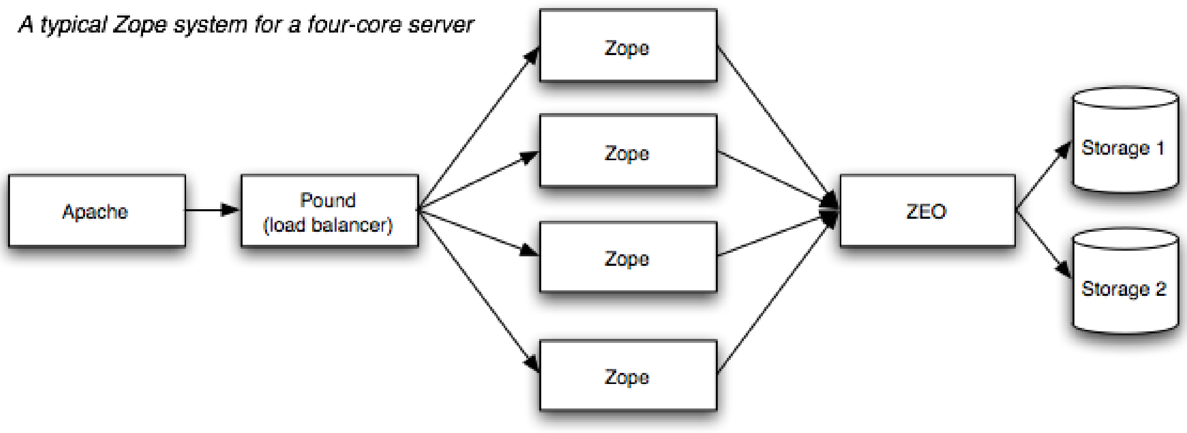
Unfortunately the default MailHost implementation used by Plone is not transaction aware. With it you can see duplicate emails sent. If this is a problem use TransactionalMailHost.

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):
```

(continues on next page)

(continued from previous page)

```
        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"]
```

(continues on next page)

(continued from previous page)

```
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.
- Volatile 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:

```python
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,
```

(continues on next page)

(continued from previous page)

```
  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:

```python
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` transaction 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.

```python
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:

```python
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.

```python
def add_message(self, message):
    """Add a message to the channel.
    message -- text of the message to be added
    """
```

(continues on next page)

(continued from previous page)

```
    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.

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`

[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.

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
```

(continues on next page)

(continued from previous page)

```
>>> 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 sub-object 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.

Modifications

- 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:
```

(continues on next page)

(continued from previous page)

```
  <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.

1.4.2 Other ZODB Resources

- IBM developerWorks Example-driven ZODB
- How To Love ZODB and Forget RDBMS
- Very old ZODB wiki

1.5 Conflict Resolution

1.5.1 Overview

Conflict resolution is a way to resolve transaction conflicts that would otherwise abort a transaction. As such, it risks data integrity in order to try to avoid throwing away potentially computationally expensive transactions.

The risk of harming data integrity should not be underestimated. Writing conflict resolution code takes some responsibility for transactional integrity away from the ZODB, and puts it in the hands of the developer writing the conflict resolution code.

The current conflict resolution code is implemented with a storage mix-in found in ZODB/ConflictResolution.py. The idea's proposal, and an explanation of the interface, can be found here: http://www.zope.org/Members/jim/ZODB/ApplicationLevelConflictResolution

Here is the most pertinent section, somewhat modified for this document's use:

> A new interface is proposed to allow object authors to provide a method for resolving conflicts. When a conflict is detected, then the database checks to see if the class of the object being saved defines the method, _p_resolveConflict. If the method is defined, then the method is called on the object. If the method succeeds, then the object change can be committed, otherwise a ConflictError is raised as usual.
>
> **def _p_resolveConflict(oldState, savedState, newState):** Return the state of the object after resolving different changes.
>
> 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.
>
> If the method cannot resolve the changes, then it should raise ZODB.POSException.ConflictError.
>
> Consider an extremely simple example, a counter:

```
from persistent import Persistent
class PCounter(Persistent):
    '`value` is readonly; increment it with `inc`.'

    # Fool BTree checks for sane comparison :/
    def __cmp__(self, other):
        return object.__cmp__(self, other)
    def __lt__(self, other):
        return object.__lt__(self, other)

    _val = 0
    def inc(self):
        self._val += 1
```

(continues on next page)

(continued from previous page)

```
    @property
    def value(self):
        return self._val
    def _p_resolveConflict(self, oldState, savedState, newState):
        oldState['_val'] = (
            savedState.get('_val', 0) +
            newState.get('_val', 0) -
            oldState.get('_val', 0))
        return oldState
```

By "state", the excerpt above means the value used by __getstate__ and __setstate__: a dictionary, in most cases. We'll look at more details below, but let's continue the example above with a simple successful resolution story.

First we create a storage and a database, and put a PCounter in the database.

```
>>> import ZODB
>>> db = ZODB.DB('Data.fs')
>>> import transaction
>>> tm_A = transaction.TransactionManager()
>>> conn_A = db.open(transaction_manager=tm_A)
>>> p_A = conn_A.root()['p'] = PCounter()
>>> p_A.value
0
>>> tm_A.commit()
```

Now get another copy of 'p' so we can make a conflict. Think of *conn_A* (connection A) as one thread, and *conn_B* (connection B) as a concurrent thread. *p_A* is a view on the object in the first connection, and *p_B* is a view on *the same persistent object* in the second connection.

```
>>> tm_B = transaction.TransactionManager()
>>> conn_B = db.open(transaction_manager=tm_B)
>>> p_B = conn_B.root()['p']
>>> p_B.value
0
>>> p_A._p_oid == p_B._p_oid
True
```

Now we can make a conflict, and see it resolved.

```
>>> p_A.inc()
>>> p_A.value
1
>>> p_B.inc()
>>> p_B.value
1
>>> tm_B.commit()
>>> p_B.value
1
>>> tm_A.commit()
>>> p_A.value
2
```

We need to synchronize connection B, in any of a variety of ways, to see the change from connection A.

```
>>> p_B.value
1
>>> trans = tm_B.begin()
```

(continues on next page)

(continued from previous page)

```
>>> p_B.value
2
```

A very similar class found in real world use is BTrees.Length.Length.

This conflict resolution approach is simple, yet powerful. However, it has a few caveats and rough edges in practice. The simplicity, then, is a bit of a disguise. Again, be warned, writing conflict resolution code means that you claim significant responsibilty for your data integrity.

Because of the rough edges, the current conflict resolution approach is slated for change (as of this writing, according to Jim Fulton, the ZODB primary author and maintainer). Others have talked about different approaches as well (see, for instance, http://www.python.org/~jeremy/weblog/031031c.html). But for now, the _p_resolveConflict method is what we have.

1.5.2 Caveats and Dangers

Here are caveats for working with this conflict resolution approach. Each sub-section has a "DANGERS" section that outlines what might happen if you ignore the warning. We work from the least danger to the most.

Conflict Resolution Is on the Server

If you are using ZEO or ZRS, be aware that the classes for which you have conflict resolution code *and* the classes of the non-persistent objects they reference must be available to import by the *server* (or ZRS primary).

DANGERS: You think you are going to get conflict resolution, but you won't.

Ignore *self*

Even though the _p_resolveConflict method has a "self", ignore it. Don't change it. You make changes by returning the state. This is effectively a class method.

DANGERS: The changes you make to the instance will be discarded. The instance is not initialized, so other methods that depend on instance attributes will not work.

Here's an example of a broken _p_resolveConflict method:

```
class PCounter2(PCounter):
    def __init__(self):
        self.data = []
    def _p_resolveConflict(self, oldState, savedState, newState):
        self.data.append('bad idea')
        return super(PCounter2, self)._p_resolveConflict(
            oldState, savedState, newState)
```

Now we'll prepare for the conflict again.

```
>>> p2_A = conn_A.root()['p2'] = PCounter2()
>>> p2_A.value
0
>>> tm_A.commit()
>>> trans = tm_B.begin() # sync
>>> p2_B = conn_B.root()['p2']
>>> p2_B.value
0
```

(continues on next page)

(continued from previous page)

```
>>> p2_A._p_oid == p2_B._p_oid
True
```

And now we will make a conflict.

```
>>> p2_A.inc()
>>> p2_A.value
1
>>> p2_B.inc()
>>> p2_B.value
1
>>> tm_B.commit()
>>> p2_B.value
1
>>> tm_A.commit() # doctest: +ELLIPSIS
Traceback (most recent call last):
...
ConflictError: database conflict error...
```

oops!

```
>>> tm_A.abort()
>>> p2_A.value
1
>>> trans = tm_B.begin()
>>> p2_B.value
1
```

Watch Out for Persistent Objects in the State

If the object state has a reference to Persistent objects (instances of classes that inherit from persistent.Persistent) then these references *will not be loaded and are inaccessible*. Instead, persistent objects in the state dictionary are ZODB.ConflictResolution.PersistentReference instances. These objects have the following interface:

```
class IPersistentReference(zope.interface.Interface):
    '''public contract for references to persistent objects from an object
    with conflicts.'''

    oid = zope.interface.Attribute(
        'The oid of the persistent object that this reference represents')

    database_name = zope.interface.Attribute(
        '''The name of the database of the reference, *if* different.

        If not different, None.''')

    klass = zope.interface.Attribute(
        '''class meta data.  Presence is not reliable.''')

    weak = zope.interface.Attribute(
        '''bool: whether this reference is weak''')

    def __cmp__(other):
        '''if other is equivalent reference, return 0; else raise ValueError.
```

(continues on next page)

(continued from previous page)

```
        Equivalent in this case means that oid and database_name are the same.

        If either is a weak reference, we only support `is` equivalence, and
        otherwise raise a ValueError even if the datbase_names and oids are
        the same, rather than guess at the correct semantics.

        It is impossible to sort reliably, since the actual persistent
        class may have its own comparison, and we have no idea what it is.
        We assert that it is reasonably safe to assume that an object is
        equivalent to itself, but that's as much as we can say.

        We don't compare on 'is other', despite the
        PersistentReferenceFactory.data cache, because it is possible to
        have two references to the same object that are spelled with different
        data (for instance, one with a class and one without).'''
```

So let's look at one of these. Let's assume we have three, *old*, *saved*, and *new*, each representing a persistent reference to the same object within a _p_resolveConflict call from the oldState, savedState, and newState[1]. They have an oid, *weak* is False, and *database_name* is None. *klass* happens to be set but this is not always the case.

```
>>> isinstance(new.oid, bytes)
True
>>> new.weak
False
>>> print(new.database_name)
None
>>> new.klass is PCounter
True
```

There are a few subtleties to highlight here. First, notice that the database_name is only present if this is a cross-database reference (see cross-database-references.txt in this directory, and examples below). The database name and oid is sometimes a reasonable way to reliably sort Persistent objects (see zope.app.keyreference, for instance) but if your code compares one PersistentReference with a database_name and another without, you need to refuse to give an answer and raise an exception, because you can't know how the unknown database_name sorts.

We already saw a persistent reference with a database_name of None. Now let's suppose *new* is an example of a

[1] We'll catch persistent references with a class mutable.

```
class PCounter3(PCounter):
    data = []
    def _p_resolveConflict(self, oldState, savedState, newState):
        PCounter3.data.append(
            (oldState.get('other'),
             savedState.get('other'),
             newState.get('other')))
        return super(PCounter3, self)._p_resolveConflict(
            oldState, savedState, newState)
```

```
>>> p3_A = conn_A.root()['p3'] = PCounter3()
>>> p3_A.other = conn_A.root()['p']
>>> tm_A.commit()
>>> trans = tm_B.begin() # sync
>>> p3_B = conn_B.root()['p3']
>>> p3_A.inc()
>>> p3_B.inc()
>>> tm_B.commit()
>>> tm_A.commit()
>>> old, saved, new = PCounter3.data[-1]
```

cross-database reference from a database named '2'[2].

```
>>> new.database_name
'2'
```

As seen, the database_name is available for this cross-database reference, and not for others. References to persistent objects, as defined in seialize.py, have other variations, such as weak references, which are handled but not discussed here[3]

[2] We need a whole different set of databases for this. See cross-database-references.txt in this directory for a discussion of what is going on here.

```
>>> databases = {}
>>> db1 = ZODB.DB('1', databases=databases, database_name='1')
>>> db2 = ZODB.DB('2', databases=databases, database_name='2')
>>> tm_multi_A = transaction.TransactionManager()
>>> conn_1A = db1.open(transaction_manager=tm_multi_A)
>>> conn_2A = conn_1A.get_connection('2')
>>> p4_1A = conn_1A.root()['p4'] = PCounter3()
>>> p5_2A = conn_2A.root()['p5'] = PCounter3()
>>> conn_2A.add(p5_2A)
>>> p4_1A.other = p5_2A
>>> tm_multi_A.commit()
>>> tm_multi_B = transaction.TransactionManager()
>>> conn_1B = db1.open(transaction_manager=tm_multi_B)
>>> p4_1B = conn_1B.root()['p4']
>>> p4_1A.inc()
>>> p4_1B.inc()
>>> tm_multi_B.commit()
>>> tm_multi_A.commit()
>>> old, saved, new = PCounter3.data[-1]
```

[3] We'll simply instantiate PersistentReferences with examples of types described in ZODB/serialize.py.

```
>>> from ZODB.ConflictResolution import PersistentReference
```

```
>>> ref1 = PersistentReference(b'my_oid')
>>> ref1.oid
'my_oid'
>>> print(ref1.klass)
None
>>> print(ref1.database_name)
None
>>> ref1.weak
False
```

```
>>> ref2 = PersistentReference((b'my_oid', 'my_class'))
>>> ref2.oid
'my_oid'
>>> ref2.klass
'my_class'
>>> print(ref2.database_name)
None
>>> ref2.weak
False
```

```
>>> ref3 = PersistentReference(['w', (b'my_oid',)])
>>> ref3.oid
'my_oid'
>>> print(ref3.klass)
None
>>> print(ref3.database_name)
None
>>> ref3.weak
True
```

Second, notice the __cmp__ behavior[4]. This is new behavior after ZODB 3.8 and addresses a serious problem for when persistent objects are compared in an _p_resolveConflict, such as that in the ZODB BTrees code. Prior to this change, it was not safe to use Persistent objects as keys in a BTree. You needed to define a __cmp__ for them to be sorted reliably out of the context of conflict resolution, but then during conflict resolution the sorting would be arbitrary, on the basis of the persistent reference's memory location. This could have lead to inconsistent state for BTrees (or BTree module buckets or tree sets or sets).

Here's an example of how the new behavior stops potentially incorrect resolution.

```
>>> ref3a = PersistentReference(['w', (b'my_oid', 'other_db')])
>>> ref3a.oid
'my_oid'
>>> print(ref3a.klass)
None
>>> ref3a.database_name
'other_db'
>>> ref3a.weak
True
```

```
>>> ref4 = PersistentReference(['m', ('other_db', b'my_oid', 'my_class')])
>>> ref4.oid
'my_oid'
>>> ref4.klass
'my_class'
>>> ref4.database_name
'other_db'
>>> ref4.weak
False
```

```
>>> ref5 = PersistentReference(['n', ('other_db', b'my_oid')])
>>> ref5.oid
'my_oid'
>>> print(ref5.klass)
None
>>> ref5.database_name
'other_db'
>>> ref5.weak
False
```

```
>>> ref6 = PersistentReference([b'my_oid']) # legacy
>>> ref6.oid
'my_oid'
>>> print(ref6.klass)
None
>>> print(ref6.database_name)
None
>>> ref6.weak
True
```

[4] All references are equal to themselves.

```
>>> ref1 == ref1 and ref2 == ref2 and ref4 == ref4 and ref5 == ref5
True
>>> ref3 == ref3 and ref3a == ref3a and ref6 == ref6 # weak references
True
```

Non-weak references with the same oid and database_name are equal.

```
>>> ref1 == ref2 and ref4 == ref5
True
```

Everything else raises a ValueError: weak references with the same oid and database, and references with a different database_name or oid.

```
>>> ref3 == ref6
Traceback (most recent call last):
...
ValueError: can't reliably compare against different PersistentReferences
```

```
>>> import BTrees
>>> treeset_A = conn_A.root()['treeset'] = BTrees.family32.OI.TreeSet()
>>> tm_A.commit()
>>> trans = tm_B.begin() # sync
>>> treeset_B = conn_B.root()['treeset']
>>> treeset_A.insert(PCounter())
1
>>> treeset_B.insert(PCounter())
1
>>> tm_B.commit()
>>> tm_A.commit() # doctest: +ELLIPSIS
Traceback (most recent call last):
...
ConflictError: database conflict error...
>>> tm_A.abort()
```

Third, note that, even if the persistent object to which the reference refers changes in the same transaction, the reference is still the same.

DANGERS: subtle and potentially serious. Beyond the two subtleties above, which should now be addressed, there is a general problem for objects that are composites of smaller persistent objects–for instance, a BTree, in which the BTree and each bucket is a persistent object; or a zc.queue.CompositePersistentQueue, which is a persistent queue of persistent queues. Consider the following situation. It is actually solved, but it is a concrete example of what might go wrong.

A BTree (persistent object) has a two buckets (persistent objects). The second bucket has one persistent object in it. Concurrently, one thread deletes the one object in the second bucket, which causes the BTree to dump the bucket; and another thread puts an object in the second bucket. What happens during conflict resolution? Remember, each persistent object cannot see the other. From the perspective of the BTree object, it has no conflicts: one transaction modified it, causing it to lose a bucket; and the other transaction did not change it. From the perspective of the bucket, one transaction deleted an object and the other added it: it will resolve conflicts and say that the bucket has the new object and not the old one. However, it will be garbage collected, and effectively the addition of the new object will be lost.

As mentioned, this story is actually solved for BTrees. As BTrees/MergeTemplate.c explains, whenever savedState or newState for a bucket shows an empty bucket, the code refuses to resolve the conflict: this avoids the situation above.

```
>>> bucket_A = conn_A.root()['bucket'] = BTrees.family32.II.Bucket()
>>> bucket_A[0] = 255
>>> tm_A.commit()
>>> trans = tm_B.begin() # sync
>>> bucket_B = conn_B.root()['bucket']
>>> bucket_B[1] = 254
>>> del bucket_A[0]
>>> tm_B.commit()
>>> tm_A.commit() # doctest: +ELLIPSIS
Traceback (most recent call last):
```

(continues on next page)

```
>>> ref1 == PersistentReference(('another_oid', 'my_class'))
Traceback (most recent call last):
...
ValueError: can't reliably compare against different PersistentReferences
```

```
>>> ref4 == PersistentReference(
...     ['m', ('another_db', 'my_oid', 'my_class')])
Traceback (most recent call last):
...
ValueError: can't reliably compare against different PersistentReferences
```

(continued from previous page)

```
...
ConflictError: database conflict error...
>>> tm_A.abort()
```

However, the story highlights the kinds of subtle problems that units made up of multiple composite Persistent objects need to contemplate. Any structure made up of objects that contain persistent objects with conflict resolution code, as a catalog index is made up of multiple BTree Buckets and Sets, each with conflict resolution, needs to think through these kinds of problems or be faced with potential data integrity issues.

1.6 Collabortation Diagrams

Caution: This document hasn't been reviewed since 2005 and is likely out of date.

This file contains several collaboration diagrams for the ZODB.

1.6.1 Simple fetch, modify, commit

Participants

- DB: `ZODB.DB.DB`
- C: `ZODB.Connection.Connection`
- S: `ZODB.FileStorage.FileStorage`
- T: `transaction.interfaces.ITransaction`
- TM: `transaction.interfaces.ITransactionManager`
- `o1`, `o2`, `...`: pre-existing persistent objects

Scenario

```
DB.open()
    create C
    TM.registerSynch(C)
TM.begin()
    create T
C.get(1) # fetches o1
C.get(2) # fetches o2
C.get(3) # fetches o3
o1.modify() # anything that modifies o1
    C.register(o1)
        T.join(C)
o2.modify()
    C.register(o2)
        # T.join(C) does not happen again
o1.modify()
    # C.register(o1) doesn't happen again, because o1 was already
    # in the changed state.
T.commit()
```

(continues on next page)

(continued from previous page)

```
    C.beforeCompletion(T)
    C.tpc_begin(T)
        S.tpc_begin(T)
    C.commit(T)
        S.store(1, ..., T)
        S.store(2, ..., T)
        # o3 is not stored, because it wasn't modified
    C.tpc_vote(T)
        S.tpc_vote(T)
    C.tpc_finish(T)
        S.tpc_finish(T, f) # f is a callback function, which arranges
                           # to call DB.invalidate (next)
            DB.invalidate(tid, {1: 1, 2: 1}, C)
                C2.invalidate(tid, {1: 1, 2: 1}) # for all connections
                                                 # C2 to DB, where C2
                                                 # is not C
    TM.free(T)
    C.afterCompletion(T)
        C._flush_invalidations()
        # Processes invalidations that may have come in from other
        # transactions.
```

1.6.2 Simple fetch, modify, abort

Participants

- DB: `ZODB.DB.DB`
- C: `ZODB.Connection.Connection`
- S: `ZODB.FileStorage.FileStorage`
- T: `transaction.interfaces.ITransaction`
- TM: `transaction.interfaces.ITransactionManager`
- `o1`, `o2`, `...`: pre-existing persistent objects

Scenario

```
DB.open()
    create C
    TM.registerSynch(C)
TM.begin()
    create T
C.get(1) # fetches o1
C.get(2) # fetches o2
C.get(3) # fetches o3
o1.modify() # anything that modifies o1
    C.register(o1)
        T.join(C)
o2.modify()
    C.register(o2)
        # T.join(C) does not happen again
o1.modify()
```

(continues on next page)

(continued from previous page)

```
    # C.register(o1) doesn't happen again, because o1 was already
    # in the changed state.
T.abort()
    C.beforeCompletion(T)
    C.abort(T)
        C._cache.invalidate(1)  # toss changes to o1
        C._cache.invalidate(2)  # toss changes to o2
        # o3 wasn't modified, and its cache entry isn't invalidated.
    TM.free(T)
    C.afterCompletion(T)
        C._flush_invalidations()
        # Processes invalidations that may have come in from other
        # transactions.
```

1.6.3 Rollback of a savepoint

Participants

- T: `transaction.interfaces.ITransaction`
- `o1`, `o2`, `o3`: some persistent objects
- `C1`, `C2`, `C3`: resource managers
- `S1`, `S2`: Transaction savepoint objects
- `s11`, `s21`, `s22`: resource-manager savepoints

Scenario

```
create T
o1.modify()
    C1.regisiter(o1)
        T.join(C1)
T.savepoint()
    C1.savepoint()
        return s11
    return S1 = Savepoint(T, [r11])
o1.modify()
    C1.regisiter(o1)
o2.modify()
    C2.regisiter(o2)
        T.join(C2)
T.savepoint()
    C1.savepoint()
        return s21
    C2.savepoint()
        return s22
    return S2 = Savepoint(T, [r21, r22])
o3.modify()
    C3.regisiter(o3)
        T.join(C3)
S1.rollback()
    S2.rollback()
```

(continues on next page)

(continued from previous page)

```
        T.discard()
            C1.discard()
            C2.discard()
            C3.discard()
                o3.invalidate()
    S2.discard()
        s21.discard() # roll back changes since previous, which is r11
            C1.discard(s21)
                o1.invalidate()
                # truncates temporary storage to s21's position
        s22.discard() # roll back changes since previous, which is r11
            C1.discard(s22)
                o2.invalidate()
                # truncates temporary storage to beginning, because
                # s22 was the first savepoint.  (Perhaps conection
                # savepoints record the log position before the
                # data were written, which is 0 in this case.
T.commit()
    C1.beforeCompletion(T)
    C2.beforeCompletion(T)
    C3.beforeCompletion(T)
    C1.tpc_begin(T)
        S1.tpc_begin(T)
    C2.tpc_begin(T)
    C3.tpc_begin(T)
    C1.commit(T)
        S1.store(1, ..., T)
    C2.commit(T)
    C3.commit(T)
    C1.tpc_vote(T)
        S1.tpc_vote(T)
    C2.tpc_vote(T)
    C3.tpc_vote(T)
    C1.tpc_finish(T)
        S1.tpc_finish(T, f) # f is a callback function, which arranges
                           c# to call DB.invalidate (next)
            DB.invalidate(tid, {1: 1}, C)
    TM.free(T)
    C1.afterCompletion(T)
        C1._flush_invalidations()
    C2.afterCompletion(T)
        C2._flush_invalidations()
    C3.afterCompletion(T)
        C3._flush_invalidations()
```

1.7 Cross-Database References

Persistent references to objects in different databases within a multi-database are allowed.

Lets set up a multi-database with 2 databases:

```
>>> import ZODB.tests.util, transaction, persistent
>>> databases = {}
>>> db1 = ZODB.tests.util.DB(databases=databases, database_name='1')
>>> db2 = ZODB.tests.util.DB(databases=databases, database_name='2')
```

And create a persistent object in the first database:

```
>>> tm = transaction.TransactionManager()
>>> conn1 = db1.open(transaction_manager=tm)
>>> p1 = MyClass()
>>> conn1.root()['p'] = p1
>>> tm.commit()
```

First, we get a connection to the second database. We get the second connection using the first connection's *get_connection* method. This is important. When using multiple databases, we need to make sure we use a consistent set of connections so that the objects in the connection caches are connected in a consistent manner.

```
>>> conn2 = conn1.get_connection('2')
```

Now, we'll create a second persistent object in the second database. We'll have a reference to the first object:

```
>>> p2 = MyClass()
>>> conn2.root()['p'] = p2
>>> p2.p1 = p1
>>> tm.commit()
```

Now, let's open a separate connection to database 2. We use it to read *p2*, use *p2* to get to *p1*, and verify that it is in database 1:

```
>>> conn = db2.open()
>>> p2x = conn.root()['p']
>>> p1x = p2x.p1
```

```
>>> p2x is p2, p2x._p_oid == p2._p_oid, p2x._p_jar.db() is db2
(False, True, True)
```

```
>>> p1x is p1, p1x._p_oid == p1._p_oid, p1x._p_jar.db() is db1
(False, True, True)
```

It isn't valid to create references outside a multi database:

```
>>> db3 = ZODB.tests.util.DB()
>>> conn3 = db3.open(transaction_manager=tm)
>>> p3 = MyClass()
>>> conn3.root()['p'] = p3
>>> tm.commit()
```

```
>>> p2.p3 = p3
>>> tm.commit() # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Traceback (most recent call last):
...
InvalidObjectReference:
  ('Attempt to store an object from a foreign database connection',
   <Connection at ...>,
   <ZODB.tests.testcrossdatabasereferences.MyClass...>)
```

```
>>> tm.abort()
```

1.7.1 Databases for new objects

Objects are normally added to a database by making them reachable from an object already in the database. This is unambiguous when there is only one database. With multiple databases, it is not so clear what happens. Consider:

```
>>> p4 = MyClass()
>>> p1.p4 = p4
>>> p2.p4 = p4
```

In this example, the new object is reachable from both *p1* in database 1 and *p2* in database 2. If we commit, which database should *p4* end up in? This sort of ambiguity could lead to subtle bugs. For that reason, an error is generated if we commit changes when new objects are reachable from multiple databases:

```
>>> tm.commit() # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Traceback (most recent call last):
...
InvalidObjectReference:
("A new object is reachable from multiple databases. Won't try to
guess which one was correct!",
<Connection at ...>,
<ZODB.tests.testcrossdatabasereferences.MyClass...>)
```

```
>>> tm.abort()
```

To resolve this ambiguity, we can commit before an object becomes reachable from multiple databases.

```
>>> p4 = MyClass()
>>> p1.p4 = p4
>>> tm.commit()
>>> p2.p4 = p4
>>> tm.commit()
>>> p4._p_jar.db().database_name
'1'
```

This doesn't work with a savepoint:

```
>>> p5 = MyClass()
>>> p1.p5 = p5
>>> s = tm.savepoint()
>>> p2.p5 = p5
>>> tm.commit() # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Traceback (most recent call last):
...
InvalidObjectReference:
("A new object is reachable from multiple databases. Won't try to guess
which one was correct!",
<Connection at ...>,
<ZODB.tests.testcrossdatabasereferences.MyClass...>)
```

```
>>> tm.abort()
```

(Maybe it should.)

We can disambiguate this situation by using the connection add method to explicitly say what database an object belongs to:

```
>>> p5 = MyClass()
>>> p1.p5 = p5
>>> p2.p5 = p5
>>> conn1.add(p5)
>>> tm.commit()
>>> p5._p_jar.db().database_name
'1'
```

This the most explicit and thus the best way, when practical, to avoid the ambiguity.

1.7.2 Dissallowing implicit cross-database references

The database contructor accepts a xrefs keyword argument that defaults to True. If False is passed, the implicit cross database references are disallowed. (Note that currently, implicit cross references are the only kind of cross references allowed.)

```
>>> databases = {}
>>> db1 = ZODB.tests.util.DB(databases=databases, database_name='1')
>>> db2 = ZODB.tests.util.DB(databases=databases, database_name='2',
...                          xrefs=False)
```

In this example, we allow cross-references from db1 to db2, but not the other way around.

```
>>> c1 = db1.open()
>>> c2 = c1.get_connection('2')
>>> c1.root.x = c2.root()
>>> transaction.commit()
>>> c2.root.x = c1.root()
>>> transaction.commit() # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Traceback (most recent call last):
...
InvalidObjectReference:
("Database '2' doesn't allow implicit cross-database references",
<Connection at ...>,
{'x': {}})
```

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

1.7.3 NOTE

This implementation is incomplete. It allows creating and using cross-database references, however, there are a number of facilities missing:

cross-database garbage collection

Garbage collection is done on a database by database basis. If an object on a database only has references to it from other databases, then the object will be garbage collected when its database is packed. The cross-database references to it will be broken.

cross-database undo

Undo is only applied to a single database. Fixing this for multiple databases is going to be extremely difficult. Undo currently poses consistency problems, so it is not (or should not be) widely used.

Cross-database aware (tolerant) export/import

The export/import facility needs to be aware, at least, of cross-database references.

1.8 Event support

Sometimes, you want to react when ZODB does certain things. In the past, ZODB provided ad hoc hook functions for this. Going forward, ZODB will use an event mechanism. ZODB.event.notify is called with events of interest.

If zope.event is installed, then ZODB.event.notify is simply an alias for zope.event. If zope.event isn't installed, then ZODB.event is a noop.

1.9 Historical Connections

1.9.1 Usage

A database can be opened with a read-only, historical connection when given a specific transaction or datetime. This can enable full-context application level conflict resolution, historical exploration and preparation for reverts, or even the use of a historical database revision as "production" while development continues on a "development" head.

A database can be opened historically `at` or `before` a given transaction serial or datetime. Here's a simple example. It should work with any storage that supports `loadBefore`.

We'll begin our example with a fairly standard set up. We

- make a storage and a database;
- open a normal connection;
- modify the database through the connection;
- commit a transaction, remembering the time in UTC;
- modify the database again; and
- commit a transaction.

```
>>> import ZODB.MappingStorage
>>> db = ZODB.MappingStorage.DB()
>>> conn = db.open()
```

```
>>> import persistent.mapping
```

```
>>> conn.root()['first'] = persistent.mapping.PersistentMapping(count=0)
```

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

We wait for some time to pass, record he time, and then make some other changes.

```
>>> import time
>>> time.sleep(.01)
```

```
>>> import datetime
>>> now = utcnow()
>>> time.sleep(.01)
>>> root = conn.root()
>>> root['second'] = persistent.mapping.PersistentMapping()
>>> root['first']['count'] += 1
```

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

Now we will show a historical connection. We'll open one using the `now` value we generated above, and then demonstrate that the state of the original connection, at the mutable head of the database, is different than the historical state.

```
>>> transaction1 = transaction.TransactionManager()
```

```
>>> historical_conn = db.open(transaction_manager=transaction1, at=now)
```

```
>>> sorted(conn.root().keys())
['first', 'second']
>>> conn.root()['first']['count']
1
```

```
>>> sorted(historical_conn.root().keys())
['first']
>>> historical_conn.root()['first']['count']
0
```

Moreover, the historical connection cannot commit changes.

```
>>> historical_conn.root()['first']['count'] += 1
>>> historical_conn.root()['first']['count']
1
>>> transaction1.commit()
Traceback (most recent call last):
...
ReadOnlyHistoryError
>>> transaction1.abort()
>>> historical_conn.root()['first']['count']
0
```

(It is because of the mutable behavior outside of transactional semantics that we must have a separate connection, and associated object cache, per thread, even though the semantics should be readonly.)

As demonstrated, a timezone-naive datetime will be interpreted as UTC. You can also pass a timezone-aware datetime or a serial (transaction id). Here's opening with a serial–the serial of the root at the time of the first commit.

```
>>> historical_serial = historical_conn.root()._p_serial
>>> historical_conn.close()
```

```
>>> historical_conn = db.open(transaction_manager=transaction1,
...                           at=historical_serial)
>>> sorted(historical_conn.root().keys())
['first']
>>> historical_conn.root()['first']['count']
0
>>> historical_conn.close()
```

We've shown the `at` argument. You can also ask to look `before` a datetime or serial. (It's an error to pass both[1]) In

[1] It is an error to try and pass both *at* and *before*.

this example, we're looking at the database immediately prior to the most recent change to the root.

```
>>> serial = conn.root()._p_serial
>>> historical_conn = db.open(
...     transaction_manager=transaction1, before=serial)
>>> sorted(historical_conn.root().keys())
['first']
>>> historical_conn.root()['first']['count']
0
```

In fact, `at` arguments are translated into `before` values because the underlying mechanism is a storage's loadBefore method. When you look at a connection's `before` attribute, it is normalized into a `before` serial, no matter what you pass into `db.open`.

```
>>> print(conn.before)
None
>>> historical_conn.before == serial
True
```

```
>>> conn.close()
```

1.9.2 Configuration

Like normal connections, the database lets you set how many total historical connections can be active without generating a warning, and how many objects should be kept in each historical connection's object cache.

```
>>> db.getHistoricalPoolSize()
3
>>> db.setHistoricalPoolSize(4)
>>> db.getHistoricalPoolSize()
4
```

```
>>> db.getHistoricalCacheSize()
1000
>>> db.setHistoricalCacheSize(2000)
>>> db.getHistoricalCacheSize()
2000
```

In addition, you can specify the minimum number of seconds that an unused historical connection should be kept.

```
>>> db.getHistoricalTimeout()
300
>>> db.setHistoricalTimeout(400)
>>> db.getHistoricalTimeout()
400
```

All three of these values can be specified in a ZConfig file.

```
>>> historical_conn = db.open(
...     transaction_manager=transaction1, at=now, before=historical_serial)
Traceback (most recent call last):
...
ValueError: can only pass zero or one of `at` and `before`
```

```
>>> import ZODB.config
>>> db2 = ZODB.config.databaseFromString('''
...     <zodb>
...       <mappingstorage/>
...       historical-pool-size 3
...       historical-cache-size 1500
...       historical-timeout 6m
...     </zodb>
... ''')
>>> db2.getHistoricalPoolSize()
3
>>> db2.getHistoricalCacheSize()
1500
>>> db2.getHistoricalTimeout()
360
```

The pool lets us reuse connections. To see this, we'll open some connections, close them, and then open them again:

```
>>> conns1 = [db2.open(before=serial) for i in range(4)]
>>> _ = [c.close() for c in conns1]
>>> conns2 = [db2.open(before=serial) for i in range(4)]
```

Now let's look at what we got. The first connection in conns 2 is the last connection in conns1, because it was the last connection closed.

```
>>> conns2[0] is conns1[-1]
True
```

Also for the next two:

```
>>> (conns2[1] is conns1[-2]), (conns2[2] is conns1[-3])
(True, True)
```

But not for the last:

```
>>> conns2[3] is conns1[-4]
False
```

Because the pool size was set to 3.

Connections are also discarded if they haven't been used in a while. To see this, let's close two of the connections:

```
>>> conns2[0].close(); conns2[1].close()
```

We'l also set the historical timeout to be very low:

```
>>> db2.setHistoricalTimeout(.01)
>>> time.sleep(.1)
>>> conns2[2].close(); conns2[3].close()
```

Now, when we open 4 connections:

```
>>> conns1 = [db2.open(before=serial) for i in range(4)]
```

We'll see that only the last 2 connections from conn2 are in the result:

```
>>> [c in conns1 for c in conns2]
[False, False, True, True]
```

If you change the historical cache size, that changes the size of the persistent cache on our connection.

```
>>> historical_conn._cache.cache_size
2000
>>> db.setHistoricalCacheSize(1500)
>>> historical_conn._cache.cache_size
1500
```

1.9.3 Invalidations

Invalidations are ignored for historical connections. This is another white box test.

```
>>> historical_conn = db.open(
...     transaction_manager=transaction1, at=serial)
>>> conn = db.open()
>>> sorted(conn.root().keys())
['first', 'second']
>>> conn.root()['first']['count']
1
>>> sorted(historical_conn.root().keys())
['first', 'second']
>>> historical_conn.root()['first']['count']
1
>>> conn.root()['first']['count'] += 1
>>> conn.root()['third'] = persistent.mapping.PersistentMapping()
>>> transaction.commit()
>>> historical_conn.close()
```

Note that if you try to open an historical connection to a time in the future, you will get an error.

```
>>> historical_conn = db.open(
...     at=utcnow()+datetime.timedelta(1))
Traceback (most recent call last):
...
ValueError: cannot open an historical connection in the future.
```

1.9.4 Warnings

First, if you use datetimes to get a historical connection, be aware that the conversion from datetime to transaction id has some pitfalls. Generally, the transaction ids in the database are only as time-accurate as the system clock was when the transaction id was created. Moreover, leap seconds are handled somewhat naively in the ZODB (largely because they are handled naively in Unix/ POSIX time) so any minute that contains a leap second may contain serials that are a bit off. This is not generally a problem for the ZODB, because serials are guaranteed to increase, but it does highlight the fact that serials are not guaranteed to be accurately connected to time. Generally, they are about as reliable as time.time.

Second, historical connections currently introduce potentially wide variance in memory requirements for the applications. Since you can open up many connections to different serials, and each gets their own pool, you may collect quite a few connections. For now, at least, if you use this feature you need to be particularly careful of your memory usage. Get rid of pools when you know you can, and reuse the exact same values for `at` or `before` when possible. If historical connections are used for conflict resolution, these connections will probably be temporary–not saved in a pool–so that the extra memory usage would also be brief and unlikely to overlap.

1.10 Persistent Classes

NOTE: persistent classes are EXPERIMENTAL and, in some sense, incomplete. This module exists largely to test changes made to support Zope 2 ZClasses, with their historical flaws.

The persistentclass module provides a meta class that can be used to implement persistent classes.

Persistent classes have the following properties:

- They cannot be turned into ghosts
- They can only contain picklable subobjects
- They don't live in regular file-system modules

Let's look at an example:

```
>>> def __init__(self, name):
...     self.name = name
```

```
>>> def foo(self):
...     return self.name, self.kind
```

```
>>> import ZODB.persistentclass
>>> C = ZODB.persistentclass.PersistentMetaClass(
...     'C', (object, ), dict(
...     __init__ = __init__,
...     __module__ = '__zodb__',
...     foo = foo,
...     kind = 'sample',
...     ))
```

This example is obviously a bit contrived. In particular, we defined the methods outside of the class. Why? Because all of the items in a persistent class must be picklable. We defined the methods as global functions to make them picklable.

Also note that we explicitly set the module. Persistent classes don't live in normal Python modules. Rather, they live in the database. We use information in `__module__` to record where in the database. When we want to use a database, we will need to supply a custom class factory to load instances of the class.

The class we created works a lot like other persistent objects. It has standard standard persistent attributes:

```
>>> C._p_oid
>>> C._p_jar
>>> C._p_serial
>>> C._p_changed
False
```

Because we haven't saved the object, the jar, oid, and serial are all None and it's not changed.

We can create and use instances of the class:

```
>>> c = C('first')
>>> c.foo()
('first', 'sample')
```

We can modify the class and none of the persistent attributes will change because the object hasn't been saved.

```
>>> import six
>>> def bar(self):
...     six.print_('bar', self.name)
>>> C.bar = bar
>>> c.bar()
bar first
```

```
>>> C._p_oid
>>> C._p_jar
>>> C._p_serial
>>> C._p_changed
False
```

Now, we can store the class in a database. We're going to use an explicit transaction manager so that we can show parallel transactions without having to use threads.

```
>>> import transaction
>>> tm = transaction.TransactionManager()
>>> connection = some_database.open(transaction_manager=tm)
>>> connection.root()['C'] = C
>>> tm.commit()
```

Now, if we look at the persistence variables, we'll see that they have values:

```
>>> C._p_oid
'\x00\x00\x00\x00\x00\x00\x00\x01'
>>> C._p_jar is not None
True
>>> C._p_serial is not None
True
>>> C._p_changed
False
```

Now, if we modify the class:

```
>>> def baz(self):
...     six.print_('baz', self.name)
>>> C.baz = baz
>>> c.baz()
baz first
```

We'll see that the class has changed:

```
>>> C._p_changed
True
```

If we abort the transaction:

```
>>> tm.abort()
```

Then the class will return to it's prior state:

```
>>> c.baz()
Traceback (most recent call last):
...
AttributeError: 'C' object has no attribute 'baz'
```

```
>>> c.bar()
bar first
```

We can open another connection and access the class there.

```
>>> tm2 = transaction.TransactionManager()
>>> connection2 = some_database.open(transaction_manager=tm2)
```

```
>>> C2 = connection2.root()['C']
>>> c2 = C2('other')
>>> c2.bar()
bar other
```

If we make changes without committing them:

```
>>> C.bar = baz
>>> c.bar()
baz first
```

```
>>> C is C2
False
```

Other connections are unaffected:

```
>>> connection2.sync()
>>> c2.bar()
bar other
```

Until we commit:

```
>>> tm.commit()
>>> connection2.sync()
>>> c2.bar()
baz other
```

Similarly, we don't see changes made in other connections:

```
>>> C2.color = 'red'
>>> tm2.commit()
```

```
>>> c.color
Traceback (most recent call last):
...
AttributeError: 'C' object has no attribute 'color'
```

until we sync:

```
>>> connection.sync()
>>> c.color
'red'
```

1.10.1 Instances of Persistent Classes

We can, of course, store instances of persistent classes in the database:

```
>>> c.color = 'blue'
>>> connection.root()['c'] = c
>>> tm.commit()
```

```
>>> connection2.sync()
>>> connection2.root()['c'].color
'blue'
```

NOTE: If a non-persistent instance of a persistent class is copied, the class may be copied as well. This is usually not the desired result.

1.10.2 Persistent instances of persistent classes

Persistent instances of persistent classes are handled differently than normal instances. When we copy a persistent instances of a persistent class, we want to avoid copying the class.

Lets create a persistent class that subclasses Persistent:

```
>>> import persistent
>>> class P(persistent.Persistent, C):
...     __module__ = '__zodb__'
...     color = 'green'
```

```
>>> connection.root()['P'] = P
```

```
>>> import persistent.mapping
>>> connection.root()['obs'] = persistent.mapping.PersistentMapping()
>>> p = P('p')
>>> connection.root()['obs']['p'] = p
>>> tm.commit()
```

You might be wondering why we didn't just stick 'p' into the root object. We created an intermediate persistent object instead. We are storing persistent classes in the root object. To create a ghost for a persistent instance of a persistent class, we need to be able to be able to access the root object and it must be loaded first. If the instance was in the root object, we'd be unable to create it while loading the root object.

Now, if we try to load it, we get a broken object:

```
>>> connection2.sync()
>>> connection2.root()['obs']['p']
<persistent broken __zodb__.P instance '\x00\x00\x00\x00\x00\x00\x00\x04'>
```

because the module, *__zodb__* can't be loaded. We need to provide a class factory that knows about this special module. Here we'll supply a sample class factory that looks up a class name in the database root if the module is *__zodb__*. It falls back to the normal class lookup for other modules:

```
>>> from ZODB.broken import find_global
>>> def classFactory(connection, modulename, globalname):
...     if modulename == '__zodb__':
...        return connection.root()[globalname]
...     return find_global(modulename, globalname)
```

```
>>> some_database.classFactory = classFactory
```

Normally, the classFactory should be set before a database is opened. We'll reopen the connections we're using. We'll assign the old connections to a variable first to prevent getting them from the connection pool:

```
>>> old = connection, connection2
>>> connection = some_database.open(transaction_manager=tm)
>>> connection2 = some_database.open(transaction_manager=tm2)
```

Now, we can read the object:

```
>>> connection2.root()['obs']['p'].color
'green'
>>> connection2.root()['obs']['p'].color = 'blue'
>>> tm2.commit()
```

```
>>> connection.sync()
>>> p = connection.root()['obs']['p']
>>> p.color
'blue'
```

1.10.3 Copying

If we copy an instance via export/import, the copy and the original share the same class:

```
>>> file = connection.exportFile(p._p_oid)
>>> _ = file.seek(0)
>>> cp = connection.importFile(file)
>>> file.close()
>>> cp.color
'blue'
```

```
>>> cp is not p
True
```

```
>>> cp.__class__ is p.__class__
True
```

```
>>> tm.abort()
```

XXX test abort of import

1.11 ZODB Utilities Module

The ZODB.utils module provides a number of helpful, somewhat random :), utility functions.

```
>>> import ZODB.utils
```

This document documents a few of them. Over time, it may document more.

1.11.1 64-bit integers and strings

ZODB uses 64-bit transaction ids that are typically represented as strings, but are sometimes manipulated as integers. Object ids are strings too and it is common to ise 64-bit strings that are just packed integers.

Functions p64 and u64 pack and unpack integers as strings:

```
>>> ZODB.utils.p64(250347764455111456)
'\x03yi\xf7"\xa8\xfb '
```

```
>>> print(ZODB.utils.u64(b'\x03yi\xf7"\xa8\xfb '))
250347764455111456
```

The contant z64 has zero packed as a 64-bit string:

```
>>> ZODB.utils.z64
'\x00\x00\x00\x00\x00\x00\x00\x00'
```

1.11.2 Transaction id generation

Storages assign transaction ids as transactions are committed. These are based on UTC time, but must be strictly increasing. The newTid function akes this pretty easy.

To see this work (in a predictable way), we'll first hack time.time:

```
>>> import time
>>> old_time = time.time
>>> time_value = 1224825068.12
>>> faux_time = lambda: time_value
>>> if isinstance(time,type):
...    time.time = staticmethod(faux_time) # Jython
... else:
...     time.time = faux_time
```

Now, if we ask for a new time stamp, we'll get one based on our faux time:

```
>>> tid = ZODB.utils.newTid(None)
>>> tid
'\x03yi\xf7"\xa54\x88'
```

newTid requires an old tid as an argument. The old tid may be None, if we don't have a previous transaction id.

This time is based on the current time, which we can see by converting it to a time stamp.

```
>>> import ZODB.TimeStamp
>>> print(ZODB.TimeStamp.TimeStamp(tid))
2008-10-24 05:11:08.120000
```

To assure that we get a new tid that is later than the old, we can pass an existing tid. Let's pass the tid we just got.

```
>>> tid2 = ZODB.utils.newTid(tid)
>>> ZODB.utils.u64(tid), ZODB.utils.u64(tid2)
(250347764454864008, 250347764454864009)
```

Here, since we called it at the same time, we got a time stamp that was only slightly larger than the previos one. Of course, at a later time, the time stamp we get will be based on the time:

```
>>> time_value = 1224825069.12
>>> tid = ZODB.utils.newTid(tid2)
>>> print(ZODB.TimeStamp.TimeStamp(tid))
2008-10-24 05:11:09.120000
```

```
>>> time.time = old_time
```

1.11.3 Locking support

Storages are required to be thread safe. The locking descriptor helps automate that. It arranges for a lock to be acquired when a function is called and released when a function exits. To demonstrate this, we'll create a "lock" type that simply prints when it is called:

```
>>> class Lock:
...     def acquire(self):
...         print('acquire')
...     def release(self):
...         print('release')
...     def __enter__(self):
...         return self.acquire()
...     def __exit__(self, *ignored):
...         return self.release()
```

Now we'll demonstrate the descriptor:

```
>>> class C:
...     _lock = Lock()
...     _lock_acquire = _lock.acquire
...     _lock_release = _lock.release
...
...     @ZODB.utils.locked
...     def meth(self, *args, **kw):
...         print('meth %r %r' %(args, kw))
```

The descriptor expects the instance it wraps to have a '_lock attribute.

```
>>> C().meth(1, 2, a=3)
acquire
meth (1, 2) {'a': 3}
release
```

1.11.4 Preconditions

Often, we want to supply method preconditions. The locking descriptor supports optional method preconditions[1].

```
>>> class C:
...     def __init__(self):
...         self._lock = Lock()
...         self._opened = True
...         self._transaction = None
...
...     def opened(self):
...         """The object is open
...         """
...         print('checking if open')
```

(continues on next page)

[1] Arguably, preconditions should be handled via separate descriptors, but for ZODB storages, almost all methods need to be locked. Combining preconditions with locking provides both efficiency and concise expressions. A more general-purpose facility would almost certainly provide separate descriptors for preconditions.

(continued from previous page)

```
...         return self._opened
...
...     def not_in_transaction(self):
...         """The object is not in a transaction
...         """
...         print('checking if in a transaction')
...         return self._transaction is None
...
...     @ZODB.utils.locked(opened, not_in_transaction)
...     def meth(self, *args, **kw):
...         print('meth %r %r' % (args, kw))
```

```
>>> c = C()
>>> c.meth(1, 2, a=3)
acquire
checking if open
checking if in a transaction
meth (1, 2) {'a': 3}
release
```

```
>>> c._transaction = 1
>>> c.meth(1, 2, a=3) # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
AssertionError:
('Failed precondition: ', 'The object is not in a transaction')
```

```
>>> c._opened = False
>>> c.meth(1, 2, a=3) # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
AssertionError: ('Failed precondition: ', 'The object is open')
```

1.12 Developers notes

1.12.1 Building

Bootstrap buildout, if necessary using `bootstrap.py`:

```
python bootstrap.py
```

Run the buildout:

```
bin/buildout
```

1.12.2 Testing

The ZODB checkouts are buildouts. When working from a ZODB checkout, first run the bootstrap.py script to initialize the buildout:

> % python bootstrap.py

and then use the buildout script to build ZODB and gather the dependencies:

> % bin/buildout

This creates a test script:

> % bin/test -v

This command will run all the tests, printing a single dot for each test. When it finishes, it will print a test summary. The exact number of tests can vary depending on platform and available third-party libraries.:

```
Ran 1182 tests in 241.269s

OK
```

The test script has many more options. Use the `-h` or `--help` options to see a file list of options. The default test suite omits several tests that depend on third-party software or that take a long time to run. To run all the available tests use the `--all` option. Running all the tests takes much longer.:

```
Ran 1561 tests in 1461.557s

OK
```

Our primary development platforms are Linux and Mac OS X. The test suite should pass without error on these platforms and, hopefully, Windows, although it can take a long time on Windows – longer if you use ZoneAlarm.

1.12.3 Generating docs

cd to the doc directory and:

```
make html
```

1.12.4 Contributing

Almost any code change should include tests.

Any change that changes features should include documentation updates.

1.13 Change History

1.13.1 5.6.1 (unreleased)

- Fix UnboundLocalError when running fsoids.py script. See issue 285.
- Rework `fsrefs` script to work significantly faster by optimizing how it does IO. See PR 340.
- Require Python 3 to build the documentation.

1.13.2 5.6.0 (2020-06-11)

- Fix race with invalidations when starting a new transaction. The bug affected Storage implementations that rely on mvccadapter, and could result in data corruption (oid loaded at wrong serial after a concurrent commit). See issue 290. As mentionned in pull request #307, interfaces are clarified about the fact that storage implementations must update at a precise moment the value that is returned by lastTransaction(): just after invalidate() or tpc_finish callback.

- Improve volatile attribute `_v_` documentation.
- Make repozo's recover mode atomic by recovering the backup in a temporary file which is then moved to the expected output file.
- Add a new option to repozo in recover mode which allows to verify backups integrity on the fly.
- Drop support for Python 3.4.
- Add support for Python 3.8.
- Fix `DB.undo()` and `DB.undoMultiple()` to close the storage they open behind the scenes when the transaction is committed or rolled back. See issue 268.
- Make TransactionMetaData in charge of (de)serializing extension data. A new `extension_bytes` attribute converts automatically from `extension`, or vice-versa. During storage iteration, `extension_bytes` holds bytes as they are stored (i.e. no deserialization happens). See issue 207.
- Make a connection's savepoint storage implement its own (approximate) `getSize` method instead of relying on the original storage. Previously, this produced confusing DEBUG logging. See issue 282.
- Fix tests with transaction 3.0.
- Fix inconsistent resolution order with zope.interface v5.
- Remove `ConnectionPool.map()`. Instead, `ConnectionPool` is now iterable. See PR 280.

1.13.3 5.5.1 (2018-10-25)

- Fix KeyError on releasing resources of a Connection when closing the DB. This requires at least version 2.4 of the `transaction` package. See issue 208.

1.13.4 5.5.0 (2018-10-13)

- Add support for Python 3.7.
- Bump the dependency on zodbpickle to at least 1.0.1. This is required to avoid a memory leak on Python 2.7. See issue 203.
- Bump the dependency on persistent to at least 4.4.0.
- Make the internal support functions for dealing with OIDs (`p64` and `u64`) somewhat faster and raise more informative exceptions on certain types of bad input. See issue 216.
- Remove support for `python setup.py test`. It hadn't been working for some time. See issue #218.
- Make the tests run faster by avoiding calls to `time.sleep()`.

1.13.5 5.4.0 (2018-03-26)

- ZODB now uses pickle protocol 3 for both Python 2 and Python 3.

  (Previously, protocol 2 was used for Python 2.)

The zodbpickle package provides a *zodbpickle.binary* string type that should be used in Python 2 to cause binary strings to be saved in a pickle binary format, so they can be loaded correctly in Python 3. Pickle protocol 3 is needed for this to work correctly.

- Object identifiers in persistent references are saved as *zodbpickle.binary* strings in Python 2, so that they are loaded correctly in Python 3.
- If an object is missing from the index while packing a `FileStorage`, report its full `oid`.
- Storage imports are a bit faster.
- Storages can be important from non-seekable sources, like file-wrapped pipes.

1.13.6 5.3.0 (2017-08-30)

- Add support for Python 3.6.
- Drop support for Python 3.3.
- Ensure that the `HistoricalStorageAdapter` forwards the `release` method to its base instance. See issue 78.
- Use a higher pickle protocol (2) for serializing objects on Python 2; previously protocol 1 was used. This is *much* more efficient for new-style classes (all persistent objects are new-style), at the cost of being very slightly less efficient for old-style classes.

  Note: On Python 2, this will now allow open `file` objects (but **not** open blobs or sockets) to be pickled (loading the object will result in a closed file); previously this would result in a `TypeError`. Doing so is not recommended as they cannot be loaded in Python 3.

  See issue 179.

1.13.7 5.2.4 (2017-05-17)

- `DB.close` now explicitly frees internal resources. This is helpful to avoid false positives in tests that check for leaks.
- Optimize getting the path to a blob file. See issue 161.
- All classes are new-style classes on Python 2 (they were already new-style on Python 3). This improves performance on PyPy. See issue 160.

1.13.8 5.2.3 (2017-04-11)

- Fix an import error. See issue 158.

1.13.9 5.2.2 (2017-04-11)

- Fixed: A blob misfeature set blob permissions so that blobs and blob directories were only readable by the database process owner, rather than honoring user-controlled permissions (e.g. `umask`). See issue 155.

1.13.10 5.2.1 (2017-04-08)

- Fixed: When opening FileStorages in read-only mode, non-existent files were silently created. Creating a read-only file-storage against a non-existent file errors.

1.13.11 5.2.0 (2017-02-09)

- Call new afterCompletion API on storages to allow them to free resources after transaction complete. See issue 147.
- Take advantage of the new transaction-manager explicit mode to avoid starting transactions unnecessarily when transactions end.
- `Connection.new_oid` delegates to its storage, not the DB. This is helpful for improving concurrency in MVCC storages like RelStorage. See issue 139.
- `persistent` is no longer required at setup time. See issue 119.
- `Connection.close` and `Connection.open` no longer race on `self.transaction_manager`, which could lead to `AttributeError`. This was a bug introduced in 5.0.1. See issue 142.

1.13.12 5.1.1 (2016-11-18)

- Fixed: `ZODB.Connection.TransactionMetaData` didn't support custom data storage that some storages rely on.

1.13.13 5.1.0 (2016-11-17)

- ZODB now translates transaction meta data, `user` and `description` from text to bytes before passing them to storages, and converts them back to text when retrieving them from storages in the `history`, `undoLog` and `undoInfo` methods.

  The `IDatabase` interface was updated to reflect that `history`, `undoLog` and `undoInfo` are available on database objects. (They were always available, but not documented in the interface.)

1.13.14 5.0.1 (2016-11-17)

- Fix an AttributeError that DemoStorage could raise if it was asked to store a blob into a temporary changes before reading a blob. See issue 103.
- Call _p_resolveConflict() even if a conflicting change doesn't change the state. This reverts to the behaviour of 3.10.3 and older.
- Closing a Connection now reverts its `transaction_manager` to None. This helps prevent errors and release resources when the `transaction_manager` was the (default) thread-local manager. See issue 114.
- Many docstrings have been improved.

1.13.15 5.0.0 (2016-09-06)

Major internal improvements and cleanups plus:

- Added a connection `prefetch` method that can be used to request that a storage prefetch data an application will need:

```
conn.prefetch(obj, ...)
```

Where arguments can be objects, object ids, or iterables of objects or object ids.

Added optional `prefetch` methods to the storage APIs. If a storage doesn't support prefetch, then the connection prefetch method is a noop.

- fstail: print the txn offset and header size, instead of only the data offset. fstail can now be used to truncate a DB at the right offset.
- Drop support for old commit protocol. All of the build-in storages implement the new protocol. This new protocol allows storages to provide better write performance by allowing multiple commits to execute in parallel.

1.13.16 5.0.0b1 (2016-08-04)

- fstail: print the txn offset and header size, instead of only the data offset. fstail can now be used to truncate a DB at the right offset.

Numerous internal cleanups, including:

- Changed the way the root object was created. Now the root object is created using a database connection, rather than by making low-level storage calls.
- Drop support for the old commit protocol.
- Internal FileStorage-undo fixes that should allow undo in some cases where it didn't work before.
- Drop the `version` argument to some methods where it was the last argument and optional.

1.13.17 5.0.0a6 (2016-07-21)

- Added a connection `prefetch` method that can be used to request that a storage prefect data an application will need:

  ```
  conn.prefetch(obj, ...)
  ```

  Where arguments can be objects, object ids, or iterables of objects or object ids.

  Added optional `prefetch` methods to the storage APIs. If a storage doesn't support prefetch, then the connection prefetch method is a noop.

1.13.18 5.0.0a5 (2016-07-06)

Drop support for old commit protocol. All of the build-in storages implement the new protocol. This new protocol allows storages to provide better write performance by allowing multiple commits to execute in parallel.

1.13.19 5.0.0a4 (2016-07-05)

See 4.4.2.

1.13.20 5.0.0a3 (2016-07-01)

See 4.4.1.

1.13.21 5.0.0a2 (2016-07-01)

See 4.4.0.

1.13.22 5.0.0a1 (2016-06-20)

Major **internal** implementation changes to the Multi Version Concurrency Control (MVCC) implementation:

- For storages that implement IMVCCStorage (RelStorage), no longer implement MVCC in ZODB.
- For other storages, MVCC is implemented using an additional storage layer. This underlying layer works by calling `loadBefore`. The low-level storage `load` method isn't used any more.

  This change allows server-based storages like ZEO and NEO to be implemented more simply and cleanly.

1.13.23 4.4.3 (2016-08-04)

- Internal FileStorage-undo fixes that should allow undo in some cases where it didn't work before.
- fstail: print the txn offset and header size, instead of only the data offset. fstail can now be used to truncate a DB at the right offset.

1.13.24 4.4.2 (2016-07-08)

Better support of the new commit protocol. This fixes issues with blobs and undo. See pull requests #77, #80, #83

1.13.25 4.4.1 (2016-07-01)

Added IMultiCommitStorage to directly represent the changes in the 4.4.0 release and to make complient storages introspectable.

1.13.26 4.4.0 (2016-06-30)

This release begins evolution to a more effcient commit protocol that allows storage implementations, like NEO, to support multiple transactions committing at the same time, for greater write parallelism.

This release updates IStorage:

- The committed transaction's ID is returned by `tpc_finish`, rather than being returned in response store and tpc_vote results.
- `tpc_vote` is now expected to return `None` or a list of object ids for objects for which conflicts were resolved.

This release works with storages that implemented the older version of the storage interface, but also supports storages that implement the updated interface.

1.13.27 4.3.1 (2016-06-06)

- Fixed: FileStorage loadBefore didn't handle deleted/undone data correctly.

1.13.28 4.3.0 (2016-05-31)

- Drop support for Python 2.6 and 3.2.
- Make the `zodbpickle` dependency required and not conditional. This fixes various packaging issues involving pip and its wheel cache. zodbpickle was only optional under Python 2.6 so this change only impacts users of that version. See https://github.com/zopefoundation/ZODB/pull/42.
- Add support for Python 3.5.
- Avoid failure during cleanup of nested databases that provide MVCC on storage level (Relstorage). https://github.com/zopefoundation/ZODB/issues/45
- Remove useless dependency to *zdaemon* in setup.py. Remove ZEO documentation. Both were leftovers from the time where ZEO was part of this repository.
- Fix possible data corruption after FileStorage is truncated to roll back a transaction. https://github.com/zopefoundation/ZODB/pull/52
- DemoStorage: add support for conflict resolution and fix history() https://github.com/zopefoundation/ZODB/pull/58
- Fixed a test that depended on implementation-specific behavior in tpc_finish

1.13.29 4.2.0 (2015-06-02)

- Declare conditional dependencies using PEP-426 environment markers (fixing interation between pip 7's wheel cache and tox). See https://github.com/zopefoundation/ZODB/issues/36.

1.13.30 4.2.0b1 (2015-05-22)

- Log failed conflict resolution attempts at `DEBUG` level. See: https://github.com/zopefoundation/ZODB/pull/29.
- Fix command-line parsing of `--verbose` and `--verify` arguments. (The short versions, `-v` and `-V`, were parsed correctly.)
- Add support for PyPy.
- Fix the methods in `ZODB.serialize` that find object references under Python 2.7 (used in scripts like `referrers`, `netspace`, and `fsrecover` among others). This requires the addition of the `zodbpickle` dependency.
- FileStorage: fix an edge case when disk space runs out while packing, do not leave the `.pack` file around. That would block any write to the to-be-packed `Data.fs`, because the disk would stay at 0 bytes free. See https://github.com/zopefoundation/ZODB/pull/21.

1.13.31 4.1.0 (2015-01-11)

- Fix registration of custom logging level names ("BLATHER", "TRACE").

  We have been registering them in the wrong order since 2004. Before Python 3.4, the stdlib `logging` module masked the error by registering them in *both* directions.
- Add support for Python 3.4.

1.13.32 4.0.1 (2014-07-13)

- Fix `POSKeyError` during `transaction.commit` when after `savepoint.rollback`. See https://github.com/zopefoundation/ZODB/issues/16
- Ensure that the pickler used in PyPy always has a `persistent_id` attribute (`inst_persistent_id` is not present on the pure-Python pickler). (PR #17)
- Provide better error reporting when trying to load an object on a closed connection.

1.13.33 4.0.0 (2013-08-18)

Finally released.

1.13.34 4.0.0b3 (2013-06-11)

- Switch to using non-backward-compatible pickles (protocol 3, without storing bytes as strings) under Python 3. Updated the magic number for file-storage files under Python3 to indicate the incompatibility.
- Fixed: A `UnicodeDecodeError` could happen for non-ASCII OIDs when using bushy blob layout.

1.13.35 4.0.0b2 (2013-05-14)

- Extended the filename renormalizer used for blob doctests to support the filenames used by ZEO in non-shared mode.
- Added `url` parameter to `setup()` (PyPI says it is required).

1.13.36 4.0.0b1 (2013-05-10)

- Skipped non-unit tests in `setup.py test`. Use the buildout to run tests requiring "layer" support.
- Included the filename in the exception message to support debugging in case `loadBlob` does not find the file.
- Added support for Python 3.2 / 3.3.

Note: ZODB 4.0.x is supported on Python 3.x for *new* applications only. Due to changes in the standard library's pickle support, the Python3 support does **not** provide forward- or backward-compatibility at the data level with Python2. A future version of ZODB may add such support.

Applications which need migrate data from Python2 to Python3 should plan to script this migration using separte databases, e.g. via a "dump-and-reload" approach, or by providing explicit fix-ups of the pickled values as transactions are copied between storages.

1.13.37 4.0.0a4 (2012-12-17)

- Enforced usage of bytes for `_p_serial` of persistent objects (fixes compatibility with recent persistent releases).

1.13.38 4.0.0a3 (2012-12-01)

- **Fixed: An elaborate test for trvial logic corrupted module state in a** way that made other tests fail spuriously.

1.13.39 4.0.0a2 (2012-11-13)

Bugs Fixed

- An unneeded left-over setting in setup.py caused installation with pip to fail.

1.13.40 4.0.0a1 (2012-11-07)

New Features

- The `persistent` and `BTrees` packages are now released as separate distributions, on which ZODB now depends.
- ZODB no longer depends on zope.event. It now uses ZODB.event, which uses zope.event if it is installed. You can override ZODB.event.notify to provide your own event handling, although zope.event is recommended.
- BTrees allowed object keys with insane comparison. (Comparison inherited from object, which compares based on in-process address.) Now BTrees raise TypeError if an attempt is made to save a key with comparison inherited from object. (This doesn't apply to old-style class instances.)

Bugs Fixed

- Ensured that the export file and index file created by `repozo` share the same timestamp.

  https://bugs.launchpad.net/zodb/+bug/993350
- Pinned the `transaction` and `manuel` dependencies to Python 2.5- compatible versions when installing under Python 2.5.

Note: Please see https://github.com/zopefoundation/ZODB/blob/master/HISTORY.rst for older versions of ZODB.

Historical ZODB Changelog

Contents

1.13.41 3.10.5 (2011-11-19)

Bugs Fixed

- Conflict resolution failed when state included cross-database persistent references with classes that couldn't be imported.

1.13.42 3.10.4 (2011-11-17)

Bugs Fixed

- Conflict resolution failed when state included persistent references with classes that couldn't be imported.

1.13.43 3.10.3 (2011-04-12)

Bugs Fixed

- "activity monitor not updated for subconnections when connection returned to pool"

  https://bugs.launchpad.net/zodb/+bug/737198
- "Blob temp file get's removed before it should", https://bugs.launchpad.net/zodb/+bug/595378

  A way this to happen is that a transaction is aborted after the commit process has started. I don't know how this would happen in the wild.

  In 3.10.3, the ZEO tpc_abort call to the server is changed to be synchronous, which should address this case. Maybe there's another case.

Performance enhancements

- Improved ZEO client cache implementation to make it less likely to evict objects that are being used.
- Small (possibly negligable) reduction in CPU in ZEO storage servers to service object loads and in networking code.

1.13.44 3.10.2 (2011-02-12)

Bugs Fixed

- 3.10 introduced an optimization to try to address BTree conflict errors arrising for basing BTree keys on object ids. The optimization caused object ids allocated in aborted transactions to be reused. Unfortunately, this optimzation led to some rather severe failures in some applications. The symptom is a conflict error in which one of the serials mentioned is zero. This optimization has been removed.

  See (for example): https://bugs.launchpad.net/zodb/+bug/665452
- ZEO server transaction timeouts weren't logged as critical.

  https://bugs.launchpad.net/zodb/+bug/670986

1.13.45 3.10.1 (2010-10-27)

Bugs Fixed

- When a transaction rolled back a savepoint after adding objects and subsequently added more objects and committed, an error could be raised "ValueError: A different object already has the same oid" causing the transaction to fail. Worse, this could leave a database in a state where subsequent transactions in the same process would fail.

  https://bugs.launchpad.net/zodb/+bug/665452
- Unix domain sockets didn't work for ZEO (since the addition of IPv6 support). https://bugs.launchpad.net/zodb/+bug/663259
- Removed a missfeature that can cause performance problems when using an external garbage collector with ZEO. When objects were deleted from a storage, invalidations were sent to clients. This makes no sense. It's wildly unlikely that the other connections/clients have copies of the garbage. In normal storage garbage collection, we don't send invalidations. There's no reason to send them when an external garbage collector is used.
- ZEO client cache simulation misshandled invalidations causing incorrect statistics and errors.

1.13.46 3.10.0 (2010-10-08)

New Features

- There are a number of performance enhancements for ZEO storage servers.
- FileStorage indexes use a new format. They are saved and loaded much faster and take less space. Old indexes can still be read, but new indexes won't be readable by older versions of ZODB.
- The API for undoing multiple transactions has changed. To undo multiple transactions in a single transaction, pass a list of transaction identifiers to a database's undoMultiple method. Calling a database's undo method multiple times in the same transaction now raises an exception.

- The ZEO protocol for undo has changed. The only user-visible consequence of this is that when ZODB 3.10 ZEO servers won't support undo for older clients.
- The storage API (IStorage) has been tightened. Now, storages should raise a StorageTransactionError when invalid transactions are passed to tpc_begin, tpc_vote, or tpc_finish.
- ZEO clients (`ClientStorage` instances) now work in forked processes, including those created via `multiprocessing.Process` instances.
- Broken objects now provide the IBroken interface.
- As a convenience, you can now pass an integer port as an address to the ZEO ClientStorage constructor.
- As a convenience, there's a new `client` function in the ZEO package for constructing a ClientStorage instance. It takes the same arguments as the ClientStorage constructor.
- DemoStorages now accept constructor athuments, close_base_on_close and close_changes_on_close, to control whether underlying storages are closed when the DemoStorage is closed.

  https://bugs.launchpad.net/zodb/+bug/118512
- Removed the dependency on zope.proxy.
- Removed support for the _p_independent mini framework, which was made moot by the introduction of multi-version concurrency control several years ago.
- Added support for the transaction retry convenience (transaction-manager attempts method) introduced in the `transaction` 1.1.0 release.
- Enhanced the database opening conveniences:
  - You can now pass storage keyword arguments to ZODB.DB and ZODB.connection.
  - You can now pass None (rather than a storage or file name) to get a database with a mapping storage.
- Databases now warn when committing very large records (> 16MB). This is to try to warn people of likely design mistakes. There is a new option (large_record_size/large-record-size) to control the record size at which the warning is issued.
- Added support for wrapper storages that transform pickle data. Applications for this include compression and encryption. An example wrapper storage implementation, ZODB.tests.hexstorage, was included for testing.

  It is important that storage implementations not assume that storages contain pickles. Renamed IStorageDB to IStorageWrapper and expanded it to provide methods for transforming and untransforming data records. Storages implementations should use these methods to get pickle data from stored records.
- Deprecated ZODB.interfaces.StorageStopIteration. Storage iterator implementations should just raise StopIteration, which means they can now be implemented as generators.
- The filestorage packer configuration option noe accepts values of the form `modname:expression`, allowing the use of packer factories with options.
- Added a new API that allows applications to make sure that current data are read. For example, with:

  ```
  self._p_jar.readCurrent(ob)
  ```

  A conflict error will be raised if the version of ob read by the transaction isn't current when the transaction is committed.

  Normally, ZODB only assures that objects read are consistent, but not necessarily up to date. Checking whether an object is up to date is important when information read from one object is used to update another.

  BTrees are an important case of reading one object to update another. Internal nodes are read to decide which leave notes are updated when a BTree is updated. BTrees now use this new API to make sure that internal nodes are up to date on updates.

- When transactions are aborted, new object ids allocated during the transaction are saved and used in subsequent transactions. This can help in situations where object ids are used as BTree keys and the sequential allocation of object ids leads to conflict errors.
- ZEO servers now support a server_status method for for getting information on the number of clients, lock requests and general statistics.
- ZEO clients now support a client_label constructor argument and client-label configuration-file option to specify a label for a client in server logs. This makes it easier to identify specific clients corresponding to server log entries, especially when there are multiple clients originating from the same machine.
- Improved ZEO server commit lock logging. Now, locking activity is logged at the debug level until the number of waiting lock requests gets above 3. Log at the critical level when the number of waiting lock requests gets above 9.
- The file-storage backup script, repozo, will now create a backup index file if an output file name is given via the –output/-o option.
- Added a '–kill-old-on-full' argument to the repozo backup options: if passed, remove any older full or incremental backup files from the repository after doing a full backup. (https://bugs.launchpad.net/zope2/+bug/143158)
- The mkzeoinst script has been moved to a separate project:

  https://pypi.org/project/zope.mkzeoinstance/

  and is no-longer included with ZODB.
- Removed untested unsupported dbmstorage fossile.
- ZEO servers no longer log their pids in every log message. It's just not interesting. :)

Bugs fixed

- When a pool timeout was specified for a database and old connections were removed due to timing out, an error occured due to a bug in the connection cleanup logic.
- When multi-database connections were no longer used and cleaned up, their subconnections weren't cleaned up properly.
- ZEO didn't work with IPv6 addrsses. Added IPv6 support contributed by Martin v. Loewis.
- A file storage bug could cause ZEO clients to have incorrect information about current object revisions after reconnecting to a database server.
- Updated the 'repozo –kill-old-on-full' option to remove any '.index' files corresponding to backups being removed.
- ZEO extension methods failed when a client reconnected to a storage. (https://bugs.launchpad.net/zodb/+bug/143344)
- Clarified the return Value for lastTransaction in the case when there aren't any transactions. Now a string of 8 nulls (aka "z64") is specified.
- Setting _p_changed on a blob wo actually writing anything caused an error. (https://bugs.launchpad.net/zodb/+bug/440234)
- The verbose mode of the fstest was broken. (https://bugs.launchpad.net/zodb/+bug/475996)
- Object ids created in a savepoint that is rolled back wren't being reused. (https://bugs.launchpad.net/zodb/+bug/588389)

- Database connections didn't invalidate cache entries when conflict errors were raised in response to checkCurrentSerialInTransaction errors. Normally, this shouldn't be a problem, since there should be pending invalidations for these oids which will cause the object to be invalidated. There have been issues with ZEO persistent cache management that have caused out of date data to remain in the cache. (It's possible that the last of these were addressed in the 3.10.0b5.) Invalidating read data when there is a conflict error provides some extra insurance.
- The interface, ZODB.interfaces.IStorage was incorrect. The store method should never return a sequence of oid and serial pairs.
- When a demo storage push method was used to create a new demo storage and the new storage was closed, the original was (incorrectly) closed.
- There were numerous bugs in the ZEO cache tracing and analysis code. Cache simulation, while not perfect, seems to be much more accurate now than it was before.

  The ZEO cache trace statistics and simulation scripts have been given more descriptive names and moved to the ZEO scripts package.
- BTree sets and tree sets didn't correctly check values passed to update or to constructors, causing Python to exit under certain circumstances.
- Fixed bug in copying a BTrees.Length instance. (https://bugs.launchpad.net/zodb/+bug/516653)
- Fixed a serious bug that caused cache failures when run with Python optimization turned on.

  https://bugs.launchpad.net/zodb/+bug/544305
- When using using a ClientStorage in a Storage server, there was a threading bug that caused clients to get disconnected.
- On Mac OS X, clients that connected and disconnected quickly could cause a ZEO server to stop accepting connections, due to a failure to catch errors in the initial part of the connection process.

  The failure to properly handle exceptions while accepting connections is potentially problematic on other platforms.

  Fixes: https://bugs.launchpad.net/zodb/+bug/135108
- Object state management wasn't done correctly when classes implemented custom _p_deavtivate methods. (https://bugs.launchpad.net/zodb/+bug/185066)

1.13.47 3.9.7 (2010-09-28)

Bugs Fixed

- Changes in way that garbage collection treats dictionaries in Python 2.7 broke the object/connection cache implementation. (https://bugs.launchpad.net/zodb/+bug/641481)

  Python 2.7 wasn't officially supported, but we were releasing binaries for it, so …
- Logrotation/repoening via a SIGUSR2 signal wasn't implemented. (https://bugs.launchpad.net/zodb/+bug/143600)
- When using multi-databases, cache-management operations on a connection, cacheMinimize and cacheGC, weren't applied to subconnections.

1.13.48 3.9.6 (2010-09-21)

Bugs Fixed

- Updating blobs in save points could cause spurious "invalidations out of order" errors. https://bugs.launchpad.net/zodb/+bug/509801

  (Thanks to Christian Zagrodnick for chasing this down.)

- If a ZEO client process was restarted while invalidating a ZEO cache entry, the cache could be left in a stage when there is data marked current that should be invalidated, leading to persistent conflict errors.
- Corrupted or invalid cache files prevented ZEO clients from starting. Now, bad cache files are moved aside.
- Invalidations of object records in ZEO caches, where the invalidation transaction ids matched the cached transaction ids should have been ignored.
- Shutting down a process while committing a transaction or processing invalidations from the server could cause ZEO persistent client caches to have invalid data. This, in turn caused stale data to remain in the cache until it was updated.
- Conflict errors didn't invalidate ZEO cache entries.
- When objects were added in savepoints and either the savepoint was rolled back (https://bugs.launchpad.net/zodb/+bug/143560) or the transaction was aborted (https://mail.zope.org/pipermail/zodb-dev/2010-June/013488.html) The objects' _p_oid and _p_jar variables weren't cleared, leading to surprizing errors.
- Objects added in transactions that were later aborted could have _p_changed still set (https://bugs.launchpad.net/zodb/+bug/615758).
- ZEO extension methods failed when a client reconnected to a storage. (https://bugs.launchpad.net/zodb/+bug/143344)
- On Mac OS X, clients that connected and disconnected quickly could cause a ZEO server to stop accepting connections, due to a failure to catch errors in the initial part of the connection process.

  The failure to properly handle exceptions while accepting connections is potentially problematic on other platforms.

  Fixes: https://bugs.launchpad.net/zodb/+bug/135108

- Passing keys or values outside the range of 32-bit ints on 64-bit platforms led to undetected overflow errors. Now these cases cause Type errors to be raised.

  https://bugs.launchpad.net/zodb/+bug/143237

- BTree sets and tree sets didn't correctly check values passed to update or to constructors, causing Python to exit under certain circumstances.
- The verbose mode of the fstest was broken. (https://bugs.launchpad.net/zodb/+bug/475996)

1.13.49 3.9.5 (2010-04-23)

Bugs Fixed

- Fixed bug in cPickleCache's byte size estimation logic. (https://bugs.launchpad.net/zodb/+bug/533015)
- Fixed a serious bug that caused cache failures when run with Python optimization turned on.

  https://bugs.launchpad.net/zodb/+bug/544305

- Fixed a bug that caused savepoint rollback to not properly set object state when objects implemented _p_invalidate methods that reloaded ther state (unghostifiable objects).

  https://bugs.launchpad.net/zodb/+bug/428039

- cross-database wekrefs weren't handled correctly.

  https://bugs.launchpad.net/zodb/+bug/435547

- The mkzeoinst script was fixed to tell people to install and use the mkzeoinstance script. :)

1.13.50 3.9.4 (2009-12-14)

Bugs Fixed

- A ZEO threading bug could cause transactions to read inconsistent data. (This sometimes caused an AssertionError in Connection._setstate_noncurrent.)
- DemoStorage.loadBefore sometimes returned invalid data which would trigger AssertionErrors in ZODB.Connection.
- History support was broken when using stprages that work with ZODB 3.8 and 3.9.
- zope.testing was an unnecessary non-testing dependency.
- Internal ZEO errors were logged at the INFO level, rather than at the error level.
- The FileStorage backup and restore script, repozo, gave a deprecation warning under Python 2.6.
- C Header files weren't installed correctly.
- The undo implementation was incorrect in ways that could cause subtle missbehaviors.

1.13.51 3.9.3 (2009-10-23)

Bugs Fixed

- 2 BTree bugs, introduced by a bug fix in 3.9.0c2, sometimes caused deletion of keys to be improperly handled, resulting in data being available via iteraation but not item access.

1.13.52 3.9.2 (2009-10-13)

Bugs Fixed

- ZEO manages a separate thread for client network IO. It created this thread on import, which caused problems for applications that implemented daemon behavior by forking. Now, the client thread isn't created until needed.
- File-storage pack clean-up tasks that can take a long time unnecessarily blocked other activity.
- In certain rare situations, ZEO client connections would hang during the initial connection setup.

1.13.53 3.9.1 (2009-10-01)

Bugs Fixed

- Conflict errors committing blobs caused ZEO servers to stop committing transactions.

1.13.54 3.9.0 (2009-09-08)

New Features (in more or less reverse chronological order)

- The Database class now has an `xrefs` keyword argument and a corresponding allow-implicit-cross-references configuration option. which default to true. When set to false, cross-database references are disallowed.
- Added support for RelStorage.
- As a convenience, the connection root method for returning the root object can now *also* be used as an object with attributes mapped to the root-object keys.
- Databases have a new method, `transaction`, that can be used with the Python (2.5 and later) `with` statement:

  ```
  db = ZODB.DB(...)
  with db.transaction() as conn:
       # ... do stuff with conn
  ```

  This uses a private transaction manager for the connection. If control exits the block without an error, the transaction is committed, otherwise, it is aborted.
- Convenience functions ZODB.connection and ZEO.connection provide a convenient way to open a connection to a database. They open a database and return a connection to it. When the connection is closed, the database is closed as well.
- The ZODB.config databaseFrom… methods now support multi-databases. If multiple zodb sections are used to define multiple databases, the databases are connected in a multi-database arrangement and the first of the defined databases is returned.
- The zeopack script has gotten a number of improvements:
  - Simplified command-line interface. (The old interface is still supported, except that support for ZEO version 1 servers has been dropped.)
  - Multiple storages can be packed in sequence.
    - This simplifies pack scheduling on servers serving multiple databases.
    - All storages are packed to the same time.
  - You can now specify a time of day to pack to.
  - The script will now time out if it can't connect to s storage in 60 seconds.
- The connection now estimates the object size based on its pickle size and informs the cache about size changes.

  The database got additional configurations options (*cache-size-bytes* and *historical-cache-size-bytes*) to limit the cache size based on the estimated total size of cached objects. The default values are 0 which has the interpretation "do not limit based on the total estimated size". There are corresponding methods to read and set the new configuration parameters.
- Connections now have a public `opened` attribute that is true when the connection is open, and false otherwise. When true, it is the seconds since the epoch (time.time()) when the connection was opened. This is a renaming of the previous `_opened` private variable.
- FileStorage now supports blobs directly.
- You can now control whether FileStorages keep .old files when packing.
- POSKeyErrors are no longer logged by ZEO servers, because they are really client errors.
- A new storage interface, IExternalGC, to support external garbage collection, http://wiki.zope.org/ZODB/ExternalGC, has been defined and implemented for FileStorage and ClientStorage.

- As a small convenience (mainly for tests), you can now specify initial data as a string argument to the Blob constructor.
- ZEO Servers now provide an option, invalidation-age, that allows quick verification of ZEO clients have been disconnected for less than a given time even if the number of transactions the client hasn't seen exceeds the invalidation queue size. This is only recommended if the storage being served supports efficient iteration from a point near the end of the transaction history.
- The FileStorage iterator now handles large files better. When iterating from a starting transaction near the end of the file, the iterator will scan backward from the end of the file to find the starting point. This enhancement makes it practical to take advantage of the new storage server invalidation-age option.
- Previously, database connections were managed as a stack. This tended to cause the same connection(s) to be used over and over. For example, the most used connection would typically be the only connection used. In some rare situations, extra connections could be opened and end up on the top of the stack, causing extreme memory wastage. Now, when connections are placed on the stack, they sink below existing connections that have more active objects.
- There is a new pool-timeout database configuration option to specify that connections unused after the given time interval should be garbage collection. This will provide a means of dealing with extra connections that are created in rare circumstances and that would consume an unreasonable amount of memory.
- The Blob open method now supports a new mode, 'c', to open committed data for reading as an ordinary file, rather than as a blob file. The ordinary file may be used outside the current transaction and even after the blob's database connection has been closed.
- ClientStorage now provides blob cache management. When using non-shared blob directories, you can set a target cache size and the cache will periodically be reduced try to keep it below the target size.

  The client blob directory layout has changed. If you have existing non-shared blob directories, you will have to remove them.
- ZODB 3.9 ZEO clients can connect to ZODB 3.8 servers. ZODB ZEO clients from ZODB 3.2 on can connect to ZODB 3.9 servers.
- When a ZEO cache is stale and would need verification, a ZEO.interfaces.StaleCache event is published (to zope.event). Applications may handle this event and take action such as exiting the application without verifying the cache or starting cold.
- There's a new convenience function, ZEO.DB, for creating databases using ZEO Client Storages. Just call ZEO.DB with the same arguments you would otherwise pass to ZEO.ClientStorage.ClientStorage:

  ```
  import ZEO
  db = ZEO.DB(('some_host', 8200))
  ```
- Object saves are a little faster
- When configuring storages in a storage server, the storage name now defaults to "1". In the overwhelmingly common case that a single storage, the name can now be omitted.
- FileStorage now provides optional garbage collection. A 'gc' keyword option can be passed to the pack method. A false value prevents garbage collection.
- The FileStorage constructor now provides a boolean pack_gc option, which defaults to True, to control whether garbage collection is performed when packing by default. This can be overridden with the gc option to the pack method.

  The ZConfig configuration for FileStorage now includes a pack-gc option, corresponding to the pack_gc constructor argument.
- The FileStorage constructor now has a packer keyword argument that allows an alternative packer to be supplied.

The ZConfig configuration for FileStorage now includes a packer option, corresponding to the packer constructor argument.

- MappingStorage now supports multi-version concurrency control and iteration and provides a better storage implementation example.
- DemoStorage has a number of new features:
  - The ability to use a separate storage, such as a file storage to store changes
  - Blob support
  - Multi-version concurrency control and iteration
  - Explicit support for demo-storage stacking via push and pop methods.
- Wen calling ZODB.DB to create a database, you can now pass a file name, rather than a storage to use a file storage.
- Added support for copying and recovery of blob storages:
  - Added a helper function, ZODB.blob.is_blob_record for testing whether a data record is for a blob. This can be used when iterating over a storage to detect blob records so that blob data can be copied.

    In the future, we may want to build this into a blob-aware iteration interface, so that records get blob file attributes automatically.
  - Added the IBlobStorageRestoreable interfaces for blob storages that support recovery via a restoreBlob method.
  - Updated ZODB.blob.BlobStorage to implement IBlobStorageRestoreable and to have a copyTransactionsFrom method that also copies blob data.
- New *ClientStorage* configuration option *drop_cache_rather_verify*. If this option is true then the ZEO client cache is dropped instead of the long (unoptimized) verification. For large caches, setting this option can avoid effective down times in the order of hours when the connection to the ZEO server was interrupted for a longer time.
- Cleaned-up the storage iteration API and provided an iterator implementation for ZEO.
- Versions are no-longer supported.
- Document conflict resolution (see ZODB/ConflictResolution.txt).
- Support multi-database references in conflict resolution.
- Make it possible to examine oid and (in some situations) database name of persistent object references during conflict resolution.
- Moved the 'transaction' module out of ZODB. ZODB depends upon this module, but it must be installed separately.
- ZODB installation now requires setuptools.
- Added *offset* information to output of *fstail* script. Added test harness for this script.
- Added support for read-only, historical connections based on datetimes or serials (TIDs). See src/ZODB/historical_connections.txt.
- Removed the ThreadedAsync module.
- Now depend on zc.lockfile

Bugs Fixed

- CVE-2009-2701: Fixed a vulnerability in ZEO storage servers when blobs are available. Someone with write access to a ZEO server configured to support blobs could read any file on the system readable by the server process and remove any file removable by the server process.
- BTrees (and TreeSets) kept references to internal keys. https://bugs.launchpad.net/zope3/+bug/294788
- BTree Sets and TreeSets don't support the standard set add method. (Now either add or the original insert method can be used to add an object to a BTree-based set.)
- The runzeo script didn't work without a configuration file. (https://bugs.launchpad.net/zodb/+bug/410571)
- Officially deprecated PersistentDict (https://bugs.launchpad.net/zodb/+bug/400775)
- Calling __setstate__ on a persistent object could under certain uncommon cause the process to crash. (https://bugs.launchpad.net/zodb/+bug/262158)
- When committing transactions involving blobs to ClientStorages with non-shared blob directories, a failure could occur in tpc_finish if there was insufficient disk space to copy the blob file or if the file wasn't available. https://bugs.launchpad.net/zodb/+bug/224169
- Savepoint blob data wasn't properly isolated. If multiple simultaneous savepoints in separate transactions modified the same blob, data from one savepoint would overwrite data for another.
- Savepoint blob data wasn't cleaned up after a transaction abort. https://bugs.launchpad.net/zodb/+bug/323067
- Opening a blob with modes 'r+' or 'a' would fail when the blob had no committed changes.
- PersistentList's sort method did not allow passing of keyword parameters. Changed its sort parameter list to match that of its (Python 2.4+) UserList base class.
- Certain ZEO server errors could cause a client to get into a state where it couldn't commit transactions. https://bugs.launchpad.net/zodb/+bug/374737
- Fixed vulnerabilities in the ZEO network protocol that allow:
  - CVE-2009-0668 Arbitrary Python code execution in ZODB ZEO storage servers
  - CVE-2009-0669 Authentication bypass in ZODB ZEO storage servers

  The vulnerabilities only apply if you are using ZEO to share a database among multiple applications or application instances and if untrusted clients are able to connect to your ZEO servers.
- Fixed the setup test command. It previously depended on private functions in zope.testing.testrunner that don't exist any more.
- ZEO client threads were unnamed, making it hard to debug thread management.
- ZEO protocol 2 support was broken. This caused very old clients to be unable to use new servers.
- zeopack was less flexible than it was before. -h should default to local host.
- The "lawn" layout was being selected by default if the root of the blob directory happened to contain a hidden file or directory such as ".svn". Now hidden files and directories are ignored when choosing the default layout.
- BlobStorage was not compatible with MVCC storages because the wrappers were being removed by each database connection. Fixed.
- Saving indexes for large file storages failed (with the error: RuntimeError: maximum recursion depth exceeded). This can cause a FileStorage to fail to start because it gets an error trying to save its index.
- Sizes of new objects weren't added to the object cache size estimation, causing the object-cache size limiting feature to let the cache grow too large when many objects were added.
- Deleted records weren't removed when packing file storages.

- Fixed analyze.py and added test.
- fixed Python 2.6 compatibility issue with ZEO/zeoserverlog.py
- using hashlib.sha1 if available in order to avoid DeprecationWarning under Python 2.6
- made runzeo -h work
- The monitor server didn't correctly report the actual number of clients.
- Packing could return spurious errors due to errors notifying disconnected clients of new database size statistics.
- Undo sometimes failed for FileStorages configured to support blobs.
- Starting ClientStorages sometimes failed with non-new but empty cache files.
- The history method on ZEO clients failed.
- Fix for bug #251037: Make packing of blob storages non-blocking.
- Fix for bug #220856: Completed implementation of ZEO authentication.
- Fix for bug #184057: Make initialisation of small ZEO client file cache sizes not fail.
- Fix for bug #184054: MappingStorage used to raise a KeyError during *load* instead of a POSKeyError.
- Fixed bug in Connection.TmpStore: load() would not defer to the backend storage for loading blobs.
- Fix for bug #181712: Make ClientStorage update *lastTransaction* directly after connecting to a server, even when no cache verification is necessary.
- Fixed bug in blob filesystem helper: the *isSecure* check was inverted.
- Fixed bug in transaction buffer: a tuple was unpacked incorrectly in *clear*.
- Bugfix the situation in which comparing persistent objects (for instance, as members in BTree set or keys of BTree) might cause data inconsistency during conflict resolution.
- Fixed bug 153316: persistent and BTrees were using *int* for memory sizes which caused errors on x86_64 Intel Xeon machines (using 64-bit Linux).
- Fixed small bug that the Connection.isReadOnly method didn't work after a savepoint.
- Bug #98275: Made ZEO cache more tolerant when invalidating current versions of objects.
- Fixed a serious bug that could cause client I/O to stop (hang). This was accompanied by a critical log message along the lines of: "RuntimeError: dictionary changed size during iteration".
- Fixed bug #127182: Blobs were subclassable which was not desired.
- Fixed bug #126007: tpc_abort had untested code path that was broken.
- Fixed bug #129921: getSize() function in BlobStorage could not deal with garbage files
- Fixed bug in which MVCC would not work for blobs.
- Fixed bug in ClientCache that occurred with objects larger than the total cache size.
- When an error occured attempting to lock a file and logging of said error was enabled.
- FileStorages previously saved indexes after a certain number of writes. This was done during the last phase of two-phase commit, which made this critical phase more subject to errors than it should have been. Also, for large databases, saves were done so infrequently as to be useless. The feature was removed to reduce the chance for errors during the last phase of two-phase commit.
- File storages previously kept an internal object id to transaction id mapping as an optimization. This mapping caused excessive memory usage and failures during the last phase of two-phase commit. This optimization has been removed.

- Refactored handling of invalidations on ZEO clients to fix a possible ordering problem for invalidation messages.
- On many systems, it was impossible to create more than 32K blobs. Added a new blob-directory layout to work around this limitation.
- Fixed bug that could lead to memory errors due to the use of a Python dictionary for a mapping that can grow large.
- Fixed bug #251037: Made packing of blob storages non-blocking.
- Fixed a bug that could cause InvalidObjectReference errors for objects that were explicitly added to a database if the object was modified after a savepoint that added the object.
- Fixed several bugs that caused ZEO cache corruption when connecting to servers. These bugs affected both persistent and non-persistent caches.
- Improved the the ZEO client shutdown support to try to avoid spurious errors on exit, especially for scripts, such as zeopack.
- Packing failed for databases containing cross-database references.
- Cross-database references to databases with empty names weren't constructed properly.
- The zeo client cache used an excessive amount of memory, causing applications with large caches to exhaust available memory.
- Fixed a number of bugs in the handling of persistent ZEO caches:
  - Cache records are written in several steps. If a process exits after writing begins and before it is finishes, the cache will be corrupt on restart. The way records are written was changed to make cache record updates atomic.
  - There was no lock file to prevent opening a cache multiple times at once, which would lead to corruption. Persistent caches now use lock files, in the same way that file storages do.
  - A bug in the cache-opening logic led to cache failure in the unlikely event that a cache has no free blocks.
- When using ZEO Client Storages, Errors occured when trying to store objects too big to fit in the ZEO cache file.
- Fixed bug in blob filesystem helper: the *isSecure* check was inverted.
- Fixed bug in transaction buffer: a tuple was unpacked incorrectly in *clear*.
- Fixed bug in Connection.TmpStore: load() would not defer to the back-end storage for loading blobs.
- Fixed bug #190884: Wrong reference to *POSKeyError* caused NameError.
- Completed implementation of ZEO authentication. This fixes issue 220856.

1.13.55 What's new in ZODB 3.8.0

General

- (unreleased) Fixed setup.py use of setuptools vs distutils, so .c and .h files are included in the bdist_egg.
- The ZODB Storage APIs have been documented and cleaned up.
- ZODB versions are now officially deprecated and support for them will be removed in ZODB 3.9. (They have been widely recognized as deprecated for quite a while.)
- Changed the automatic garbage collection when opening a connection to only apply the garbage collections on those connections in the pool that are closed. (This fixed issue 113923.)

ZEO

- (3.8a1) ZEO's strategoes for avoiding client cache verification were improved in the case that servers are restarted. Before, if transactions were committed after the restart, clients that were up to date or nearly up to date at the time of the restart and then connected had to verify their caches. Now, it is far more likely that a client that reconnects soon after a server restart won't have to verify its cache.
- (3.8a1) Fixed a serious bug that could cause clients that disconnect from and reconnect to a server to get bad invalidation data if the server serves multiple storages with active writes.
- (3.8a1) It is now theoretically possible to use a ClientStorage in a storage server. This might make it possible to offload read load from a storage server at the cost of increasing write latency. This should increase write throughput by offloading reads from the final storage server. This feature is somewhat experimental. It has tests, but hasn't been used in production.

Transactions

- (3.8a1) Add a doom() and isDoomed() interface to the transaction module.

  First step towards the resolution of http://www.zope.org/Collectors/Zope3-dev/655

  A doomed transaction behaves exactly the same way as an active transaction but raises an error on any attempt to commit it, thus forcing an abort.

  Doom is useful in places where abort is unsafe and an exception cannot be raised. This occurs when the programmer wants the code following the doom to run but not commit. It is unsafe to abort in these circumstances as a following get() may implicitly open a new transaction.

  Any attempt to commit a doomed transaction will raise a DoomedTransaction exception.
- (3.8a1) Clean up the ZODB imports in transaction.

  Clean up weird import dance with ZODB. This is unnecessary since the transaction module stopped being imported in ZODB/__init__.py in rev 39622.
- (3.8a1) Support for subtransactions has been removed in favor of save points.

Blobs

- (3.8b1) Updated the Blob implementation in a number of ways. Some of these are backward incompatible with 3.8a1:

  o The Blob class now lives in ZODB.blob

  o The blob openDetached method has been replaced by the committed method.
- (3.8a1) Added new blob feature. See the ZODB/Blobs directory for documentation.

  ZODB now handles (reasonably) large binary objects efficiently. Useful to use from a few kilobytes to at least multiple hundred megabytes.

BTrees

- (3.8a1) Added support for 64-bit integer BTrees as separate types.

  (For now, we're retaining compile-time support for making the regular integer BTrees 64-bit.)

- (3.8a1) Normalize names in modules so that BTrees, Buckets, Sets, and TreeSets can all be accessed with those names in the modules (e.g., BTrees.IOBTree.BTree). This is in addition to the older names (e.g., BTrees.IOBTree.IOBTree). This allows easier drop-in replacement, which can especially be simplify code for packages that want to support both 32-bit and 64-bit BTrees.
- (3.8a1) Describe the interfaces for each module and actually declare the interfaces for each.
- (3.8a1) Fix module references so klass.__module__ points to the Python wrapper module, not the C extension.
- (3.8a1) introduce module families, to group all 32-bit and all 64-bit modules.

1.13.56 What's new in ZODB3 3.7.0

Release date: 2007-04-20

Packaging

- (3.7.0b3) ZODB is now packaged without it's dependencies

  ZODB no longer includes copies of dependencies such as ZConfig, zope.interface and so on. It now treats these as dependencies. If ZODB is installed with easy_install or zc.buildout, the dependencies will be installed automatically.
- (3.7.0b3) ZODB is now a buildout

  ZODB checkouts are now built and tested using zc.buildout.
- (3.7b4) Added logic to avoid spurious errors from the logging system on exit.
- (3.7b2) Removed the "sync" mode for ClientStorage.

  Previously, a ClientStorage could be in either "sync" mode or "async" mode. Now there is just "async" mode. There is now a dedicicated asyncore main loop dedicated to ZEO clients.

  Applications no-longer need to run an asyncore main loop to cause client storages to run in async mode. Even if an application runs an asyncore main loop, it is independent of the loop used by client storages.

  This addresses a test failure on Mac OS X, http://www.zope.org/Collectors/Zope3-dev/650, that I believe was due to a bug in sync mode. Some asyncore-based code was being called from multiple threads that didn't expect to be.

  Converting to always-async mode revealed some bugs that weren't caught before because the tests ran in sync mode. These problems could explain some problems we've seen at times with clients taking a long time to reconnect after a disconnect.

  Added a partial heart beat to try to detect lost connections that aren't otherwise caught, http://mail.zope.org/pipermail/zodb-dev/2005-June/008951.html, by perioidically writing to all connections during periods of inactivity.

Connection management

- (3.7a1) When more than `pool_size` connections have been closed, `DB` forgets the excess (over `pool_size`) connections closed first. Python's cyclic garbage collection can take "a long time" to reclaim them (and may in fact never reclaim them if application code keeps strong references to them), but such forgotten connections can never be opened again, so their caches are now cleared at the time `DB` forgets them. Most applications won't notice a difference, but applications that open many connections, and/or store many large objects in connection caches, and/or store limited resources (such as RDB connections) in connection caches may benefit.

BTrees

- Support for 64-bit integer keys and values has been provided as a compile-time option for the "I" BTrees (e.g. IIBTree).

Documentation

- (3.7a1) Thanks to Stephan Richter for converting many of the doctest files to ReST format. These are now chapters in the Zope 3 apidoc too.

IPersistent

- (3.7a1) The documentation for `_p_oid` now specifies the concrete type of oids (in short, an oid is either None or a non-empty string).

Testing

- (3.7b2) Fixed test-runner output truncation.

  A bug was fixed in the test runner that caused result summaries to be omitted when running on Windows.

Tools

- (3.7a1) The changeover from zLOG to the logging module means that some tools need to perform minimal logging configuration themselves. Changed the zeoup script to do so and thus enable it to emit error messages.

BTrees

- (3.7a1) Suppressed warnings about signedness of characters when compiling under GCC 4.0.x. See http://www.zope.org/Collectors/Zope/2027.

Connection

- (3.7a1) An optimization for loading non-current data (MVCC) was inadvertently disabled in `_setstate()`; this has been repaired.

persistent

- (3.7a1) Suppressed warnings about signedness of characters when compiling under GCC 4.0.x. See http://www.zope.org/Collectors/Zope/2027.
- (3.7a1) PersistentMapping was inadvertently pickling volatile attributes (http://www.zope.org/Collectors/Zope/2052).

After Commit hooks

- (3.7a1) Transaction objects have a new method, `addAfterCommitHook(hook, *args, **kws)`. Hook functions registered with a transaction are called after the transaction commits or aborts. For example, one might want to launch non transactional or asynchrnonous code after a successful, or aborted, commit. See `test_afterCommitHook()` in `transaction/tests/test_transaction.py` for a tutorial doctest, and the `ITransaction` interface for details.

1.13.57 What's new in ZODB3 3.6.2?

Release date: 15-July-2006

DemoStorage

- **(3.6.2) DemoStorage was unable to wrap base storages who did not have** an '_oid' attribute: most notably, ZEO.ClientStorage (http://www.zope.org/Collectors/Zope/2016).

Following is combined news from internal releases (to support ongoing Zope2 / Zope3 development). These are the dates of the internal releases:

- 3.6.1 27-Mar-2006
- 3.6.0 05-Jan-2006
- 3.6b6 01-Jan-2006
- 3.6b5 18-Dec-2005
- 3.6b4 04-Dec-2005
- 3.6b3 06-Nov-2005
- 3.6b2 25-Oct-2005
- 3.6b1 24-Oct-2005
- 3.6a4 07-Oct-2005
- 3.6a3 07-Sep-2005
- 3.6a2 06-Sep-2005
- 3.6a1 04-Sep-2005

Removal of Features Deprecated in ZODB 3.4

(3.6b2) ZODB 3.6 no longer contains features officially deprecated in the ZODB 3.4 release. These include:

- `get_transaction()`. Use `transaction.get()` instead. `transaction.commit()` is a shortcut spelling of `transaction.get().commit()`, and `transaction.abort()` of `transaction.get().abort()`. Note that importing ZODB no longer installs `get_transaction` as a name in Python's `__builtin__` module either.
- The `begin()` method of `Transaction` objects. Use the `begin()` method of a transaction manager instead. `transaction.begin()` is a shortcut spelling to call the default transaction manager's `begin()` method.
- The `dt` argument to `Connection.cacheMinimize()`.
- The `Connection.cacheFullSweep()` method. Use `cacheMinimize()` instead.
- The `Connection.getTransaction()` method. Pass a transaction manager to `DB.open()` instead.

- The `Connection.getLocalTransaction()` method. Pass a transaction manager to `DB.open()` instead.
- The `cache_deactivate_after` and `version_cache_deactivate_after` arguments to the `DB` constructor.
- The `temporary`, `force`, and `waitflag` arguments to `DB.open()`. `DB.open()` no longer blocks (there's no longer a fixed limit on the number of open connections).
- The `transaction` and `txn_mgr``arguments to ``DB.open()`. Use the `transaction_manager` argument instead.
- The `getCacheDeactivateAfter`, `setCacheDeactivateAfter`, `getVersionCacheDeactivateAfter` and `setVersionCacheDeactivateAfter` methods of `DB`.

Persistent

- (3.6.1) Suppressed warnings about signedness of characters when compiling under GCC 4.0.x. See http://www.zope.org/Collectors/Zope/2027.
- (3.6a4) ZODB 3.6 introduces a change to the basic behavior of Persistent objects in a particular end case. Before ZODB 3.6, setting `obj._p_changed` to a true value when `obj` was a ghost was ignored: `obj` remained a ghost, and getting `obj._p_changed` continued to return `None`. Starting with ZODB 3.6, `obj` is activated instead (unghostified), and its state is changed from the ghost state to the changed state. The new behavior is less surprising and more robust.
- (3.6b5) The documentation for `_p_oid` now specifies the concrete type of oids (in short, an oid is either None or a non-empty string).

Commit hooks

- (3.6a1) The `beforeCommitHook()` method has been replaced by the new `addBeforeCommitHook()` method, with a more-robust signature. `beforeCommitHook()` is now deprecated, and will be removed in ZODB 3.8. Thanks to Julien Anguenot for contributing code and tests.

Connection management

- (3.6b6) When more than `pool_size` connections have been closed, `DB` forgets the excess (over `pool_size`) connections closed first. Python's cyclic garbage collection can take "a long time" to reclaim them (and may in fact never reclaim them if application code keeps strong references to them), but such forgotten connections can never be opened again, so their caches are now cleared at the time `DB` forgets them. Most applications won't notice a difference, but applications that open many connections, and/or store many large objects in connection caches, and/or store limited resources (such as RDB connections) in connection caches may benefit.

ZEO

- (3.6a4) Collector 1900. In some cases of pickle exceptions raised by low-level ZEO communication code, callers of `marshal.encode()` could attempt to catch an exception that didn't actually exist, leading to an erroneous `AttributeError` exception. Thanks to Tres Seaver for the diagnosis.

BaseStorage

- (3.6a4) Nothing done by `tpc_abort()` should raise an exception. However, if something does (an error case), `BaseStorage.tpc_abort()` left the commit lock in the acquired state, causing any later attempt to commit changes hang.

Multidatabase

- (3.6b1) The `database_name` for a database in a multidatabase collection can now be specified in a config file's `<zodb>` section, as the value of the optional new `database_name` key. The `.databases` attribute cannot be specified in a config file, but can be passed as the optional new `databases` argument to the `open()` method of a ZConfig factory for type `ZODBDatabase`. For backward compatibility, Zope 2.9 continues to allow using the name in its `<zodb_db name>` config section as the database name (note that `<zodb_db>` is defined by Zope, not by ZODB – it's a Zope-specific extension of ZODB's `<zodb>` section).

PersistentMapping

- (3.6.1) PersistentMapping was inadvertently pickling volatile attributes (http://www.zope.org/Collectors/Zope/2052).
- (3.6b4) `PersistentMapping` makes changes by a `pop()` method call persistent now (http://www.zope.org/Collectors/Zope/2036).
- (3.6a1) The `PersistentMapping` class has an `__iter__()` method now, so that objects of this type work well with Python's iteration protocol. For example, if `x` is a `PersistentMapping` (or Python dictionary, or BTree, or `PersistentDict`, ...), then `for key in x:` iterates over the keys of `x`, `list(x)` creates a list containing `x`'s keys, `iter(x)` creates an iterator for `x`'s keys, and so on.

Tools

- (3.6b5) The changeover from zLOG to the logging module means that some tools need to perform minimal logging configuration themselves. Changed the zeoup script to do so and thus enable it to emit error messages.

BTrees

- (3.6.1) Suppressed warnings about signedness of characters when compiling under GCC 4.0.x. See http://www.zope.org/Collectors/Zope/2027.
- (3.6a1) BTrees and Buckets now implement the `setdefault()` and `pop()` methods. These are exactly like Python's dictionary methods of the same names, except that `setdefault()` requires both arguments (and Python is likely to change to require both arguments too – defaulting the `default` argument to `None` has no viable use cases). Thanks to Ruslan Spivak for contributing code, tests, and documentation.
- (3.6a1) Collector 1873. It wasn't possible to construct a BTree or Bucket from, or apply their update() methods to, a PersistentMapping or PersistentDict. This works now.

ZopeUndo

- (3.6a4) Collector 1810. A previous bugfix (#1726) broke listing undoable transactions for users defined in a non-root acl_users folder. Zope logs a acl_users path together with a username (separated by a space) and this previous fix failed to take this into account.

Connection

- (3.6b5) An optimization for loading non-current data (MVCC) was inadvertently disabled in `_setstate()`; this has been repaired.

Documentation

- (3.6b3) Thanks to Stephan Richter for converting many of the doctest files to ReST format. These are now chapters in the Zope 3 apidoc too.
- (3.6b4) Several misspellings of "occurred" were repaired.

Development

- (3.6a1) The source code for the old ExtensionClass-based Persistence package moved, from ZODB to the Zope 2.9 development tree. ZODB 3.5 makes no use of Persistence, and, indeed, the Persistence package could not be compiled from a ZODB release, since some of the C header files needed appear only in Zope.
- (3.6a3) Re-added the `zeoctl` module, for the same reasons `mkzeoinst` was re-added (see below).
- (3.6a2) The `mkzeoinst` module was re-added to ZEO, because Zope3 has a script that expects to import it from there. ZODB's `mkzeoinst` script was rewritten to invoke the `mkzeoinst` module.

`transact`

- (3.6b4) Collector 1959: The undocumented `transact` module no longer worked. It remains undocumented and untested, but thanks to Janko Hauser it's possible that it works again ;-).

1.13.58 What's new in ZODB3 3.5.1?

Release date: 26-Sep-2005

Following is combined news from internal releases (to support ongoing Zope3 development). These are the dates of the internal releases:

- 3.5.1b2 07-Sep-2005
- 3.5.1b1 06-Sep-2005

Build

- (3.5.1b2) Re-added the `zeoctl` module, for the same reasons `mkzeoinst` was re-added (see below).
- (3.5.1b1) The `mkzeoinst` module was re-added to ZEO, because Zope3 has a script that expects to import it from there. ZODB's `mkzeoinst` script was rewritten to invoke the `mkzeoinst` module.

ZopeUndo

- (3.5.1) Collector 1810. A previous bugfix (#1726) broke listing undoable transactions for users defined in a non-root acl_users folder. Zope logs a acl_users path together with a username (separated by a space) and this previous fix failed to take this into account.

1.13.59 What's new in ZODB3 3.5.0?

Release date: 31-Aug-2005

Following is combined news from internal releases (to support ongoing Zope3 development). These are the dates of the internal releases:

- 3.5a7 11-Aug-2005
- 3.5a6 04-Aug-2005
- 3.5a5 19-Jul-2005
- 3.5a4 14-Jul-2005
- 3.5a3 17-Jun-2005
- 3.5a2 16-Jun-2005
- 3.5a1 10-Jun-2005

Savepoints

- (3.5.0) As for deprecated subtransaction commits, the intent was that making a savepoint would invoke incremental garbage collection on Connection memory caches, to try to reduce the number of objects in cache to the configured cache size. Due to an oversight, this didn't happen, and stopped happening for subtransaction commits too. Making a savepoint (or doing a subtransaction commit) does invoke cache gc now.
- (3.5a3) When a savepoint is made, the states of objects modified so far are saved to a temporary storage (an instance of class `TmpStore`, although that's an internal implementation detail). That storage needs to implement the full storage API too, but was missing the `loadBefore()` method needed for MVCC to retrieve non-current revisions of objects. This could cause spurious errors if a transaction with a pending savepoint needed to fetch an older revision of some object.
- (3.5a4) The `ISavepoint` interface docs said you could roll back to a given savepoint any number of times (until the transaction ends, or until you roll back to an earlier savepoint's state), but the implementation marked a savepoint as invalid after its first use. The implementation has been repaired, to match the docs.

ZEO client cache

- (3.5a6) Two memory leaks in the ZEO client cache were repaired, a major one involving `ZEO.cache.Entry` objects, and a minor one involving empty lists.

Subtransactions are deprecated

- (3.5a4) Subtransactions are deprecated, and will be removed in ZODB 3.7. Use savepoints instead. Savepoints are more powerful, and code using subtransactions does not mix well with code using savepoints (a subtransaction commit forces all current savepoints to become unusable, so code using subtransactions can hurt newer code trying to use savepoints). In general, a subtransaction commit done just to free memory can be changed from:

  ```
  transaction.commit(1)
  ```

  to:

  ```
  transaction.savepoint(True)
  ```

That is, make a savepoint, and forget it. As shown, it's best to pass `True` for the optional `optimistic` argument in this case: because there's no possibility of asking for a rollback later, there's no need to insist that all data managers support rollback.

In rarer cases, a subtransaction commit is followed later by a subtransaction abort. In that case, change the initial:

```
transaction.commit(1)
```

to:

```
sp = transaction.savepoint()
```

and in place of the subtransaction abort:

```
transaction.abort(1)
```

roll back the savepoint instead:

```
sp.rollback()
```

- (3.5a4) Internal uses of subtransactions (transaction `commit()` or `abort()` passing a true argument) were rewritten to use savepoints instead.

Multi-database

- (3.5a1) Preliminary support for persistent cross-database references has been added. See `ZODB/cross-database-references.txt` for an introduction.

Tools

- (3.5a6, 3.5a7) Collector #1847. The ZEO client cache tracing and simulation tools weren't updated to work with ZODB 3.3, and the introduction of MVCC required major reworking of the tracing and simulation code. These tools are in a working state again, although so far lightly tested on just a few applications. In `doc/ZEO/`, see the heavily revised `trace.txt` and `cache.txt`.
- (3.5a5) Collector #1846: If an uncommitted transaction was found, fsrecover.py fell into an infinite loop.

Windows

- (3.5a6) As developed in a long thread starting at http://mail.zope.org/pipermail/zope/2005-July/160433.html there appears to be a race bug in the Microsoft Windows socket implementation, rarely visible in ZEO when multiple processes try to create an "asyncore trigger" simultaneously. Windows-specific code in `ZEO/zrpc/trigger.py` changed to work around this bug when it occurs.

ThreadedAsync.LoopCallback

- (3.5a5) This once again physically replaces Python's `asyncore.loop` function with its own loop function, because it turns out Zope relied on the seemingly unused `LoopCallback.exit_status` global, which was removed in the change described below. Python's `asyncore.loop` is again not invoked, so any breakpoints or debugging prints added to that are again "lost".

- (3.5a4) This replaces Python's `asyncore.loop` function with its own, in order to get notified when `loop()` is first called. The signature of `asyncore.loop` changed in Python 2.4, but `LoopCallback.loop`'s signature didn't change to match. The code here was repaired to be compatible with both old and new signatures, and also repaired to invoke Python's `asyncore.loop()` instead of replacing it entirely (so, for example, debugging prints added to Python's `asyncore.loop` won't be lost anymore).

FileStorage

- (3.5a4) Collector #1830. In some error cases when reading a FileStorage index, the code referenced an undefined global.
- (3.5a4) Collector #1822. The `undoLog()` and `undoInfo()` methods were changed in 3.4a9 to return the documented results. Alas, some pieces of (non-ZODB) code relied on the actual behavior. When the `first` and `last` arguments are both >= 0, these methods now treat them as if they were Python slice indices, including the *first* index but excluding the `last` index. This matches former behavior, although it contradicts older ZODB UML documentation. The documentation in `ZODB.interfaces.IStorageUndoable` was changed to match the new intent.
- (3.5a2) The `_readnext()` method now returns the transaction size as the value of the "size" key. Thanks to Dieter Maurer for the patch, from http://mail.zope.org/pipermail/zodb-dev/2003-October/006157.html. "This is very valuable when you want to spot strange transaction sizes via Zope's 'Undo' tab".

BTrees

- (3.5.a5) Collector 1843. When a non-integer was passed to a method like `keys()` of a Bucket or Set with integer keys, an internal error code was overlooked, leading to everything from "delayed errors" to segfaults. Such cases raise TypeError now, as intended.
- (3.5a4) Collector 1831. The BTree `minKey()` and `maxKey()` methods gave a misleading message if no key satisfying the constraints existed in a non-empty tree.
- (3.5a4) Collector 1829. Clarified that the `minKey()` and `maxKey()` methods raise an exception if no key exists satsifying the constraints.
- (3.5a4) The ancient `convert.py` script was removed. It was intended to convert "old" BTrees to "new" BTrees, but the "old" BTree implementation was removed from ZODB years ago.

1.13.60 What's new in ZODB3 3.4.1?

Release date: 09-Aug-2005

Following are dates of internal releases (to support ongoing Zope 2 development) since ZODB 3.4's last public release:

- 3.4.1b5 08-Aug-2005
- 3.4.1b4 07-Aug-2005
- 3.4.1b3 04-Aug-2005
- 3.4.1b2 02-Aug-2005
- 3.4.1b1 26-Jul-2005
- 3.4.1a6 19-Jul-2005
- 3.4.1a5 12-Jul-2005
- 3.4.1a4 08-Jul-2005

- 3.4.1a3 02-Jul-2005
- 3.4.1a2 29-Jun-2005
- 3.4.1a1 27-Jun-2005

Savepoints

- (3.4.1a1) When a savepoint is made, the states of objects modified so far are saved to a temporary storage (an instance of class `TmpStore`, although that's an internal implementation detail). That storage needs to implement the full storage API too, but was missing the `loadBefore()` method needed for MVCC to retrieve non-current revisions of objects. This could cause spurious errors if a transaction with a pending savepoint needed to fetch an older revision of some object.
- (3.4.1a5) The `ISavepoint` interface docs said you could roll back to a given savepoint any number of times (until the transaction ends, or until you roll back to an earlier savepoint's state), but the implementation marked a savepoint as invalid after its first use. The implementation has been repaired, to match the docs.
- (3.4.1b4) Collector 1860: use an optimistic savepoint in ExportImport (there's no possiblity of rollback here, so no need to insist that the data manager support rollbacks).

ZEO client cache

- (3.4.1b3) Two memory leaks in the ZEO client cache were repaired, a major one involving `ZEO.cache.Entry` objects, and a minor one involving empty lists.

Subtransactions

- (3.4.1a5) Internal uses of subtransactions (transaction `commit()` or `abort()` passing a true argument) were rewritten to use savepoints instead. Application code is strongly encouraged to do this too: subtransactions are weaker, will be deprecated soon, and do not mix well with savepoints (when you do a subtransaction commit, all current savepoints are made unusable). In general, a subtransaction commit done just to free memory can be changed from:

  ```
  transaction.commit(1)
  ```

  to:

  ```
  transaction.savepoint(True)
  ```

  That is, make a savepoint, and forget it. As shown, it's best to pass `True` for the optional `optimistic` argument in this case: because there's no possibility of asking for a rollback later, there's no need to insist that all data managers support rollback.

  In rarer cases, a subtransaction commit is followed later by a subtransaction abort. In that case, change the initial:

  ```
  transaction.commit(1)
  ```

  to:

  ```
  sp = transaction.savepoint()
  ```

  and in place of the subtransaction abort:

```
transaction.abort(1)
```

roll back the savepoint instead:

```
sp.rollback()
```

FileStorage

- (3.4.1a3) Collector #1830. In some error cases when reading a FileStorage index, the code referenced an undefined global.
- (3.4.1a2) Collector #1822. The `undoLog()` and `undoInfo()` methods were changed in 3.4a9 to return the documented results. Alas, some pieces of (non-ZODB) code relied on the actual behavior. When the *first* and *last* arguments are both >= 0, these methods now treat them as if they were Python slice indices, including the *first* index but excluding the *last* index. This matches former behavior, although it contradicts older ZODB UML documentation. The documentation in `ZODB.interfaces.IStorageUndoable` was changed to match the new intent.
- (3.4.1a1) The `UndoSearch._readnext()` method now returns the transaction size as the value of the "size" key. Thanks to Dieter Maurer for the patch, from http://mail.zope.org/pipermail/zodb-dev/2003-October/006157.html. "This is very valuable when you want to spot strange transaction sizes via Zope's 'Undo' tab".

ThreadedAsync.LoopCallback

- (3.4.1a6) This once again physically replaces Python's `asyncore.loop` function with its own loop function, because it turns out Zope relied on the seemingly unused `LoopCallback.exit_status` global, which was removed in the change described below. Python's `asyncore.loop` is again not invoked, so any breakpoints or debugging prints added to that are again "lost".
- (3.4.1a1) This replaces Python's `asyncore.loop` function with its own, in order to get notified when `loop()` is first called. The signature of `asyncore.loop` changed in Python 2.4, but `LoopCallback.loop`'s signature didn't change to match. The code here was repaired to be compatible with both old and new signatures, and also repaired to invoke Python's `asyncore.loop()` instead of replacing it entirely (so, for example, debugging prints added to Python's `asyncore.loop` won't be lost anymore).

Windows

- (3.4.1b2) As developed in a long thread starting at http://mail.zope.org/pipermail/zope/2005-July/160433.html there appears to be a race bug in the Microsoft Windows socket implementation, rarely visible in ZEO when multiple processes try to create an "asyncore trigger" simultaneously. Windows-specific code in `ZEO/zrpc/trigger.py` changed to work around this bug when it occurs.

Tools

- (3.4.1b1 thru 3.4.1b5) Collector #1847. The ZEO client cache tracing and simulation tools weren't updated to work with ZODB 3.3, and the introduction of MVCC required major reworking of the tracing and simulation code. These tools are in a working state again, although so far lightly tested on just a few applications. In `doc/ZEO/`, see the heavily revised `trace.txt` and `cache.txt`.
- (3.4.1a6) Collector #1846: If an uncommitted transaction was found, fsrecover.py fell into an infinite loop.

DemoStorage

- (3.4.1a1) The implementation of `undoLog()` was wrong in several ways; repaired.

BTrees

- (3.4.1a6) Collector 1843. When a non-integer was passed to a method like `keys()` of a Bucket or Set with integer keys, an internal error code was overlooked, leading to everything from "delayed errors" to segfaults. Such cases raise TypeError now, as intended.
- (3.4.1a4) Collector 1831. The BTree `minKey()` and `maxKey()` methods gave a misleading message if no key satisfying the constraints existed in a non-empty tree.
- (3.4.1a3) Collector 1829. Clarified that the `minKey()` and `maxKey()` methods raise an exception if no key exists satsifying the constraints.

1.13.61 What's new in ZODB3 3.4?

Release date: 09-Jun-2005

Following is combined news from the "internal releases" (to support ongoing Zope 2.8 and Zope3 development) since the last public ZODB 3.4 release. These are the dates of the internal releases:

- 3.4c2 06-Jun-2005
- 3.4c1 03-Jun-2005
- 3.4b3 27-May-2005
- 3.4b2 26-May-2005

Connection, DB

- (3.4b3) `.transaction_manager` is now a public attribute of IDataManager, and is the instance of ITransactionManager used by the data manager as its transaction manager. There was previously no way to ask a data manager which transaction manager it was using. It's intended that `transaction_manager` be treated as read-only.
- (3.4b3) For sanity, the `txn_mgr` argument to `DB.open()`, `Connection.__init__()`, and `Connection._setDB()` has been renamed to `transaction_manager`. `txn_mgr` is still accepted, but is deprecated and will be removed in ZODB 3.6. Any code that was using the private `._txn_mgr` attribute of `Connection` will break immediately.

Development

- (3.4b2) ZODB's `test.py` is now a small driver for the shared `zope.testing.testrunner`. See the latter's documentation for command-line arguments.

Error reporting

- (3.4c1) In the unlikely event that `referencesf()` reports an unpickling error (for example, a corrupt database can cause this), the message it produces no longer contains unprintable characters.

Tests

- (3.4c2) `checkCrossDBInvalidations` suffered spurious failures too often on slow and/or busy machines. The test is willing to wait longer for success now.

1.13.62 What's new in ZODB3 3.4b1?

Release date: 19-May-2005

What follows is combined news from the "internal releases" (to support ongoing Zope 2.8 and Zope3 development) since the last public ZODB 3.4 release. These are the dates of the internal releases:

- 3.4b1 19-May-2005
- 3.4a9 12-May-2005
- 3.4a8 09-May-2005
- 3.4a7 06-May-2005
- 3.4a6 05-May-2005
- 3.4a5 25-Apr-2005
- 3.4a4 23-Apr-2005
- 3.4a3 13-Apr-2005
- 3.4a2 03-Apr-2005

transaction

- (3.4a7) If the first activity seen by a new `ThreadTransactionManager` was an explicit `begin()` call, then synchronizers registered after that (but still during the first transaction) were not communicated to the transaction object. As a result, the `afterCompletion()` methods of registered synchronizers weren't called when the first transaction ended.
- (3.4a6) Doing a subtransaction commit erroneously processed invalidations, which could lead to an inconsistent view of the database. For example, let T be the transaction of which the subtransaction commit was a part. If T read a persistent object O's state before the subtransaction commit, did not commit new state of its own for O during its subtransaction commit, and O was modified before the subtransaction commit by a different transaction, then the subtransaction commit processed an invalidation for O, and the state T read for O originally was discarded in T. If T went on to access O again, it saw the newly committed (by a different transaction) state for O:

  ```
  o_attr = O.some_attribute
  get_transaction().commit(True)
  assert o_attr == O.some_attribute
  ```

  could fail, and despite that T never modifed O.
- (3.4a4) Transactions now support savepoints. Savepoints allow changes to be periodically checkpointed within a transaction. You can then rollback to a previously created savepoint. See `transaction/savepoint.txt`.
- (3.4a6) A `getBeforeCommitHooks()` method was added. It returns an iterable producing the registered beforeCommit hooks.

- (3.4a6) The `ISynchronizer` interface has a new `newTransaction()` method. This is invoked whenever a transaction manager's `begin()` method is called. (Note that a transaction object's (as opposed to a transaction manager's) `begin()` method is deprecated, and `newTransaction()` is not called when using the deprecated method.)
- (3.4a6) Relatedly, `Connection` implements `ISynchronizer`, and `Connection`'s `afterCompletion()` and `newTransaction()` methods now call `sync()` on the underlying storage (if the underlying storage has such a method), in addition to processing invalidations. The practical implication is that storage synchronization will be done automatically now, whenever a transaction is explicitly started, and after top-level transaction commit or abort. As a result, `Connection.sync()` should virtually never be needed anymore, and will eventually be deprecated.
- (3.4a3) Transaction objects have a new method, `beforeCommitHook(hook, *args, **kws)`. Hook functions registered with a transaction are called at the start of a top-level commit, before any of the work is begun, so a hook function can perform any database operations it likes. See `test_beforeCommitHook()` in `transaction/tests/test_transaction.py` for a tutorial doctest, and the `ITransaction` interface for details. Thanks to Florent Guillaume for contributing code and tests.
- (3.4a3) Clarifications were made to transaction interfaces.

Support for ZODB4 savepoint-aware data managers has been dropped

- (3.4a4) In adding savepoint support, we dropped the attempted support for ZODB4 data managers that support savepoints. We don't think that this will affect anyone.

ZEO

- (3.4a4) The ZODB and ZEO version numbers are now the same. Concretely:

  ```
  import ZODB, ZEO
  assert ZODB.__version__ == ZEO.version
  ```

  no longer fails. If interested, see the README file for details about earlier version numbering schemes.
- (3.4b1) ZConfig version 2.3 adds new socket address types, for smoother default behavior across platforms. The hostname portion of socket-binding-address defaults to an empty string, which acts like INADDR_ANY on Windows and Linux (bind to any interface). The hostname portion of socket-connection-address defaults to "127.0.0.1" (aka "localhost"). In config files, the types of `zeo` section keys `address` and `monitor-address` changed to socket-binding-address, and the type of the `zeoclient` section key `server` changed to socket-connection-address.
- (3.4a4) The default logging setup in `runzeo.py` was broken. It was changed so that running `runzeo.py` from a command line now, and without using a config file, prints output to the console much as ZODB 3.2 did.

ZEO on Windows

Thanks to Mark Hammond for these `runzeo.py` enhancements on Windows:

- (3.4b1) Collector 1788: Repair one of the new features below.
- (3.4a4) A pid file (containing the process id as a decimal string) is created now for a ZEO server started via `runzeo.py`. External programs can read the pid from this file and derive a "signal name" used in a new signal-emulation scheme for Windows. This is only necessary on Windows, but the pid file is created on all platforms that implement `os.getpid()`, as long as the `pid-filename` option is set, or environment variable `INSTANCE_HOME` is defined. The `pid-filename` option can be set in a ZEO config file, or passed as the new `--pid-file` argument to `runzeo.py`.

- (3.4a4) If available, `runzeo.py` now uses Zope's new 'Signal' mechanism for Windows, to implement clean shutdown and log rotation handlers for Windows. Note that the Python in use on the ZEO server must also have the Python Win32 extensions installed for this to be useful.

Tools

- (3.4a4) `fsdump.py` now displays the size (in bytes) of data records. This actually went in several months go, but wasn't noted here at the time. Thanks to Dmitry Vasiliev for contributing code and tests.

FileStorage

- (3.4a9) The `undoLog()` and `undoInfo()` methods almost always returned a wrong number of results, one too many if `last < 0` (the default is such a case), or one too few if `last >= 0`. These have been repaired, new tests were added, and these methods are now documented in `ZODB.interfaces.IStorageUndoable`.
- (3.4a2) A `pdb.set_trace()` call was mistakenly left in method `FileStorage.modifiedInVersion()`.

ZConfig

- (3.4b1) The "standalone" release of ZODB now includes ZConfig version 2.3.

DemoStorage

- (3.4a4) Appropriate implementations of the storage API's `registerDB()` and `new_oid()` methods were added, delegating to the base storage. This was needed to support wrapping a ZEO client storage as a `DemoStorage` base storage, as some new Zope tests want to do.

BaseStorage

- (3.4a4) `new_oid()`'s undocumented `last=` argument was removed. It was used only for internal recursion, and injured code sanity elsewhere because not all storages included it in their `new_oid()`'s signature. Straightening this out required adding `last=` everywhere, or removing it everywhere. Since recursion isn't actually needed, and there was no other use for `last=`, removing it everywhere was the obvious choice.

Tests

- (3.4a3) The various flavors of the `check2ZODBThreads` and `check7ZODBThreads` tests are much less likely to suffer sproadic failures now.
- (3.4a2) The test `checkOldStyleRoot` failed in Zope3, because of an obscure dependence on the `Persistence` package (which Zope3 doesn't use).

ZApplication

- (3.4a8) The file `ZApplication.py` was moved, from ZODB to Zope(2). ZODB and Zope3 don't use it, but Zope2 does.
- (3.4a7) The `__call__` method didn't work if a non-None `connection` string argument was passed. Thanks to Stefan Holek for noticing.

1.13.63 What's new in ZODB3 3.4a1?

Release date: 01-Apr-2005

transaction

- `get_transaction()` is officially deprecated now, and will be removed in ZODB 3.6. Use the `transaction` package instead. For example, instead of:

  ```
  import ZODB
  ...
  get_transaction().commit()
  ```

  do:

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

DB

- There is no longer a hard limit on the number of connections that `DB.open()` will create. In other words, `DB.open()` never blocks anymore waiting for an earlier connection to close, and `DB.open()` always returns a connection now (while it wasn't documented, it was possible for `DB.open()` to return `None` before).

  `pool_size` continues to default to 7, but its meaning has changed: if more than `pool_size` connections are obtained from `DB.open()` and not closed, a warning is logged; if more than twice `pool_size`, a critical problem is logged. `pool_size` should be set to the maximum number of connections from the `DB` instance you expect to have open simultaneously.

  In addition, if a connection obtained from `DB.open()` becomes unreachable without having been explicitly closed, when Python's garbage collection reclaims that connection it no longer counts against the `pool_size` thresholds for logging messages.

  The following optional arguments to `DB.open()` are deprecated: `transaction`, `waitflag`, `force` and `temporary`. If one is specified, its value is ignored, and `DeprecationWarning` is raised. In ZODB 3.6, these optional arguments will be removed.

- Lightweight support for "multi-databases" is implemented. These are collections of named DB objects and associated open Connections, such that the Connection for any DB in the collection can be obtained from a Connection from any other DB in the collection. See the new test file ZODB/tests/multidb.txt for a tutorial doctest. Thanks to Christian Theune for his work on this during the PyCon 2005 ZODB sprint.

ZEO compatibility

There are severe restrictions on using ZEO servers and clients at or after ZODB 3.3 with ZEO servers and clients from ZODB versions before 3.3. See the reworked `Compatibility` section in `README.txt` for details. If possible, it will be easiest to move clients and servers to 3.3+ simultaneously. With care, it's possible to use a 3.3+ ZEO server with pre-3.3 ZEO clients, but not possible to use a pre-3.3 ZEO server with 3.3+ ZEO clients.

BTrees

- A new family of BTree types, in the `IFBTree` module, map signed integers (32 bits) to C floats (also 32 bits). The intended use is to help construct search indices, where, e.g., integer word or document identifiers map to scores of some kind. This is easier than trying to work with scaled integer scores in an IIBTree, and Zope3 has moved to IFBTrees for these purposes in its search code.

FileStorage

- Addded a record iteration protocol to FileStorage. You can use the record iterator to iterate over all current revisions of data pickles in the storage.

  In order to support calling via ZEO, we don't implement this as an actual iterator. An example of using the record iterator protocol is as follows:

  ```
  storage = FileStorage('anexisting.fs')
  next_oid = None
  while True:
      oid, tid, data, next_oid = storage.record_iternext(next_oid)
      # do something with oid, tid and data
      if next_oid is None:
          break
  ```

  The behavior of the iteration protocol is now to iterate over all current records in the database in ascending oid order, although this is not a promise to do so in the future.

Tools

New tool fsoids.py, for heavy debugging of FileStorages; shows all uses of specified oids in the entire database (e.g., suppose oid 0x345620 is missing – did it ever exist? if so, when? who referenced it? when was the last transaction that modified an object that referenced it? which objects did it reference? what kind of object was it?). ZODB/test/testfsoids.py is a tutorial doctest.

fsIndex

Efficient, general implementations of minKey() and maxKey() methods were added. fsIndex is a special hybrid kind of BTree used to implement FileStorage indices. Thanks to Chris McDonough for code and tests.

1.13.64 What's new in ZODB3 3.3.1?

Release date: DD-MMM-2005

Tests

The various flavors of the check2ZODBThreads and check7ZODBThreads tests are much less likely to suffer sproadic failures now.

1.13.65 What's new in ZODB3 3.3.1c1?

Release date: 01-Apr-2005

BTrees

Collector #1734: BTrees conflict resolution leads to index inconsistencies.

Silent data loss could occur due to BTree conflict resolution when one transaction T1 added a new key to a BTree containing at least three buckets, and a concurrent transaction T2 deleted all keys in the bucket to which the new key was added. Conflict resolution then created a bucket containing the newly added key, but the bucket remained isolated, disconnected from the BTree. In other words, the committed BTree didn't contain the new key added by T1. Conflict resolution doesn't have enough information to repair this, so `ConflictError` is now raised in such cases.

ZEO

Repaired subtle race conditions in establishing ZEO connections, both client- and server-side. These account for intermittent cases where ZEO failed to make a connection (or reconnection), accompanied by a log message showing an error caught in `asyncore` and having a traceback ending with:

```
UnpicklingError:  invalid load key, 'Z'.
```

or:

```
ZRPCError:  bad handshake '(K\x00K\x00U\x0fgetAuthProtocol)t.'
```

or:

```
error:  (9, 'Bad file descriptor')
```

or an `AttributeError`.

These were exacerbated when running the test suite, because of an unintended busy loop in the test scaffolding, which could starve the thread trying to make a connection. The ZEO reconnection tests may run much faster now, depending on platform, and should suffer far fewer (if any) intermittent "timed out waiting for storage to connect" failures.

ZEO protocol and compatibility

ZODB 3.3 introduced multiversion concurrency control (MVCC), which required changes to the ZEO protocol. The first 3.3 release should have increased the internal ZEO protocol version number (used by ZEO protocol negotiation when a client connects), but neglected to. This has been repaired.

Compatibility between pre-3.3 and post-3.3 ZEO clients and servers remains very limited. See the newly updated `Compatibility` section in `README.txt` for details.

FileStorage

- The `.store()` and `.restore()` methods didn't update the storage's belief about the largest oid in use when passed an oid larger than the largest oid the storage already knew about. Because `.restore()` in particular is used by `copyTransactionsFrom()`, and by the first stage of ZRS recovery, a large database could be created that believed the only oid in use was oid 0 (the special oid reserved for the root object). In rare cases, it could go on from there assigning duplicate oids to new objects, starting over from oid 1 again. This has been repaired. A new `set_max_oid()` method was added to the `BaseStorage` class so that derived storages can update the largest oid in use in a threadsafe way.
- A FileStorage's index file tried to maintain the index's largest oid as a separate piece of data, incrementally updated over the storage's lifetime. This scheme was more complicated than necessary, so was also more brittle and slower than necessary. It indirectly participated in a rare but critical bug: when a FileStorage was created via `copyTransactionsFrom()`, the "maximum oid" saved in the index file was always 0. Use that FileStorage, and it could then create "new" oids starting over at 0 again, despite that those oids were already in use by old

objects in the database. Packing a FileStorage has no reason to try to update the maximum oid in the index file either, so this kind of damage could (and did) persist even across packing.

The index file's maximum-oid data is ignored now, but is still written out so that `.index` files can be read by older versions of ZODB. Finding the true maximum oid is done now by exploiting that the main index is really a kind of BTree (long ago, this wasn't true), and finding the largest key in a BTree is inexpensive.

- A FileStorage's index file could be updated on disk even if the storage was opened in read-only mode. That bug has been repaired.
- An efficient `maxKey()` implementation was added to class `fsIndex`.

Pickle (in-memory Connection) Cache

You probably never saw this exception:

```
ValueError:  Can not re-register object under a different oid
```

It's been changed to say what it meant:

```
ValueError:  A different object already has the same oid
```

This happens if an attempt is made to add distinct objects to the cache that have the same oid (object identifier). ZODB should never do this, but it's possible for application code to force such an attempt.

PersistentMapping and PersistentList

Backward compatibility code has been added so that the sanest of the ZODB 3.2 dotted paths for `PersistentMapping` and `PersistentList` resolve. These are still preferred:

- `from persistent.list import PersistentList`
- `from persistent.mapping import PersistentMapping`

but these work again too:

- `from ZODB.PersistentList import PersistentList`
- `from ZODB.PersistentMapping import PersistentMapping`

BTrees

The BTrees interface file neglected to document the optional `excludemin` and `excludemax` arguments to the `keys()`, `values()` and `items()` methods. Appropriate changes were merged in from the ZODB4 BTrees interface file.

Tools

- `mkzeoinst.py`'s default port number changed from to 9999 to 8100, to match the example in Zope's `zope.conf`.

fsIndex

An efficient `maxKey()` method was implemented for the `fsIndex` class. This makes it possible to determine the largest oid in a `FileStorage` index efficiently, directly, and reliably, replacing a more delicate scheme that tried to keep track of this by saving an oid high water mark in the index file and incrementally updating it.

1.13.66 What's new in ZODB3 3.3.1a1?

Release date: 11-Jan-2005

ZEO client cache

- Collector 1536: The `cache-size` configuration option for ZEO clients was being ignored. Worse, the client cache size was only one megabyte, much smaller than the advertised default of 20MB. Note that the default is carried over from a time when gigabyte disks were expensive and rare; 20MB is also too small on most modern machines.
- Fixed a nasty bug in cache verification. A persistent ZEO cache uses a disk file, and, when active, has some in-memory data structures too to speed operation. Invalidations processed as part of startup cache verification were reflected in the in-memory data structures, but not correctly in the disk file. So if an object revision was invalidated as part of verification, the object wasn't loaded again before the connection was closed, and the object revision remained in the cache file until the connection was closed, then the next time the cache file was opened it could believe that the stale object revision in the file was actually current.
- Fixed a bug wherein an object removed from the client cache didn't properly mark the file slice it occupied as being available for reuse.

ZEO

Collector 1503: excessive logging. It was possible for a ZEO client to log "waiting for cache verification to finish" messages at a very high rate, producing gigabytes of such messages in short order. `ClientStorage._wait_sync()` was changed to log no more than one such message per 5 minutes.

persistent

Collector #1350: ZODB has a default one-thread-per-connection model, and two threads should never do operations on a single connection simultaneously. However, ZODB can't detect violations, and this happened in an early stage of Zope 2.8 development. The low-level `ghostify()` and `unghostify()` routines in `cPerisistence.c` were changed to give some help in detecting this when it happens. In a debug build, both abort the process if thread interference is detected. This is extreme, but impossible to overlook. In a release build, `unghostify()` raises `SystemError` if thread damage is detected; `ghostify()` ignores the problem in a release build (`ghostify()` is supposed to be so simple that it "can't fail").

ConflictError

New in 3.3, a `ConflictError` exception may attempt to insert the path to the object's class in its message. However, a ZEO server may not have access to application class implementations, and then the attempt by the server to raise `ConflictError` could raise `ImportError` instead while trying to determine the object's class path. This was confusing. The code has been changed to obtain the class path from the object's pickle, without trying to import application modules or classes.

FileStorage

Collector 1581: When an attempt to pack a corrupted `Data.fs` file was made, it was possible for the pack routine to die with a reference to an undefined global while it was trying to raise `CorruptedError`. It raises `CorruptedError`, as it always intended, in these cases now.

Install

The C header file `ring.h` is now installed.

Tools

- `BTrees.check.display()` now displays the oids (if any) of the BTree's or TreeSet's constituent objects.

1.13.67 What's new in ZODB3 3.3?

Release date: 06-Oct-2004

ZEO

The encoding of RPC calls between server and client was being done with protocol 0 ("text mode") pickles, which could require sending four times as many bytes as necessary. Protocol 1 pickles are used now. Thanks to Andreas Jung for the diagnosis and cure.

ZODB/component.xml

`cache-size` parameters were changed from type `integer` to type `byte-size`. This allows you to specify, for example, "`cache-size 20MB`" to get a 20 megabyte cache.

transaction

The deprecation warning for `Transaction.begin()` was changed to point to the caller, instead of to `Transaction.begin()` itself.

Connection

Restored Connection's private `_opened` attribute. This was still referenced by `DB.connectionDebugInfo()`, and Zope 2 calls the latter.

FileStorage

Collector #1517: History tab for ZPT does not work. `FileStorage.history()` was reading the user, description, and extension fields out of the object pickle, due to starting the read at a wrong location. Looked like cut-and-paste repetition of the same bug in `FileStorage.FileIterator` noted in the news for 3.3c1.

1.13.68 What's new in ZODB3 3.3 release candidate 1?

Release date: 14-Sep-2004

Connection

ZODB intends to raise `ConnnectionStateError` if an attempt is made to close a connection while modifications are pending (the connection is involved in a transaction that hasn't been `abort()`'ed or `commit()`'ed). It was missing the case where the only pending modifications were made in subtransactions. This has been fixed. If an attempt to close a connection with pending subtransactions is made now:

```
ConnnectionStateError: Cannot close a connection with a pending subtransaction
```

is raised.

transaction

- Transactions have new, backward-incompatible behavior in one respect: if a `Transaction.commit()`, `Transaction.commit(False)`, or `Transaction.commit(True)` raised an exception, prior behavior was that the transaction effectively aborted, and a new transaction began. A primary bad consequence was that, if in a sequence of subtransaction commits, one of the commits failed but the exception was suppressed, all changes up to and including the failing commit were lost, but later subtransaction commits in the sequence got no indication that something had gone wrong, nor did the final (top level) commit. This could easily lead to inconsistent data being committed, from the application's point of view.

  The new behavior is that a failing commit "sticks" until explicitly cleared. Now if an exception is raised by a `commit()` call (whether subtransaction or top level) on a Transaction object `T`:

  – Pending changes are aborted, exactly as they were for a failing commit before.

  – But `T` remains the current transaction object (if `tm` is `T`'s transaction manger, `tm.get()` continues to return `T`).

  – All subsequent attempts to do `T.commit()`, `T.join()`, or `T.register()` raise the new `TransactionFailedError` exception. Note that if you try to modify a persistent object, that object's resource manager (usually a `Connection` object) will attempt to `join()` the failed transaction, and `TransactionFailedError` will be raised right away.

  So after a transaction or subtransaction commit fails, that must be explicitly cleared now, either by invoking `abort()` on the transaction object, or by invoking `begin()` on its transaction manager.

- Some explanations of new transaction features in the 3.3a3 news were incorrect, and this news file has been retroactively edited to repair that. See news for 3.3a3 below.

- If ReadConflictError was raised by an attempt to load an object with a `_p_independent()` method that returned false, attempting to commit the transaction failed to (re)raise ReadConflictError for that object. Note that ZODB intends to prevent committing a transaction in which a ReadConflictError occurred; this was an obscure case it missed.

- Growing pains: ZODB 3.2 had a bug wherein `Transaction.begin()` didn't abort the current transaction if the only pending changes were in a subtransaction. In ZODB 3.3, it's intended that a transaction manager be used to effect `begin()` (instead of invoking `Transaction.begin()`), and calling `begin()` on a transaction manager didn't have this old bug. However, `Transaction.begin()` still exists in 3.3, and it had a worse bug: it never aborted the transaction (not even if changes were pending outside of subtransactions). `Transaction.begin()` has been changed to abort the transaction. `Transaction.begin()` is also deprecated. Don't use it. Use `begin()` on the relevant transaction manager instead. For example,

  ```
  >>> import transaction
  >>> txn = transaction.begin()  # start a txn using the default TM
  ```

  if using the default `ThreadTransactionManager` (see news for 3.3a3 below). In 3.3, it's intended that a single `Transaction` object is used for exactly one transaction. So, unlike as in 3.2, when somtimes `Transaction` objects were reused across transactions, but sometimes weren't, when you do `Transaction.begin()` in 3.3 a brand new transaction object is created. That's why this use is deprecated. Code of the form:

```
>>> txn = transaction.get()
>>> ...
>>> txn.begin()
>>> ...
>>> txn.commit()
```

can't work as intended in 3.3, because `txn` is no longer the current `Transaction` object the instant `txn.begin()` returns.

BTrees

The BTrees __init__.py file is now just a comment. It had been trying to set up support for (long gone) "int sets", and to import an old version of Zope's Interface package, which doesn't even ship with ZODB. The latter in particular created problems, at least clashing with PythonCAD's Interface package.

POSException

Collector #1488 (TemporaryStorage – going backward in time). This confusion was really due to that the detail on a ConflictError exception didn't make sense. It called the current revision "was", and the old revision "now". The detail is much more informative now. For example, if the exception said:

```
ConflictError: database conflict error (oid 0xcb22,
serial was 0x03441422948b4399, now 0x034414228c3728d5)
```

before, it now says:

```
ConflictError: database conflict error (oid 0xcb22,
serial this txn started with 0x034414228c3728d5 2002-04-14 20:50:32.863000,
serial currently committed 0x03441422948b4399 2002-04-14 20:50:34.815000)
```

ConflictError

The undocumented `get_old_serial()` and `get_new_serial()` methods were swapped (the first returned the new serial, and the second returned the old serial).

Tools

`FileStorage.FileIterator` was confused about how to read a transaction's user and description fields, which caused several tools to display binary gibberish for these values.

`ZODB.utils.oid_repr()` changed to add a leading "0x", and to strip leading zeroes. This is used, e.g., in the detail of a `POSKeyError` exception, to identify the missing oid. Before, the output was ambiguous. For example, oid 17 was displayed as 0000000000000011. As a Python integer, that's octal 9. Or was it meant to be decimal 11? Or was it meant to be hex? Now it displays as 0x11.

fsrefs.py:

When run with `-v`, produced tracebacks for objects whose creation was merely undone. This was confusing. Tracebacks are now produced only if there's "a real" problem loading an oid.

If the current revision of object O refers to an object P whose creation has been undone, this is now identified as a distinct case.

Captured and ignored most attempts to stop it via Ctrl+C. Repaired.

Now makes two passes, so that an accurate report can be given of all invalid references.

`analyze.py` produced spurious "len of unsized object" messages when finding a data record for an object uncreation or version abort. These no longer appear.

`fsdump.py`'s `get_pickle_metadata()` function (which is used by several tools) was confused about what to do when the ZODB pickle started with a pickle `GLOBAL` opcode. It actually loaded the class then, which it intends never to do, leading to stray messages on stdout when the class wasn't available, and leading to a strange return value even when it was available (the repr of the type object was returned as "the module name", and an empty string was returned as "the class name"). This has been repaired.

1.13.69 What's new in ZODB3 3.3 beta 2

Release date: 13-Aug-2004

Transaction Managers

Zope3-dev Collector #139: Memory leak involving buckets and connections

The transaction manager internals effectively made every Connection object immortal, except for those explicitly closed. Since typical practice is not to close connections explicitly (and closing a DB happens not to close the connections to it – although that may change), this caused massive memory leaks when many connections were opened. The transaction manager internals were reworked to use weak references instead, so that connection memory (and other registered synch objects) now get cleaned up when nothing other than the transaction manager knows about them.

Storages

Collector #1327: FileStorage init confused by time travel

If the system clock "went backwards" a long time between the times a FileStorage was closed and reopened, new transaction ids could be smaller than transaction ids already in the storage, violating a key invariant. Now transaction ids are guaranteed to be increasing even when this happens. If time appears to have run backwards at all when a FileStorage is opened, a new message saying so is logged at warning level; if time appears to have run backwards at least 30 minutes, the message is logged at critical level (and you should investigate to find and repair the true cause).

Tools

repozo.py: Thanks to a suggestion from Toby Dickenson, backups (whether incremental or full) are first written to a temp file now, which is fsync'ed at the end, and only after that succeeds is the file renamed to YYYY-MM-DD-HH-MM-SS.ext form. In case of a system crash during a repozo backup, this at least makes it much less likely that a backup file with incomplete or incorrect data will be left behind.

fsrefs.py: Fleshed out the module docstring, and repaired a bug wherein spurious error msgs could be produced after reporting a problem with an unloadable object.

Test suite

Collector #1397: testTimeStamp fails on FreeBSD

> The BSD distributions are unique in that their mktime() implementation usually ignores the input tm_isdst value. Test checkFullTimeStamp() was sensitive to this platform quirk.

Reworked the way some of the ZEO tests use threads, so that unittest is more likely to notice the real cause of a failure (which usually occurs in a thread), and less likely to latch on to spurious problems resulting from the real failure.

1.13.70 What's new in ZODB3 3.3 beta 1

Release date: 07-Jun-2004

3.3b1 is the first ZODB release built using the new zpkg tools:

> http://zope.org/Members/fdrake/zpkgtools/

This appears to have worked very well. The structure of the tarball release differs from previous releases because of it, and the set of installed files includes some that were not installed in previous releases. That shouldn't create problems, so let us know if it does! We'll fine-tune this for the next release.

BTrees

Fixed bug indexing BTreeItems objects with negative indexes. This caused reverse iteration to return each item twice. Thanks to Casey Duncan for the fix.

ZODB

Methods removed from the database (ZODB.DB.DB) class: cacheStatistics(), cacheMeanAge(), cacheMeanDeac(), and cacheMeanDeal(). These were undocumented, untested, and unused. The first always returned an empty tuple, and the rest always returned None.

When trying to do recovery to a time earlier than that of the most recent full backup, repozo.py failed to find the appropriate files, erroneously claiming "No files in repository before <specified time>". This has been repaired.

Collector #1330: repozo.py -R can create corrupt .fs. When looking for the backup files needed to recreate a Data.fs file, repozo could (unintentionally) include its meta .dat files in the list, or random files of any kind created by the user in the backup directory. These would then get copied verbatim into the reconstructed file, filling parts with junk. Repaired by filtering the file list to include only files with the data extensions repozo.py creates (.fs, .fsz, .deltafs, and .deltafsz). Thanks to James Henderson for the diagnosis.

fsrecover.py couldn't work, because it referenced attributes that no longer existed after the MVCC changes. Repaired that, and added new tests to ensure it continues working.

Collector #1309: The reference counts reported by DB.cacheExtremeDetails() for ghosts were one too small. Thanks to Dieter Maurer for the diagnosis.

Collector #1208: Infinite loop in cPickleCache. If a persistent object had a __del__ method (probably not a good idea regardless, but we don't prevent it) that referenced an attribute of self, the code to deactivate objects in the cache could get into an infinite loop: ghostifying the object could lead to calling its __del__ method, the latter would load the object into cache again to satsify the attribute reference, the cache would again decide that the object should be ghostified, and so on. The infinite loop no longer occurs, but note that objects of this kind still aren't sensible (they're effectively immortal). Thanks to Toby Dickenson for suggesting a nice cure.

1.13.71 What's new in ZODB3 3.3 alpha 3

Release date: 16-Apr-2004

transaction

There is a new transaction package, which provides new interfaces for application code and for the interaction between transactions and resource managers.

The top-level transaction package has functions `commit()`, `abort()`, `get()`, and `begin()`. They should be used instead of the magic `get_transaction()` builtin, which will be deprecated. For example:

```
>>> get_transaction().commit()
```

should now be written as

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

The new API provides explicit transaction manager objects. A transaction manager (TM) is responsible for associating resource managers with a "current" transaction. The default TM, implemented by class `ThreadedTransactionManager`, assigns each thread its own current transaction. This default TM is available as `transaction.manager`. The `TransactionManager` class assigns all threads to the same transaction, and is an explicit replacement for the `Connection.setLocalTransaction()` method:

A transaction manager instance can be passed as the transaction_manager argument to `DB.open()`. If you do, the connection will use the specified transaction manager instead of the default TM. The current transaction is obtained by calling `get()` on a TM. For example:

```
>>> tm = transaction.TransactionManager()
>>> cn = db.open(transaction_manager=tm)
[...]
>>> tm.get().commit()
```

The `setLocalTransaction()` and `getTransaction()` methods of Connection are deprecated. Use an explicit TM passed via `transaction_manager=` to `DB.open()` instead. The `setLocalTransaction()` method still works, but it returns a TM instead of a Transaction.

A TM creates Transaction objects, which are used for exactly one transaction. Transaction objects still have `commit()`, `abort()`, `note()`, `setUser()`, and `setExtendedInfo()` methods.

Resource managers, e.g. Connection or RDB adapter, should use a Transaction's `join()` method instead of its `register()` method. An object that calls `join()` manages its own resources. An object that calls `register()` expects the TM to manage the objects.

Data managers written against the ZODB 4 transaction API are now supported in ZODB 3.

persistent

A database can now contain persistent weak references. An object that is only reachable from persistent weak references will be removed by pack().

The persistence API now distinguishes between deactivation and invalidation. This change is intended to support objects that can't be ghosts, like persistent classes. Deactivation occurs when a user calls _p_deactivate() or when the cache evicts objects because it is full. Invalidation occurs when a transaction updates the object. An object that can't be a ghost must load new state when it is invalidated, but can ignore deactivation.

Persistent objects can implement a __getnewargs__() method that will be used to provide arguments that should be passed to __new__() when instances (including ghosts) are created. An object that implements __getnewargs__() must be loaded from storage even to create a ghost.

There is new support for writing hooks like __getattr__ and __getattribute__. The new hooks require that user code call special persistence methods like _p_getattr() inside their hook. See the ZODB programming guide for details.

The format of serialized persistent references has changed; that is, the on-disk format for references has changed. The old format is still supported, but earlier versions of ZODB will not be able to read the new format.

ZODB

Closing a ZODB Connection while it is registered with a transaction, e.g. has pending modifications, will raise a ConnnectionStateError. Trying to load objects from or store objects to a closed connection will also raise a ConnnectionStateError.

ZODB connections are synchronized on commit, even when they didn't modify objects. This feature assumes that the thread that opened the connection is also the thread that uses it. If not, this feature will cause problems. It can be disabled by passing synch=False to open().

New broken object support.

New add() method on Connection. User code should not assign the _p_jar attribute of a new persistent object directly; a deprecation warning is issued in this case.

Added a get() method to Connection as a preferred synonym for __getitem__().

Several methods and/or specific optional arguments of methods have been deprecated. The cache_deactivate_after argument used by DB() and Connection() is deprecated. The DB methods getCacheDeactivateAfter(), getVersionCacheDeactivateAfter(), setCacheDeactivateAfter(), and setVersionCacheDeactivateAfter() are also deprecated.

The old-style undo() method was removed from the storage API, and transactionalUndo() was renamed to undo().

The BDBStorages are no longer distributed with ZODB.

Fixed a serious bug in the new pack implementation. If pack was called on the storage and passed a time earlier than a previous pack time, data could be lost. In other words, if there are any two pack calls, where the time argument passed to the second call was earlier than the first call, data loss could occur. The bug was fixed by causing the second call to raise a StorageError before performing any work.

Fixed a rare bug in pack: if a pack started during a small window of time near the end of a concurrent transaction's commit, it was possible for the pack attempt to raise a spurious

> CorruptedError: … transaction with checkpoint flag set

exception. This did no damage to the database, or to the transaction in progress, but no pack was performed then.

By popular demand, FileStorage.pack() no longer propagates a

> FileStorageError: The database has already been packed to a later time or no changes have been made since the last pack

exception. Instead that message is logged (at INFO level), and the pack attempt simply returns then (no pack is performed).

ZEO

Fixed a bug that prevented the -m / –monitor argument from working.

zdaemon

Added a -m / –mask option that controls the umask of the subprocess.

zLOG

The zLOG backend has been removed. zLOG is now just a facade over the standard Python logging package. Environment variables like STUPID_LOG_FILE are no longer honored. To configure logging, you need to follow the directions in the logging package documentation. The process is currently more complicated than configured zLOG. See test.py for an example.

ZConfig

This release of ZODB contains ZConfig 2.1.

More documentation has been written.

Make sure keys specified as attributes of the <default> element are converted by the appropriate key type, and are re-checked for derived sections.

Refactored the ZConfig.components.logger schema components so that a schema can import just one of the "eventlog" or "logger" sections if desired. This can be helpful to avoid naming conflicts.

Added a reopen() method to the logger factories.

Always use an absolute pathname when opening a FileHandler.

Miscellaneous

The layout of the ZODB source release has changed. All the source code is contained in a src subdirectory. The primary motivation for this change was to avoid confusion caused by installing ZODB and then testing it interactively from the source directory; the interpreter would find the uncompiled ZODB package in the source directory and report an import error.

A reference-counting bug was fixed, in the logic calling a modified persistent object's data manager's register() method. The primary symptom was rare assertion failures in Python's cyclic garbage collection.

The Connection class's onCommitAction() method was removed.

Some of the doc strings in ZODB are now written for processing by epydoc.

Several new test suites were written using doctest instead of the standard unittest TestCase framework.

MappingStorage now implements getTid().

ThreadedAsync: Provide a way to shutdown the servers using an exit status.

The mkzeoinstance script looks for a ZODB installation, not a Zope installation. The received wisdom is that running a ZEO server without access to the appserver code avoids many mysterious problems.

1.13.72 What's new in ZODB3 3.3 alpha 2

Release date: 06-Jan-2004

This release contains a major overhaul of the persistence machinery, including some user-visible changes. The Persistent base class is now a new-style class instead of an ExtensionClass. The change enables the use of features like properties with persistent object classes. The Persistent base class is now contained in the persistent package.

The Persistence package is included for backwards compatibility. The Persistence package is used by Zope to provide special ExtensionClass-compatibility features like a non-C3 MRO and an __of__ method. ExtensionClass is not included with this release of ZODB3. If you use the Persistence package, it will print a warning and import Persistent from persistent.

In short, the new persistent package is recommended for non-Zope applications. The following dotted class names are now preferred over earlier names:

- persistent.Persistent
- persistent.list.PersistentList
- persistent.mapping.PersistentMapping
- persistent.TimeStamp

The in-memory, per-connection object cache (pickle cache) was changed to participate in garbage collection. This should reduce the number of memory leaks, although we are still tracking a few problems.

Multi-version concurrency control

ZODB now supports multi-version concurrency control (MVCC) for storages that support multiple revisions. FileStorage and BDBFullStorage both support MVCC. In short, MVCC means that read conflicts should almost never occur. When an object is modified in one transaction, other concurrent transactions read old revisions of the object to preserve consistency. In earlier versions of ZODB, any access of the modified object would raise a ReadConflictError.

The ZODB internals changed significantly to accommodate MVCC. There are relatively few user visible changes, aside from the lack of read conflicts. It is possible to disable the MVCC feature using the mvcc keyword argument to the DB open() method, ex.: db.open(mvcc=False).

ZEO

Changed the ZEO server and control process to work with a single configuration file; this is now the default way to configure these processes. (It's still possible to use separate configuration files.) The ZEO configuration file can now include a "runner" section used by the control process and ignored by the ZEO server process itself. If present, the control process can use the same configuration file.

Fixed a performance problem in the logging code for the ZEO protocol. The logging code could call repr() on arbitrarily long lists, even though it only logged the first 60 bytes; worse, it called repr() even if logging was currently disabled. Fixed to call repr() on individual elements until the limit is reached.

Fixed a bug in zrpc (when using authentication) where the MAC header wasn't being read for large messages, generating errors while unpickling commands sent over the wire. Also fixed the zeopasswd.py script, added testcases and provided a more complete commandline interface.

Fixed a misuse of the _map variable in zrpc Connectio objects, which are also asyncore.dispatcher objects. This allows ZEO to work with CVS Python (2.4). _map is used to indicate whether the dispatcher users the default socket_map or a custom socket_map. A recent change to asyncore caused it to use _map in its add_channel() and del_channel() methods, which presumes to be a bug fix (may get ported to 2.3). That causes our dubious use of _map to be a problem, because we also put the Connections in the global socket_map. The new asyncore won't remove it from the global socket map, because it has a custom _map.

The prefix used for log messages from runzeo.py was changed from RUNSVR to RUNZEO.

Miscellaneous

ReadConflictError objects now have an ignore() method. Normally, a transaction that causes a read conflict can't be committed. If the exception is caught and its ignore() method called, the transaction can be committed. Application code may need this in advanced applications.

1.13.73 What's new in ZODB3 3.3 alpha 1

Release date: 17-Jul-2003

New features of Persistence

The Persistent base class is a regular Python type implemented in C. It should be possible to create new-style classes that inherit from Persistent, and, thus, use all the new Python features introduced in Python 2.2 and 2.3.

The __changed__() method on Persistent objects is no longer supported.

New features in BTrees

BTree, Bucket, TreeSet and Set objects are now iterable objects, playing nicely with the iteration protocol introduced in Python 2.2, and can be used in any context that accepts an iterable object. As for Python dicts, the iterator constructed for BTrees and Buckets iterates over the keys.

```
>>> from BTrees.OOBTree import OOBTree
>>> b = OOBTree({"one": 1, "two": 2, "three": 3, "four": 4})
>>> for key in b: # iterates over the keys
...     print key
four
one
three
two
>>> list(enumerate(b))
[(0, 'four'), (1, 'one'), (2, 'three'), (3, 'two')]
>>> i = iter(b)
>>> i.next()
'four'
>>> i.next()
'one'
>>> i.next()
'three'
>>> i.next()
'two'
>>>
```

As for Python dicts in 2.2, BTree and Bucket objects have new .iterkeys(), .iteritems(), and .itervalues() methods. TreeSet and Set objects have a new .iterkeys() method. Unlike as for Python dicts, these new methods accept optional min and max arguments to effect range searches. While Bucket.keys() produces a list, Bucket.iterkeys() produces an iterator, and similarly for Bucket values() versus itervalues(), Bucket items() versus iteritems(), and Set keys() versus iterkeys(). The iter{keys,values,items} methods of BTrees and the iterkeys() method of Treesets also produce iterators, while their keys() (etc) methods continue to produce BTreeItems objects (a form of "lazy" iterator that predates Python 2.2's iteration protocol).

```
>>> sum(b.itervalues())
10
>>> zip(b.itervalues(), b.iterkeys())
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
>>>
```

BTree, Bucket, TreeSet and Set objects also implement the __contains__ method new in Python 2.2, which means that testing for key membership can be done directly now via the "in" and "not in" operators:

```
>>> "won" in b
False
>>> "won" not in b
True
>>> "one" in b
True
>>>
```

All old and new range-search methods now accept keyword arguments, and new optional excludemin and excludemax keyword arguments. The new keyword arguments allow doing a range search that's exclusive at one or both ends (doesn't include min, and/or doesn't include max).

```
>>> list(b.keys())
['four', 'one', 'three', 'two']
>>> list(b.keys(max='three'))
['four', 'one', 'three']
>>> list(b.keys(max='three', excludemax=True))
['four', 'one']
>>>
```

Other improvements

The exceptions generated by write conflicts now contain the name of the conflicted object's class. This feature requires support for the storage. All the standard storages support it.

1.13.74 What's new in ZODB3 3.2

Release date: 08-Oct-2003

Nothing has changed since release candidate 1.

1.13.75 What's new in ZODB3 3.2 release candidate 1

Release date: 01-Oct-2003

Added a summary to the Doc directory. There are several new documents in the 3.2 release, including "Using zdctl and zdrun to manage server processes" and "Running a ZEO Server HOWTO."

Fixed ZEO's protocol negotiation mechanism so that a client ZODB 3.1 can talk to a ZODB 3.2 server.

Fixed a memory leak in the ZEO server. The server was leaking a few KB of memory per connection.

Fixed a memory leak in the ZODB object cache (cPickleCache). The cache did not release two references to its Connection, causing a large cycle of objects to leak when a database was closed.

Fixed a bug in the ZEO code that caused it to leak socket objects on Windows. Specifically, fix the trigger mechanism so that both sockets created for a trigger are closed.

Fixed a bug in the ZEO storage server that caused it to leave temp files behind. The CommitLog class contains a temp file, but it was not closing the file.

Changed the order of setuid() and setgid() calls in zdrun, so that setgid() is called first.

Added a timeout to the ZEO test suite that prevents hangs. The test suite creates ZEO servers with randomly assigned ports. If the port happens to be in use, the test suite would hang because the ZEO client would never stop trying to connect. The fix will cause the test to fail after a minute, but should prevent the test runner from hanging.

The logging package was updated to include the latest version of the logging package from Python CVS. Note that this package is only installed for Python 2.2. In later versions of Python, it is available in the Python standard library.

The ZEO1 directory was removed from the source distribution. ZEO1 is not supported, and we never intended to include it in the release.

1.13.76 What's new in ZODB3 3.2 beta 3

Release date: 23-Sep-2003

Note: The changes listed for this release include changes also made in ZODB 3.1.x releases and ported to the 3.2 release.

This version of ZODB 3.2 is not compatible with Python 2.1. Early versions were explicitly designed to be compatible with Zope 2.6. That plan has been dropped, because Zope 2.7 is already in beta release.

Several of the classes in ZEO and ZODB now inherit from object, making them new-style classes. The primary motivation for the change was to make it easier to debug memory leaks. We don't expect any behavior to change as a result.

A new feature to allow removal of connection pools for versions was ported from Zope 2.6. This feature is needed by Zope to avoid denial of service attacks that allow a client to create an arbitrary number of version pools.

Fixed several critical ZEO bugs.

- If several client transactions were blocked waiting for the storage and one of the blocked clients disconnected, the server would attempt to restart one of the other waiting clients. Since the disconnected client did not have the storage lock, this could lead to deadlock. It could also cause the assertion "self._client is None" to fail.
- If a storage server fails or times out between the vote and the finish, the ZEO cache could get populated with objects that didn't make it to the storage server.
- If a client loses its connection to the server near the end of a transaction, it is now guaranteed to get a ClientDisconnected error even if it reconnects before the transaction finishes. This is necessary because the server will always abort the transaction. In some cases, the client would never see an error for the aborted transaction.
- In tpc_finish(), reordered the calls so that the server's tpc_finish() is called (and must succeed) before we update the ZEO client cache.
- The storage name is now prepended to the sort key, to ensure a unique global sort order if storages are named uniquely. This can prevent deadlock in some unusual cases.

Fixed several serious flaws in the implementation of the ZEO authentication protocol.

- The smac layer would accept a message without a MAC even after the session key was established.
- The client never initialized its session key, so it never checked incoming messages or created MACs for outgoing messags.
- The smac layer used a single HMAC instance for sending and receiving messages. This approach could only work if client and server were guaranteed to process all messages in the same total order, which could only happen in simple scenarios like unit tests.

Fixed a bug in ExtensionClass when comparing ExtensionClass instances. The code could raise RuntimeWarning under Python 2.3, and produce incorrect results on 64-bit platforms.

Fixed bug in BDBStorage that could lead to DBRunRecoveryErrors when a transaction was aborted after performing operations like commit version or undo that create new references to existing pickles.

Fixed a bug in Connection.py that caused it to fail with an AttributeError if close() was called after the database was closed.

The test suite leaves fewer log files behind, although it still leaves a lot of junk. The test.py script puts each tests temp files in a separate directory, so it is easier to see which tests are causing problems. Unfortunately, it is still to tedious to figure out why the identified tests are leaving files behind.

This release contains the latest and greatest version of the BDBStorage. This storage has still not seen testing in a production environment, but it represents the current best design and most recent code culled from various branches where development has occurred.

The Tools directory contains a number of small improvements, a few new tools, and README.txt that catalogs the tools. Many of the tools are installed by setup.py; those scripts will now have a #! line set automatically on Unix.

Fixed bugs in Tools/repozo.py, including a timing-dependent one that could cause the following invocation of repozo to do a full backup when an incremental backup would have sufficed.

A pair of new scripts from Jim Fulton can be used to synthesize workloads and measure ZEO performance: see zodbload.py and zeoserverlog.py in the Tools directory. Note that these require Zope.

Tools/checkbtrees.py was strengthened in two ways:

- In addition to running the _check() method on each BTree B found, BTrees.check.check(B) is also run. The check() function was written after checkbtrees.py, and identifies kinds of damage B._check() cannot find.
- Cycles in the object graph no longer lead to unbounded output. Note that preventing this requires remembering the oid of each persistent object found, which increases the memory needed by the script.

1.13.77 What's new in ZODB3 3.2 beta 2

Release date: 16-Jun-2003

Fixed critical race conditions in ZEO's cache consistency code that could cause invalidations to be lost or stale data to be written to the cache. These bugs can lead to data loss or data corruption. These bugs are relatively unlikely to be provoked in sites with few conflicts, but the possibility of failure existed any time an object was loaded and stored concurrently.

Fixed a bug in conflict resolution that failed to ghostify an object if it was involved in a conflict. (This code may be redundant, but it has been fixed regardless.)

The ZEO server was fixed so that it does not perform any I/O until all of a transactions' invalidations are queued. If it performs I/O in the middle of sending invalidations, it would be possible to overlap a load from a client with the invalidation being sent to it.

The ZEO cache now handles invalidations atomically. This is the same sort of bug that is described in the 3.1.2b1 section below, but it affects the ZEO cache.

Fixed several serious bugs in fsrecover that caused it to fail catastrophically in certain cases because it thought it had found a checkpoint (status "c") record when it was in the middle of the file.

Two new features snuck into this beta release.

The ZODB.transact module provides a helper function that converts a regular function or method into a transactional one.

The ZEO client cache now supports Adaptable Persistence (APE). The cache used to expect that all OIDs were eight bytes long.

1.13.78 What's new in ZODB3 3.2 beta 1

Release date: 30-May-2003

ZODB

Invalidations are now processed atomically. Each transaction will see all the changes caused by an earlier transaction or none of them. Before this patch, it was possible for a transaction to see invalid data because it saw only a subset of the invalidations. This is the most likely cause of reported BTrees corruption, where keys were stored in the wrong bucket. When a BTree bucket splits, the bucket and the bucket's parent are both modified. If a transaction sees the invalidation for the bucket but not the parent, the BTree in memory will be internally inconsistent and keys can be put in the wrong bucket. The atomic invalidation fix prevents this problem.

A number of minor reference count fixes in the object cache were fixed. That's the cPickleCache.c file.

It was possible for a transaction that failed in tpc_finish() to lose the traceback that caused the failure. The transaction code was fixed to report the original error as well as any errors that occur while trying to recover from the original error.

The "other" argument to copyTransactionsFrom() only needs to have an .iterator() method. For convenience, change FileStorage's and BDBFullStorage's iterator to have this method, which just returns self.

Mount points are now visible from mounted objects.

Fixed memory leak involving database connections and caches. When a connection or database was closed, the cache and database leaked, because of a circular reference involving the cache. Fixed the cache to explicitly clear out its contents when its connection is closed.

The ZODB cache has fewer methods. It used to expose methods that could mutate the dictionary, which allowed users to violate internal invariants.

ZConfig

It is now possible to configure ZODB databases and storages and ZEO servers using ZConfig.

ZEO & zdaemon

ZEO now supports authenticated client connections. The default authentication protocol uses a hash-based challenge-response protocol to prove identity and establish a session key for message authentication. The architecture is pluggable to allow third-parties to developer better authentication protocols.

There is a new HOWTO for running a ZEO server. The draft in this release is incomplete, but provides more guidance than previous releases. See the file Doc/ZEO/howto.txt.

The ZEO storage server's transaction timeout feature was refactored and made slightly more rebust.

A new ZEO utility script, ZEO/mkzeoinst.py, was added. This creates a standard directory structure and writes a configuration file with mostly default values, and a bootstrap script that can be used to manage and monitor the server using zdctl.py (see below).

Much work was done to improve zdaemon's zdctl.py and zdrun.py scripts. (In the alpha 1 release, zdrun.py was called zdaemon.py, but installing it in <prefix>/bin caused much breakage due to the name conflict with the zdaemon package.) Together with the new mkzeoinst.py script, this makes controlling a ZEO server a breeze.

A ZEO client will not read from its cache during cache verification. This fix was necessary to prevent the client from reading inconsistent data.

The isReadOnly() method of a ZEO client was fixed to return the false when the client is connected to a read-only fallback server.

The sync() method of ClientStorage and the pending() method of a zrpc connection now do both input and output.

The short_repr() function used to generate log messages was fixed so that it does not blow up creating a repr of very long tuples.

Storages

FileStorage has a new pack() implementation that fixes several reported problems that could lead to data loss.

Two small bugs were fixed in DemoStorage. undoLog() did not handle its arguments correctly and pack() could accidentally delete objects created in versions.

Fixed trivial bug in fsrecover that prevented it from working at all.

FileStorage will use fsync() on Windows starting with Python 2.2.3.

FileStorage's commit version was fixed. It used to stop after the first object, leaving all the other objects in the version.

BTrees

Trying to store an object of a non-integer type into an IIBTree or OIBTree could leave the bucket in a variety of insane states. For example, trying

> b[obj] = "I'm a string, not an integer"

where b is an OIBTree. This manifested as a refcount leak in the test suite, but could have been much worse (most likely in real life is that a seemingly arbitrary existing key would "go missing").

When deleting the first child of a BTree node with more than one child, a reference to the second child leaked. This could cause the entire bucket chain to leak (not be collected as garbage despite not being referenced anymore).

Other minor BTree leak scenarios were also fixed.

Tools

New tool zeoqueue.py for parsing ZEO log files, looking for blocked transactions.

New tool repozo.py (originally by Anthony Baxter) for performing incremental backups of Data.fs files.

The fsrecover.py script now does a better job of recovering from errors the occur in the middle of a transaction record. Fixed several bugs that caused partial or total failures in earlier versions.

1.13.79 What's new in ZODB3 3.2 alpha 1

Release date: 17-Jan-2003

Most of the changes in this release are performance and stability improvements to ZEO. A major packaging change is that there won't be a separate ZEO release. The new ZConfig is a noteworthy addtion (see below).

ZODB

An experimental new transaction API was added. The Connection class has a new method, setLocalTransaction(). ZODB applications can call this method to bind transactions to connections rather than threads. This is especially useful for GUI applications, which often have only one thread but multiple independent activities within that thread (generally one per window). Thanks to Christian Reis for championing this feature.

Applications that take advantage of this feature should not use the get_transaction() function. Until now, ZODB itself sometimes assumed get_transaction() was the only way to get the transaction. Minor corrections have been added. The ZODB test suite, on the other hand, can continue to use get_transaction(), since it is free to assume that transactions are bound to threads.

ZEO

There is a new recommended script for starting a storage server. We recommend using ZEO/runzeo.py instead of ZEO/start.py. The start.py script is still available in this release, but it will no longer be maintained and will eventually be removed.

There is a new zdaemon implementation. This version is a separate script that runs an arbitrary daemon. To run the ZEO server as a daemon, you would run "zdrun.py runzeo.py". There is also a simple shell, zdctl.py, that can be used to manage a daemon. Try "zdctl.py -p runzeo.py".

There is a new version of the ZEO protocol in this release and a first stab at protocol negotiation. (It's a first stab because the protocol checking supporting in ZODB 3.1 was too primitive to support anything better.) A ZODB 3.2 ZEO client can talk to an old server, but a ZODB 3.2 server can't talk to an old client. It's safe to upgrade all the clients first and upgrade the server last. The ZEO client cache format changed, so you'll need to delete persistent caches before restarting clients.

The ZEO cache verification protocol was revised to require many fewer messages in cases where a client or server restarts quickly.

The performance of full cache verification has improved dramatically. Measurements from Jim were somewhere in 2x-5x. The implementation was fixed to use the very-fast getSerial() method on the storage instead of the comparatively slow load().

The ZEO server has an optional timeout feature that will abort a connection that does not commit within a certain amount of time. The timeout works by closing the socket the client is using, causing both client and server to abort the transaction and continue. This is a drastic step, but can be useful to prevent a hung client or other bug from blocking a server indefinitely.

A bug was fixed in the ZEO protocol that allowed clients to read stale cache data while cache verification was being performed. The fixed version prevents the client from using the storage until after verification completes.

The ZEO server has an experimental monitoring interface that reports usage statistics for the storage server including number of connected clients and number of transactions active and committed. It can be enabled by passing the -m flag to runsvr.py.

The ZEO ClientStorage no longer supports the environment variables CLIENT_HOME, INSTANCE_HOME, or ZEO_CLIENT.

The ZEO1 package is still included with this release, but there is no longer an option to install it.

BTrees

The BTrees package now has a check module that inspects a BTree to check internal invariants. Bugs in older versions of the code code leave a BTree in an inconsistent state. Calling BTrees.check.check() on a BTree object should verify its consistency. (See the NEWS section for 3.1 beta 1 below to for the old BTrees bugs.)

Fixed a rare conflict resolution problem in the BTrees that could cause an segfault when the conflict resolution resulted in any empty bucket.

Installation

The distutils setup now installs several Python scripts. The runzeo.py and zdrun.py scripts mentioned above and several fsXXX.py scripts from the Tools directory.

The test.py script does not run all the ZEO tests by default, because the ZEO tests take a long time to run. Use –all to run all the tests. Otherwise a subset of the tests, mostly using MappingStorage, are run.

Storages

There are two new storages based on Sleepycat's BerkeleyDB in the BDBStorage package. Barry will have to write more here, because I don't know how different they are from the old bsddb3Storage storages. See Doc/BDBStorage.txt for more information.

It now takes less time to open an existing FileStorage. The FileStorage uses a BTree-based index that is faster to pickle and unpickle. It also saves the index periodically so that subsequent opens will go fast even if the storage was not closed cleanly.

Misc

The new ZConfig package, which will be used by Zope and ZODB, is included. ZConfig provides a configuration syntax, similar to Apache's syntax. The package can be used to configure the ZEO server and ZODB databases. See the module ZODB.config for functions to open the database from configuration. See ZConfig/doc for more info.

The zLOG package now uses the logging package by Vinay Sajip, which will be included in Python 2.3.

The Sync extension was removed from ExtensionClass, because it was not used by ZODB.

1.13.80 What's new in ZODB3 3.1.4?

Release date: 11-Sep-2003

A new feature to allow removal of connection pools for versions was ported from Zope 2.6. This feature is needed by Zope to avoid denial of service attacks that allow a client to create an arbitrary number of version pools.

A pair of new scripts from Jim Fulton can be used to synthesize workloads and measure ZEO performance: see zodbload.py and zeoserverlog.py in the Tools directory. Note that these require Zope.

Tools/checkbtrees.py was strengthened in two ways:

- In addition to running the _check() method on each BTree B found, BTrees.check.check(B) is also run. The check() function was written after checkbtrees.py, and identifies kinds of damage B._check() cannot find.
- Cycles in the object graph no longer lead to unbounded output. Note that preventing this requires remembering the oid of each persistent object found, which increases the memory needed by the script.

1.13.81 What's new in ZODB3 3.1.3?

Release date: 18-Aug-2003

Fixed several critical ZEO bugs.

- If a storage server fails or times out between the vote and the finish, the ZEO cache could get populated with objects that didn't make it to the storage server.
- If a client loses its connection to the server near the end of a transaction, it is now guaranteed to get a ClientDisconnected error even if it reconnects before the transaction finishes. This is necessary because the server will always abort the transaction. In some cases, the client would never see an error for the aborted transaction.
- In tpc_finish(), reordered the calls so that the server's tpc_finish() is called (and must succeed) before we update the ZEO client cache.
- The storage name is now prepended to the sort key, to ensure a unique global sort order if storages are named uniquely. This can prevent deadlock in some unusual cases.

A variety of fixes and improvements to Berkeley storage (aka BDBStorage) were back-ported from ZODB 4. This release now contains the most current version of the Berkeley storage code. Many tests have been back-ported, but not all.

Modified the Windows tests to wait longer at the end of ZEO tests for the server to shut down. Before Python 2.3, there is no waitpid() on Windows, and, thus, no way to know if the server has shut down. The change makes the Windows ZEO tests much less likely to fail or hang, at the cost of increasing the time needed to run the tests.

Fixed a bug in ExtensionClass when comparing ExtensionClass instances. The code could raise RuntimeWarning under Python 2.3, and produce incorrect results on 64-bit platforms.

Fixed bugs in Tools/repozo.py, including a timing-dependent one that could cause the following invocation of repozo to do a full backup when an incremental backup would have sufficed.

Added Tools/README.txt that explains what each of the scripts in the Tools directory does.

There were many small changes and improvements to the test suite.

1.13.82 What's new in ZODB3 3.1.2 final?

Fixed bug in FileStorage pack that caused it to fail if it encountered an old undo record (status "u").

Fixed several bugs in FileStorage pack that could cause OverflowErrors for storages > 2 GB.

Fixed memory leak in TimeStamp.laterThan() that only occurred when it had to create a new TimeStamp.

Fixed two BTree bugs that were fixed on the head a while ago:

- bug in fsBTree that would cause byValue searches to end early. (fsBTrees are never used this way, but it was still a bug.)
- bug that lead to segfault if BTree was mutated via deletion while it was being iterated over.

1.13.83 What's new in ZODB3 3.1.2 beta 2?

Fixed critical race conditions in ZEO's cache consistency code that could cause invalidations to be lost or stale data to be written to the cache. These bugs can lead to data loss or data corruption. These bugs are relatively unlikely to be provoked in sites with few conflicts, but the possibility of failure existed any time an object was loaded and stored concurrently.

Fixed a bug in conflict resolution that failed to ghostify an object if it was involved in a conflict. (This code may be redundant, but it has been fixed regardless.)

The ZEO server was fixed so that it does not perform any I/O until all of a transactions' invalidations are queued. If it performs I/O in the middle of sending invalidations, it would be possible to overlap a load from a client with the invalidation being sent to it.

The ZEO cache now handles invalidations atomically. This is the same sort of bug that is described in the 3.1.2b1 section below, but it affects the ZEO cache.

Fixed several serious bugs in fsrecover that caused it to fail catastrophically in certain cases because it thought it had found a checkpoint (status "c") record when it was in the middle of the file.

1.13.84 What's new in ZODB3 3.1.2 beta 1?

ZODB

Invalidations are now processed atomically. Each transaction will see all the changes caused by an earlier transaction or none of them. Before this patch, it was possible for a transaction to see invalid data because it saw only a subset of the invalidations. This is the most likely cause of reported BTrees corruption, where keys were stored in the wrong bucket. When a BTree bucket splits, the bucket and the bucket's parent are both modified. If a transaction sees the invalidation for the bucket but not the parent, the BTree in memory will be internally inconsistent and keys can be put in the wrong bucket. The atomic invalidation fix prevents this problem.

A number of minor reference count fixes in the object cache were fixed. That's the cPickleCache.c file.

It was possible for a transaction that failed in tpc_finish() to lose the traceback that caused the failure. The transaction code was fixed to report the original error as well as any errors that occur while trying to recover from the original error.

ZEO

A ZEO client will not read from its cache during cache verification. This fix was necessary to prevent the client from reading inconsistent data.

The isReadOnly() method of a ZEO client was fixed to return the false when the client is connected to a read-only fallback server.

The sync() method of ClientStorage and the pending() method of a zrpc connection now do both input and output.

The short_repr() function used to generate log messages was fixed so that it does not blow up creating a repr of very long tuples.

Storages

FileStorage has a new pack() implementation that fixes several reported problems that could lead to data loss.

Two small bugs were fixed in DemoStorage. undoLog() did not handle its arguments correctly and pack() could accidentally delete objects created in versions.

Fixed trivial bug in fsrecover that prevented it from working at all.

FileStorage will use fsync() on Windows starting with Python 2.2.3.

FileStorage's commit version was fixed. It used to stop after the first object, leaving all the other objects in the version.

BTrees

Trying to store an object of a non-integer type into an IIBTree or OIBTree could leave the bucket in a variety of insane states. For example, trying

> b[obj] = "I'm a string, not an integer"

where b is an OIBTree. This manifested as a refcount leak in the test suite, but could have been much worse (most likely in real life is that a seemingly arbitrary existing key would "go missing").

When deleting the first child of a BTree node with more than one child, a reference to the second child leaked. This could cause the entire bucket chain to leak (not be collected as garbage despite not being referenced anymore).

Other minor BTree leak scenarios were also fixed.

Other

Comparing a Missing.Value object to a C type that provide its own comparison operation could lead to a segfault when the Missing.Value was on the right-hand side of the comparison operator. The Missing class was fixed so that its coercion and comparison operations are safe.

Tools

Four tools are now installed by setup.py: fsdump.py, fstest.py, repozo.py, and zeopack.py.

1.13.85 What's new in ZODB3 3.1.1 final?

Release date: 11-Feb-2003

Tools

Updated repozo.py tool

1.13.86 What's new in ZODB3 3.1.1 beta 2?

Release date: 03-Feb-2003

The Transaction "hosed" feature is disabled in this release. If a transaction fails during the tpc_finish() it is not possible, in general, to know whether the storage is in a consistent state. For example, a ZEO server may commit the data and then fail before sending confirmation of the commit to the client. If multiple storages are involved in a transaction, the problem is exacerbated: One storage may commit the data while another fails to commit. In previous versions of ZODB, the database would set a global "hosed" flag that prevented any other transaction from committing until an administrator could check the status of the various failed storages and ensure that the database is in a consistent state. This approach favors data consistency over availability. The new approach is to log a panic but continue. In practice, availability seems to be more important than consistency. The failure mode is exceedingly rare in either case.

The BTrees-based fsIndex for FileStorage is enabled. This version of the index is faster to load and store via pickle and uses less memory to store keys. We had intended to enable this feature in an earlier release, but failed to actually do it; thus, it's getting enabled as a bug fix now.

Two rare bugs were fixed in BTrees conflict resolution. The most probable symptom of the bug would have been a segfault. The bugs were found via synthetic stress tests rather than bug reports.

A value-based consistency checker for BTrees was added. See the module BTrees.check for the checker and other utilities for working with BTrees.

A new script called repozo.py was added. This script, originally written by Anthony Baxter, provides an incremental backup scheme for FileStorage based storages.

zeopack.py has been fixed to use a read-only connection.

Various small autopack-related race conditions have been fixed in the Berkeley storage implementations. There have been some table changes to the Berkeley storages so any storage you created in 3.1.1b1 may not work. Part of these changes was to add a storage version number to the schema so these types of incompatible changes can be avoided in the future.

Removed the chance of bogus warnings in the FileStorage iterator.

ZEO

The ZEO version number was bumped to 2.0.2 on account of the following minor feature additions.

The performance of full cache verification has improved dramatically. Measurements from Jim were somewhere in 2x-5x. The implementation was fixed to use the very-fast getSerial() method on the storage instead of the comparatively slow load().

The ZEO server has an optional timeout feature that will abort a connection that does not commit within a certain amount of time. The timeout works by closing the socket the client is using, causing both client and server to abort the transaction and continue. This is a drastic step, but can be useful to prevent a hung client or other bug from blocking a server indefinitely.

If a client was disconnected during a transaction, the tpc_abort() call did not properly reset the internal state about the transaction. The bug caused the next transaction to fail in its tpc_finish(). Also, any ClientDisconnected exceptions raised during tpc_abort() are ignored.

ZEO logging has been improved by adding more logging for important events, and changing the logging level for existing messages to a more appropriate level (usually lower).

1.13.87 What's new in ZODB3 3.1.1 beta 1?

Release date: 10-Dev-2002

It was possible for earlier versions of ZODB to deadlock when using multiple storages. If multiple transactions committed concurrently and both transactions involved two or more shared storages, deadlock was possible. This problem has been fixed by introducing a sortKey() method to the transaction and storage APIs that is used to define an ordering on transaction participants. This solution will prevent deadlocks provided that all transaction participants that use locks define a valid sortKey() method. A warning is raised if a participant does not define sortKey(). For backwards compatibility, BaseStorage provides a sortKey() that uses __name__.

Added code to ThreadedAsync/LoopCallback.py to work around a bug in asyncore.py: a handled signal can cause unwanted reads to happen.

A bug in FileStorage related to object uncreation was fixed. If an a transaction that created an object was undone, FileStorage could write a bogus data record header that could lead to strange errors if the object was loaded. An attempt to load an uncreated object now raises KeyError, as expected.

The restore() implementation in FileStorage wrote incorrect backpointers for a few corner cases involving versions and undo. It also failed if the backpointer pointed to a record that was before the pack time. These specific bugs have been fixed and new test cases were added to cover them.

A bug was fixed in conflict resolution that raised a NameError when a class involved in a conflict could not be loaded. The bug did not affect correctness, but prevent ZODB from caching the fact that the class was unloadable. A related bug prevented spurious AttributeErrors when a class could not be loaded. It was also fixed.

The script Tools/zeopack.py was fixed to work with ZEO 2. It was untested and had two silly bugs.

Some C extensions included standard header files before including Python.h, which is not allowed. They now include Python.h first, which eliminates compiler warnings in certain configurations.

The BerkeleyDB based storages have been merged from the trunk, providing a much more robust version of the storages. They are not backwards compatible with the old storages, but the decision was made to update them in this micro release because the old storages did not work for all practical purposes. For details, see Doc/BDBStorage.txt.

1.13.88 What's new in ZODB3 3.1 final?

Release date: 28-Oct-2002

If an error occurs during conflict resolution, the store will silently catch the error, log it, and continue as if the conflict was unresolvable. ZODB used to behave this way, and the change to catch only ConflictError was causing problems in deployed systems. There are a lot of legitimate errors that should be caught, but it's too close to the final release to make the substantial changes needed to correct this.

1.13.89 What's new in ZODB3 3.1 beta 3?

Release date: 21-Oct-2002

A small extension was made to the iterator protocol. The Record objects, which are returned by the per-transaction iterators, contain a new *data_txn* attribute. It is None, unless the data contained in the record is a logical copy of an earlier transaction's data. For example, when transactional undo modifies an object, it creates a logical copy of the earlier transaction's data. Note that this provide a stronger statement about consistency than whether the data in two records is the same; it's possible for two different updates to an object to coincidentally have the same data.

The restore() method was extended to take the data_txn attribute mentioned above as an argument. FileStorage uses the new argument to write a backpointer if possible.

A few bugs were fixed.

The setattr slot of the cPersistence C API was being initialized to NULL. The proper initialization was restored, preventing crashes in some applications with C extensions that used persistence.

The return value of TimeStamp's __cmp__ method was clipped to return only 1, 0, -1.

The restore() method was fixed to write a valid backpointer if the update being restored is in a version.

Several bugs and improvements were made to zdaemon, which can be used to run the ZEO server. The parent now forwards signals to the child as intended. Pidfile handling was improved and the trailing newline was omitted.

1.13.90 What's new in ZODB3 3.1 beta 2?

Release date: 4-Oct-2002

A few bugs have been fixed, some that were found with the help of Neal Norwitz's PyChecker.

The zeoup.py tool has been fixed to allow connecting to a read-only storage, when the –nowrite option is given.

Casey Duncan fixed a few bugs in the recent changes to undoLog().

The fstest.py script no longer checks that each object modified in a transaction has a serial number that matches the transaction id. This invariant is no longer maintained; several new features in the 3.1 release depend on it.

The ZopeUndo package was added. If ZODB3 is being used to run a ZEO server that will be used with Zope, it is usually best if the server and the Zope client don't share any software. The Zope undo framework, however, requires that a Prefix object be passed between client and server. To support this use, ZopeUndo was created to hold the Prefix object.

Many bugs were fixed in ZEO, and a couple of features added. See *ZEO-NEWS.txt* for details.

The ZODB guide included in the Doc directory has been updated. It is still incomplete, but most of the references to old ZODB packages have been removed. There is a new section that briefly explains how to use BTrees.

The zeoup.py tool connects using a read-only connection when –nowrite is specified. This feature is useful for checking on read-only ZEO servers.

1.13.91 What's new in ZODB3 3.1 beta 1?

Release date: 12-Sep-2002

We've changed the name and version number of the project, but it's still the same old ZODB. There have been a lot of changes since the last release.

New ZODB cache

Toby Dickenson implemented a new Connection cache for ZODB. The cache is responsible for pointer swizzling (translating between oids and Python objects) and for keeping recently used objects in memory. The new cache is a big improvement over the old cache. It strictly honors its size limit, where size is specified in number of objects, and it evicts objects in least recently used (LRU) order.

Users should take care when setting the cache size, which has a default value of 400 objects. The old version of the cache often held many more objects than its specified size. An application may not perform as well with a small cache size, because the cache no longer exceeds the limit.

Storages

The index used by FileStorage was reimplemented using a custom BTrees object. The index maps oids to file offsets, and is kept in memory at all times. The new index uses about 1/4 the memory of the old, dictionary-based index. See the module ZODB.fsIndex for details.

A security flaw was corrected in transactionalUndo(). The transaction ids returned by undoLog() and used for transactionalUndo() contained a file offset. An attacker could construct a pickle with a bogus transaction record in its binary data, deduce the position of the pickle in the file from the undo log, then submit an undo with a bogus file position that caused the pickle to get written as a regular data record. The implementation was fixed so that file offsets are not included in the transaction ids.

Several storages now have an explicit read-only mode. For example, passing the keyword argument read_only=1 to FileStorage will make it read-only. If a write operation is performed on a read-only storage, a ReadOnlyError will be raised.

The storage API was extended with new methods that support the Zope Replication Service (ZRS), a proprietary Zope Corp product. We expect these methods to be generally useful. The methods are:

- restore(oid, serialno, data, version, transaction)

  Perform a store without doing consistency checks. A client can use this method to provide a new current revision of an object. The `serialno` argument is the new serialno to use for the object, not the serialno of the previous revision.

- lastTransaction()

  Returns the transaction id of the last committed transaction.

- lastSerial(oid)

  Return the current serialno for `oid` or None.

- iterator(start=None, stop=None)

  The iterator method isn't new, but the optional `start` and `stop` arguments are. These arguments can be used to specify the range of the iterator – an inclusive range [start, stop].

FileStorage is now more cautious about creating a new file when it believes a file does not exist. This change is a workaround for bug in Python versions upto and including 2.1.3. If the interpreter was builtin without large file support but the platform had it, os.path.exists() would return false for large files. The fix is to try to open the file first, and decide whether to create a new file based on errno.

The undoLog() and undoInfo() methods of FileStorage can run concurrently with other methods. The internal storage lock is released periodically to give other threads a chance to run. This should increase responsiveness of ZEO clients when used with ZEO 2.

New serial numbers are assigned consistently for abortVersion() and commitVersion(). When a version is committed, the non-version data gets a new serial number. When a version is aborted, the serial number for non-version data does not change. This means that the abortVersion() transaction record has the unique property that its transaction id is not the serial number of the data records.

Berkeley Storages

Berkeley storage constructors now take an optional *config* argument, which is an instance whose attributes can be used to configure such BerkeleyDB policies as an automatic checkpointing interval, lock table sizing, and the log directory. See bsddb3Storage/BerkeleyBase.py for details.

A getSize() method has been added to all Berkeley storages.

Berkeley storages open their environments with the DB_THREAD flag.

Some performance optimizations have been implemented in Full storage, including the addition of a helper C extension when used with Python 2.2. More performance improvements will be added for the ZODB 3.1 final release.

A new experimental Autopack storage was added which keeps only a certain amount of old revision information. The concepts in this storage will be folded into Full and Autopack will likely go away in ZODB 3.1 final. ZODB 3.1 final will also have much improved Minimal and Full storages, which eliminate Berkeley lock exhaustion problems, reduce memory use, and improve performance.

It is recommended that you use BerkeleyDB 4.0.14 and PyBSDDB 3.4.0 with the Berkeley storages. See bsddb3Storage/README.txt for details.

BTrees

BTrees no longer ignore exceptions raised when two keys are compared.

Tim Peters fixed several endcase bugs in the BTrees code. Most importantly, after a mix of inserts and deletes in a BTree or TreeSet, it was possible (but unlikely) for the internal state of the object to become inconsistent. Symptoms then varied; most often this manifested as a mysterious failure to find a key that you knew was present, or that tree.keys() would yield an object that disgreed with the tree about how many keys there were.

If you suspect such a problem, BTrees and TreeSets now support a ._check() method, which does a thorough job of examining the internal tree pointers for consistency. It raises AssertionError if it finds any problems, else returns None. If ._check() raises an exception, the object is damaged, and rebuilding the object is the best solution. All known ways for a BTree or TreeSet object to become internally inconsistent have been repaired.

Other fixes include:

- Many fixes for range search endcases, including the "range search bug:" If the smallest key S in a bucket in a BTree was deleted, doing a range search on the BTree with S on the high end could claim that the range was empty even when it wasn't.
- Zope Collector #419: repaired off-by-1 errors and IndexErrors when slicing BTree-based data structures. For example, an_IIBTree.items()[0:0] had length 1 (should be empty) if the tree wasn't empty.
- The BTree module functions weightedIntersection() and weightedUnion() now treat negative weights as documented. It's hard to explain what their effects were before this fix, as the sign bits were getting confused with an internal distinction between whether the result should be a set or a mapping.

ZEO

For news about ZEO2, see the file ZEO-NEWS.txt.

This version of ZODB ships with two different versions of ZEO. It includes ZEO 2.0 beta 1, the recommended new version. (ZEO 2 will reach final release before ZODB3.) The ZEO 2.0 protocol is not compatible with ZEO 1.0, so we have also included ZEO 1.0 to support people already using ZEO 1.0.

Other features

When a ConflictError is raised, the exception object now has a sensible structure, thanks to a patch from Greg Ward. The exception now uses the following standard attributes: oid, class_name, message, serials. See the ZODB.POSException.ConflictError doc string for details.

It is now easier to customize the registration of persistent objects with a transaction. The low-level persistence mechanism in cPersistence.c registers with the object's jar instead of with the current transaction. The jar (Connection) then registers with the transaction. This redirection would allow specialized Connections to change the default policy on how the transaction manager is selected without hacking the Transaction module.

Empty transactions can be committed without interacting with the storage. It is possible for registration to occur unintentionally and for a persistent object to compensate by making itself as unchanged. When this happens, it's possible to commit a transaction with no modified objects. The change allows such transactions to finish even on a read-only storage.

Two new tools were added to the Tools directory. The `analyze.py` script, based on a tool by Matt Kromer, prints a summary of space usage in a FileStorage Data.fs. The `checkbtrees.py` script scans a FileStorage Data.fs. When it finds a BTrees object, it loads the object and calls the `_check` method. It prints warning messages for any corrupt BTrees objects found.

Documentation

The user's guide included with this release is still woefully out of date.

Other bugs fixed

If an exception occurs inside an _p_deactivate() method, a traceback is printed on stderr. Previous versions of ZODB silently cleared the exception.

ExtensionClass and ZODB now work correctly with a Python debug build.

All C code has been fixed to use a consistent set of functions from the Python memory API. This allows ZODB to be used in conjunction with pymalloc, the default allocator in Python 2.3.

zdaemon, which can be used to run a ZEO server, more clearly reports the exit status of its child processes.

The ZEO server will reinitialize zLOG when it receives a SIGHUP. This allows log file rotation without restarting the server.

1.13.92 What's new in StandaloneZODB 1.0 final?

Release date: 08-Feb-2002

All copyright notices have been updated to reflect the fact that the ZPL 2.0 covers this release.

Added a cleanroom PersistentList.py implementation, which multiply inherits from UserDict and Persistent.

Some improvements in setup.py and test.py for sites that don't have the Berkeley libraries installed.

A new program, zeoup.py was added which simply verifies that a ZEO server is reachable. Also, a new program zeopack.py was added which connects to a ZEO server and packs it.

1.13.93 What's new in StandaloneZODB 1.0 c1?

Release Date: 25-Jan-2002

This was the first public release of the StandaloneZODB from Zope Corporation. Everything's new! :)

1.14 Reference Documentation

1.14.1 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.

1.14.2 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.

`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.

For proper MVCC operation, the return value is the id of the last transaction for which invalidation notifications are completed.

In particular for client-server implementations, lastTransaction should return a cached value (rather than querying the server). A preliminary call to sync() can be done to get the actual last TID at the wanted time.

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

`__len__``()`
The approximate number of objects in the storage

This is used soley for informational purposes.

`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.

`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.

`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()*).

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 at 0x7f9cdb9e8350 ZODB.interfaces.IS`

The data record, bytes

`data_txn = <zope.interface.interface.Attribute object at 0x7f9cdb9e8390 ZODB.interface`

The previous transaction id, bytes

`oid = <zope.interface.interface.Attribute object at 0x7f9cdb9dfcd0 ZODB.interfaces.ISt`

The object id, bytes

`tid = <zope.interface.interface.Attribute object at 0x7f9cdb9e82d0 ZODB.interfaces.ISt`

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 at 0x7f9cdb9e8450 ZODB.interfaces.ISt`

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.org/project/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.org/project/zc.beforestorage/.

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

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

1.14.3 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.

A transaction manager supports the "context manager" protocol: Its *__enter__* begins a new transaction; its *__exit__* commits the current transaction if no exception has occured; otherwise, it aborts it.

`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..

[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.

The *~ISynchronizer.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.)

`get``()`
Get the current transaction.

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

`commit``()`
Commit the current transaction.

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

`abort``()`
Abort the current transaction.

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

`doom``()`
Doom the current transaction.

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

`isDoomed``()`
Return 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.

`user = <zope.interface.interface.Attribute object at 0x7f9cdba04790 transaction.interf`
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).

`description = <zope.interface.interface.Attribute object at 0x7f9cdba04710 transaction`
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).

`commit`()
Finalize the transaction.

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

`abort`()
Abort the transaction.

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

`doom`()
Doom the transaction.

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

Otherwise the transaction will behave as if it was active.

`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.

`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*.

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

Parameters

- **`name`** (*`text`*) – is the text (unicode) name of the extension property to set
- **`value`** – must be picklable and json serializable

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).

`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.

Parameters

- **`args`** (*`sequence`*) – Additional positional arguments to be passed to the hook. The default is to pass no positional arguments.

- **kws** (*dict*) – Keyword arguments to pass to the hook. The default is to pass no keyword arguments.

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 *ITransactionManager.registerSynch* instead.

getBeforeCommitHooks()

Return iterable producing the registered *addBeforeCommitHook* 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.

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* and *kws* are interpreted as for *addBeforeCommitHook* (with the exception that there is always one positional argument, the commit status). As with *addBeforeCommitHook*, multiple hooks can be registered, savepoint creation doesn’t call any hooks, and calling a hook consumes its registration.

getAfterCommitHooks()

Return iterable producing the registered *addAfterCommitHook* hooks.

As with *getBeforeCommitHooks*, 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.

- The ZODB Book (in progress)

CHAPTER 2

Downloads

ZODB is distributed through the Python Package Index.

You can install the ZODB using pip command:

```
$ pip install ZODB
```

CHAPTER 3

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 repository
- persistent documentation and its repository.
- transaction documentation and its repository
- BTrees documentation and their repository
- ZEO (client-server framework) documentation and its repository
- relstorage documentation and its repository
- zodburi documentation and its repository
- NEO documentation and its repository
- readonlystorage repository
- newt db documentation and its repository

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

Index