Introduction to the Zope Object Database

Jim Fulton, *Digital Creations, jim@digicool.com*

Abstract

The Zope Object Database provides an object-oriented database for Python that provides a high-degree of transparency. Applications can take advantage of object database features with few, if any, changes to application logic. Usage of the database is described and illustrated with an example. Features such as a plug-able storage interface, rich transaction support, undo, and a powerful object cache are described.

1. Introduction

Many applications need to store data for use over multiple application executions, or to use more data than can practically be stored in memory. A number of approaches can be used to manage large amounts of persistent data. Perhaps the most common approach is to use relational database systems. Relational database systems provide a simple model for organizing data into tables, and most can handle large amounts of data effectively. Because of their simple data model, relational databases are easy to understand, at least for small problems. Unfortunately, relational databases can become quite cumbersome when the problem domain does not fit a simple tabular organization.

An advantage of relational database systems is their programming-language neutrality. Data are stored in tables, which are language independent. An application must read data from tables into program variables before use and must write modified data back to tables when necessary. This puts a significant burden on the application developer. A significant amount of application logic is devoted to translation of data to and from the relational model.

An alternative is to retain the tabular structure in the program. For example, rather than populating objects from tables, simply create and use table objects within the application. In this case, high-level tools can be used to load tables from the relational database. With sufficient knowledge of database keys, tools could automate saving data when tables are changed. A disadvantage of this approach is that it forces the application to be written to the relational model, rather than in an object-oriented fashion. The benefits of object orientation, such as encapsulation and association of logic with data are lost.

Object databases provide a tighter integration between an applications object model and data storage. Data are not stored in tables, but in ways that reflect the organization of the information in the problem domain. Application developers are freed from writing logic for moving data to and from storage[1].

The purpose of this paper is to present an object database for Python, the Zope Object Database (ZODB). The goals of the paper are to describe the use and benefits of the ZODB, provide a high-level architectural view, to highlight interesting technical issues, and to describe recent and future developments.

2. Application development

This part of the paper provides an introduction to application development with the ZODB.

2.1 Example: an issue tracking system

This section will present a simple issue tracking system as a means for showing how the Zope object database can be used.

Consider an application that manages a collection of issues. The data for this application might be implemented in an 'Issue' module, as shown in Example 1:

There is an `Issues` class that manages a collection of issues and a text index to support full-text search for issues. An issue may have comments, which may have comments, and so on, recursively. The text for an issue and it's comments is indexed so that issues can be searched for based on issue and comment text.

1. Many object databases, fall short on this last point, as developers must write serialization logic, although this logic is at least encapsulated.

```
from TextIndex import TextIndex

class Issues:

  def __init__(self):
    self._index=TextIndex()
    self._issues=[]

  def addIssue(self, issue):
    issue.setId(len(self._issues))
    self._issues.append(issue)

  def __getitem__(self, i):
    return self._issues[i]

  def search(self, text):
    return map(
       self.__getitem__,
       self._index.search(text))

class Comment:
  _text=''

  def __init__(self, text, parent):
    self._parent=parent
    self.edit(text)
    self._comments=[]

  def text(self):
    return self.text

  def edit(self, text):
    self._unindex(self._text)
    self._text=text
    self._index(self._text)

  def _index(self, text):
    self._parent._index(text)

  def _unindex(self, text):
    self._parent._unindex(text)

  def __getitem__(self, i):
    return self._comments[i]

  def comment(self, text):
    self._comments.append(
        Comment(text, self))

class Issue(Comment):
  _id=None

  def __init__(self, title, text,
               parent):
     Comment.__init__(self, text,
        parent)
     self._title = title

  def setId(self, id):
    self._id=id
    self._index(self._text)

  def title(self): return self._title

  def _index(self, text):
    if self._id is not None:
       self._parent._index(
         text, self._id)

  def _unindex(self, text):
    if self._id is not None:
        self._parent._index(
          text, self._id)
```

Example 1. A simple Issue module.

An application for managing issues will typically be some sort of server or long-running application, like a web application or an interactive graphical application. For brevity, the application will be presented here as a collection of scripts that operate on issues data.

A script for adding issues might be along the lines of that shown in Example 2

```
import Issue, sys

issues=Issue.Issues()

issue=Issue.Issue(
   sys.argv[1], sys.argv[2],
   issues)
issues.addIssue(issue)
```

Example 2. A script for adding an issue

An obvious problem with this script is that it recreates the issue database each time. Obviously, some logic needs to be added to make data persistent between script invocations. The data could be stored in a relational database, but it would be cumbersome to map the hierarchical issue data to a relational model, let alone the text index, which is a "black box" from the point of view of the issue application.

A simple way to add persistence is to save the data in a file in Python pickle format (Example3)

```
import Issue, sys, pickle, os

issues=pickle.Unpickler(
 open('issues.pickle')).load()

issue=Issue.Issue(
   sys.argv[1], sys.argv[2],
   issues)
issues.addIssue(issue)

pickle.Pickler(
  open('issues.pickle','w')
  ).dump(issues)
```

Example 3. A script for adding an issue and saving the issue data in pickler format

The data are stored in the file, `issues.pickle`. When we start the application, the data are read by opening the file, creating an unpickler on it, and calling the `load` method on the unpickler to load the data. After adding the issue, the data must be written to the file by opening the file for writing, creating a pickler on it, and calling the pickler's `dump` method to save the data.

Before calling the add script, the data file must be created with an initialization script (Example 4).

This approach is very simple, but does not scale very well. The entire database is read or written every time an issue is read or saved. A better approach is to use the ZODB. To do this, there are a few changes that need to

```
import Issue, pickle

pickle.Pickler(
  open('issues.pickle','w')
  ).dump(Issue.Issues())
```

Example 4. A script for initializing a pickle file with an empty issues collection.

be made to the application. First, the application classes must be changed to mix-in a special persistence class (Example 5)

Changing the application classes is straightforward. The first change needed is to add the `Persistence.Persistent` base class.

We need to add a line to the `addIssue` and `comment` methods to notify the persistence system that objects have changed:

```
self._p_changed=1
```

This change is necessary because we have modified a list sub-object that doesn't participate in persistence. The normal automatic detection of object changed doesn't work in this case. See "The rules of persistence" later in this paper for further discussion of this change.

The text index is a bit more problematic. We need a modified version of the text index that mixes in the persistent base class as well. This is shown by using a different version of the text index. Modifying the text index is problematic because the text-index is outside the application and it would be preferable if the text index did not have to be changed.

Finally, the application scripts must be modified. The new add script is shown in example 6.

```
import sys, ZODB, ZODB.FileStorage
import Issue

db=ZODB.DB(
       ZODB.FileStorage.FileStorage(
         'issues.fs'))
issues=db.open().root()['issues']

issue=Issue.Issue(
         sys.argv[1], sys.argv[2],
        issues)
issues.addIssue(issue)

get_transaction().commit()
```

Example 6. A script for adding an issue by updating an issues collection on a ZODB.

This add script is similar to the previous one except for a few details. First, note the order of the imports. In particular, the application module, `Issue`, is loaded after `ZODB`. In this case, the import order is important. The `Issue` module imports the `Persistence` module. This module is initially empty. When `ZODB` is imported, it populates the `Persistence` module with

```
import PTextIndex, Persistence

class Issues(Persistence.Persistent):

  def __init__(self):
    self._index=TextIndex()
    self._issues=[]

  def addIssue(self, issue):
    issue.setId(len(self._issues))
    self._issues.append(issue)
    self._p_changed=1

  def __getitem__(self, i):
    return self._issues[i]

  def search(self, text):
    return map(
      self.__getitem__,
      self._index.search(text))

class Comment(Persistence.Persistent):
  _text=''

  def __init__(self, text, parent):
    self._parent=parent
    self.edit(text)

  def text(self): return self.text

  def edit(self, text):
    self._unindex(self._text)
    self._text=text
    self._index(self._text)

  def _index(self, text):
    self._parent._index(text)

  def _unindex(self, text):
    self._parent._unindex(text)

  def __getitem__(self, i):
    return self._comments[i]

  def comment(self, text):
     self._comments.append(
       Comment(text, self))
    self._p_changed=1


class Issue(Comment):
  _id=None

  def __init__(self, title, text, parent):
    Comment.__init__(self,
                     text, parent)
    self._title = title

  def setId(self, id):
    self._id=id
    self._index(self._text)

  def title(self): return self._title

  def _index(self, text):
    if self._id is not None:
      self._parent._index(
        text, self._id)

    def _unindex(self, text):
      if self._id is not None:
        self._parent._index(
           text, self._id)
```

Example 5. A simple issue module modified to use the ZODB

classes, like `Persistent`, that depend on ZODB. This sequence of imports may seem odd, but it allows ZODB to be renamed without affecting much application code. This was very useful when switching from the older version of the object database, `BoboPOS`, to `ZODB`.

Rather than loading all of the data from a pickle file, we open the object database, open a connection to the database, and get the root object, named "issues" from the database. The Zope object database allows a number of different kinds of low-level storage managers to be used. We must first create a storage object, and then create a database object using the storage object. In this example, we used a "file" storage, which is a ZODB storage that stores data in a single file. Other storages are or soon will be available, such as dbm file storages and storages that use relational databases.

It's important to note in this example, that we're only loading a small part of the database into memory. Essentially, only the issue container and issue place-holders are loaded into memory. Issue state and issue comments are not loaded.

Rather than dumping the entire database as a single pickle, we simply commit a transaction. This is an important feature of the ZODB. The application programmer does not have to be aware of the objects that were changed in a computation. The application programmer simply needs to define when work should be saved. This is especially important in object-oriented applications. For an application programmer to control what objects need to be saved would require knowledge of object internals. For example, a user of issues would need to know that a `Issues` objects contain indexes that needed to be saved when an issue was added or modified.

ZODB installs a function, `get_transaction` function in the Python `__builtins__` module. This is done so that transaction-aware tools can use a transaction manager without depending on specific database implementations. To commit the current transaction, call `get_transaction` to get the current transaction, and then call the transaction's `commit` method to commit the transaction, as shown in example 6.

As when storing data in a pickle file, we need a script that initializes the database (example 7).

In a long running application, such as a web application or a graphical application, database open and creation are typically performed during application start-up, so this code is not required in every part of the application that modifies data. Further, transaction boundaries are

```
import ZODB, ZODB.FileStorage
import Issue

db=ZODB.DB(
   ZODB.FileStorage.FileStorage(
      'issues.fs', create=1))

root=db.open().root()
root['issues']=Issue.Issues()

get_transaction().commit()
```

Example 7. A script for initializing a ZODB with an issues collection.

usually defined outside of the ordinary application code. In a web application, a transaction might be committed at the end of a web request, as is done in Zope. In a graphical application, there might be menu options for "saving work" that commits a transaction. Typically, application code doesn't need to define transaction boundaries.

Usually, business logic doesn't contain any database related code, with the exception of mixing in the `Persistent` base class in class statements. There are some cases when the application developer does have to be aware of persistence issues. These cases will be discussed in later sections of the paper.

2.2 Database organization

The ZODB database spreads object storage over multiple records. Each stored persistent object has it's own database record. When an object is modified and saved to the database, only the object's record is affected. Records for unchanged persistent sub-objects are unaffected. Each object has a persistent object id that uniquely identifies the object within the database and is used to lookup object data in the database.

The database has a designated "root" object, which provides access to application root objects by name. An application typically provides a single root object as in the example given earlier in this paper. All other objects are accessed through object traversal from the root, where object traversal might be performed by attribute access, item access, or method call.

There is no application level organization imposed by the ZODB. There is no database-imposed notion of tables or indexes. Applications are free to impose any organization on the object database. One could implement a relational database on top of the ZODB. Indexes are readily implemented on top of ZODB. Zope includes a number of high- and low-level indexing facilities built on the ZODB.

2.3 The rules of persistence

Most applications require few changes to use the ZODB. There are, however, a few rules that must be followed. This section details what these rules are and the reasons behind them.

A major goal of the ZODB is to make persistence as automatic as possible. Infrastructure exists to automate two critical tasks:

1. Notifying the persistence system when an object has changed

   The persistence system keeps track of changes to objects so that only changed objects are saved when a transaction is committed and so that old state can be restored when a transaction is aborted.

2. Notifying the persistence system when an object has been accessed

   When dealing with large databases, the persistence system moves objects into memory when they are needed and out of memory when they are no longer being used. To know when an object is no longer used, it is necessary to track whether an object is being accessed.

To allow the infrastructure to automate these tasks, the following rules must be followed:

1. Persistent object classes must subclass persistent object classes.

   There is a standard persistent base class `Persistence.Persistent`, that is typically subclassed, directly or indirectly. This class provides implementations of the special Python methods `__getattr__` and `__setattr__` that notify the persistence system when an object is accessed or modified. This is the key mechanism by which the tasks described above are automated.

   For standard Python class instances, the special method `__getattr__` is called only when a normal attribute look-up fails. To know when an object can be removed from memory, it is necessary to execute logic on every attribute access. For this reason, the persistent base class is not an ordinary Python class. It is, instead, an ExtensionClass [Fulton96]. Extension classes are not technically Python classes, but are class-like objects that provide features found both in Python classes and built-in types. Any sub-class of an extension class is an extension class, so all persistent object classes are extension classes.

2. All sub-objects of persistent objects must be persistent or immutable.

   This rule is necessary because, without it, the persistence system would not be notified of persistent object state changes.

   Like most rules, this rule can be broken with care, as is done in the issue tracking system. A persistent object can use mutable non-persistent sub-objects if it notifies the persistence system that the sub-object has changed. It can do this in two ways. It can notify the persistence system directly by assigning a true value to the attribute _p_changed, as in:

   ```
   def addIssue(self, issue):
     issue.setId(i=len(self._issues)
     self._issues.append(issue)
     self._p_changed=1
   ```

   or it can notify the persistence system indirectly by re-assigning the sub-object attribute:

   ```
   def addIssue(self, issue):
     issue.setId(len(self._issues))
     self._issues.append(issue)
     self._issues=self._issues
   ```

3. A persistent object must not implement `__getattr__` or `__setattr__`.

   These special methods are already implemented by the persistence system. Overriding them correctly, while possible, is extremely difficult.

4. Persistent objects must be pickle-able.

   The ZODB stores objects in Python pickle format [van Rossum99]. All of the rules for pickling objects apply. See the documentation for the Python `pickle` module for more details.

   Sometimes, a persistent object may temporarily contain unpickleable sub-objects. This is possible as long as the unpickleable objects are not included in the object's pickled state. The object's pickled state is obtained during pickling by calling the object's `__getstate__` method with no arguments. The persistent base class, `Persistence.Persistent`, provides an implementation of `__getstate__` that returns the items in an object's instance dictionary excluding items with keys that start with the prefix "`_v_`" or "`_p_`". The easiest way to prevent data from being pickled is to assign it to an attribute with a name beginning with "`_v_`"[1].

   An object's state may be freed at any time by the ZODB to conserve memory usage. For this reason, an object must be prepared to recompute sub-objects that are not included in the pickled state. A

1. Attributes with names beginning with "`_p_`" are reserved for use by the ZODB.

convenient place to do this is in the `__setstate__` method, which is called when an object's state is loaded from a database. For example, one might have a persistent object that provides an interface to an external file. The persistent object stores the file name in it's persistent state and uses a "volatile" variable to hold the open file:

```
class pfile(Persistence.Persistent):

     def __init__(self, file_name):
         self._file_name=file_name
         self._v_file=open(file_name)

     def __setstate__(self, state):

       Persistence.Persistent. \
         __setstate__(
            self, state)

       self._v_file=open(
         self._file_name)
```

5. Instance attribute names beginning with "_p_" are reserved for use by the ZODB.

In addition to the rules of persistence above, the following advice is worth heeding by authors of any pickleable objects:

- Never implement the obsolete `__getinitargs__` pickling method. This method introduces significant backward compatibility problems.
- Avoid implementing custom pickle state by overriding the pickling methods `__getstate__` and `__setstate__`. Overriding these methods provides greater control and can allow significant optimizations, however, experience has shown that using custom pickle state formats introduces brittleness to an application that is rarely justified by the optimization benefits.

2.4 Object copies and states

With regard to persistence, Python objects have one state, which is existence. They enter existence when they are created, and leave existence when they are destroyed.

Objects that are made persistent with the standard pickle module can be in two states, in memory, and pickled, and can have multiple copies, any of which are in one of the two states. Objects are created only once, even though they may be copied to and from storage many times. The constructor is called only when an object is created initially[1].

ZODB persistent objects can have additional states. Like ordinary pickled objects, persistent objects can have copies that are stored somewhere as pickles. Persistent objects are created only once, but may be copied two and from storage many times. When in memory, ZODB persistent objects may be in one of several states. The object states and transactions are summarized in figure 1. The states are described below.

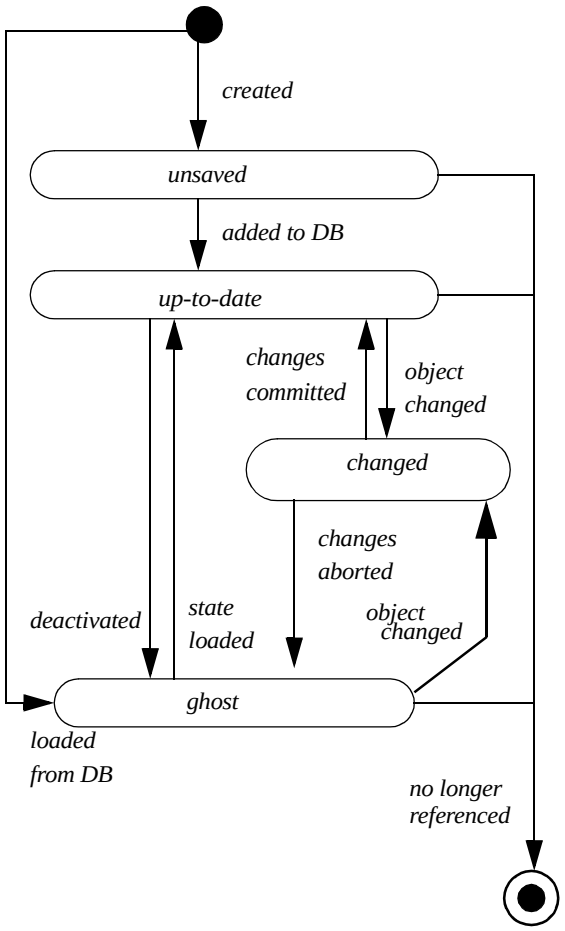

Figure 1. State diagram showing in-memory persistent object states and transitions.

1. unless the obsolete `__getinitargs__` method is used

unsaved

When an object is first created, it is in an unsaved state. An unsaved object can transition to the up-to-date state when it is stored in the database, by referencing it in an object that is stored in the database and then committing a transaction. When an object is first stored in the database, a copy is stored in the database and the object enters the up-to-date state.

An unsaved object can cease to exist like any other Python object.

up-to-date

An object that has been saved in the database and that has it's state loaded in memory is in the up-to-date state.

An object in the up-to-date state transitions to the changed state when it is modified. At the time of the transition, the object registers with the transaction management system, so that the object's state may be saved if a transaction is committed, or rolled back if the transaction is aborted.

If an up-to-date object hasn't been used in a long period of time, the object cache manager may decide to deactivate the object and free the object's state. The object is still in memory, but it has no state. The object transitions to the ghost state.

If an object is in the up-to-date state but is referenced only by the object cache, then it may be removed from memory.

changed

When an object is changed, it enters the changed state. If the current transaction commits, the object's state is copied to the database and the object transitions to the up-to-date state.

If the current transaction aborts, the object is deactivated and transition to the ghost state.

ghost

An object in the ghost state exists in memory, but has no state loaded.

If an attribute of the object is accessed, then the object's state is loaded from the database and the object enters the up-to-date state. It is also possible to set an attribute on an object that is in the ghost state, in which case it transitions directly from the ghost state to the changed state.

If an object in the ghost state is referenced only by the object cache, then it may be removed from memory.

The application programmer is not responsible for implementing the state transitions described above, but it is useful to understand that and how they take place.

2.5 Error recovery

When data are to be saved permanently, an application commits the current transaction. An application can also abort the current transaction by calling the `abort` method on the current transaction returned by the `get_transaction` function.

When a transaction is aborted, all changes made to persistent objects by the transaction are undone. This provides an extremely powerful facility for recovery from errors.

2.6 Object evolution

Lifetimes for persistent objects are typically very long. It is likely that the implementation of an object's behavior or data structures will change over time.

Change is accommodated by the ZODB[1] in a number of ways. Changes in object methods are easily accommodated because classes are, for the most part, not stored in the object database. Changes to class implementation are reflected in instances the next time an application is executed.

Changes in data structures require some care. Adding attributes to instances is straightforward if a default value can be provided in a class definition. More complex data structure changes must be handled in `__setstate__` methods. A `__setstate__` method can check for old state structures and convert them to new structures when an object's state is loaded from the database.

3. Architecture and features

This section presents a high-level architectural view of the ZODB and discusses several important features with their architectural impacts. A detailed UML model of the ZODB is provided by [Fulton99]. The architecture is shown in a layered representation in figure 2.

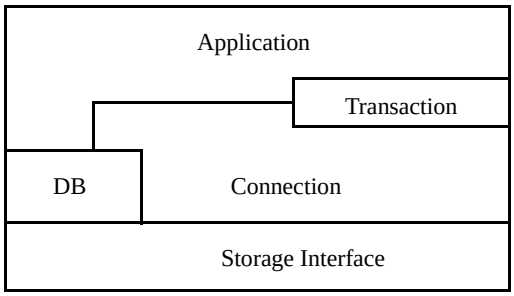

Figure 2. Layered view of the ZODB architecture

Database connections are responsible for moving data to and from storage. Transactions keep track of objects that have changed and coordinate commit and rollback of object changes.

A well-defined storage interface allows different storage managers, with varying levels of service, to be used to manage low-level object storage. The plug-able storage interface affords a great deal of flexibility for managing object data. A basic file storage is provided with ZODB, but other storages are available or planned, including relational-database-based storages, dbm-file-based storages, and Berkely-DB-based storages.

Database, or DB, objects coordinate management of storages and database connections. Applications use DB objects to define the storage to be used, to obtain database connections, and to perform administrative tasks, such as database maintenance.

3.1 Transactions and concurrency

A critical feature of the ZODB is transactions. Transactions can be thought of as small programs that have two important features:

Concurrency control	Transactions execute as though they have exclusive access to data. Multiple transactions can run concurrently and application logic does not have to take concurrent access into account
Atomicity	With respect to persistent object changes, transactions either succeed or fail. In particular, no partial changes to persistent data are retained unless a transaction commits.

The ZODB supports multiple threads in an application that access the same persistent objects. Each thread uses one or more database connections to access the database. Each database connection has it's own copies of persistent objects. Application logic is expressed in object methods. Because each thread has it's own copies of persistent objects, access to an object's methods[2] is limited to a single thread, and application logic can be written without concern for concurrent access.

The ZODB uses an optimistic time-stamp protocol. Changes to individual object copies are made independently, so individual (copies of) objects do not need to be locked. Changes are synchronized when transactions are committed.

Only one transaction is permitted to commit to a storage at a time. If two threads modify the same object in multiple connections, one thread is guaranteed to commit first. When the second thread commits, a `ConflictError` exception will be raised. The application should catch conflict errors and re-execute

1. The characteristics of object evolution in the ZODB are equally applicable to any persistence mechanism based on Python pickles.

2. Or, more precisely, to an object copy's methods.

transactions[1]. When the transaction is re-executed, the states of the affected objects reflect changes made by the committed transactions.

Atomicity greatly simplifies error handling, and is especially important for object-oriented applications because it enables information hiding. Without atomicity, application error recovery logic would need visibility to state of any objects with state that needs to be recovered.

The transaction manager in the ZODB implements a two-phase commit protocol that allows multiple databases to be used in the same application. This could include multiple ZODB databases and multiple relational databases. In Zope, a transaction can effect data in the ZODB and data in one or more relational databases. For example, a transaction might update a Zope object and a row in an Oracle table. If an error occurs, changes made to the ZODB and to the Oracle table will be rolled back.

3.1.1 Sub-transactions

The ZODB provides two levels of nested transactions. Transactions may be subdivided into sub-transactions. Sub-transactions can be committed and aborted without affecting the containing transaction. For example, a transaction may abort a sub-transaction and continue execution. Any changes made in the sub-transaction are undone before execution proceeds. Thus sub-transactions provide fine-grained error recovery.

Sub-transactions are commonly used to reduce memory consumption in transactions that modify many objects. Changed objects cannot be deactivated and remain in memory until a transaction commits. With sub-transactions, objects can be committed and removed from memory without making the changes final, since the enclosing transaction may still be aborted.

3.1.2 Versions

Transactions can also participate in "versions". Versions are similar to long-running transactions. Changes can be committed to a version within the database. Only users of that version see changes made in the version. A version can be committed to the main database, or can be committed to other versions.

Versions provide a mechanism for making changes over a long period of time and to many objects. Changes made in the version are not visible until they are committed and the changes made in a version can be easily discarded. In Zope, this feature allows significant changes to be made to live web sites without effecting users of the sites.

Version support must be provided in the storage used. Version support is an optional storage service.

A locking protocol, rather than a time-stamp protocol is used for coordinating changes to objects in versions. Because versions may extend over long periods of time, it is unreasonable to expect an optimistic time-stamp protocol to be affective.

3.2 Cache management

Each ZODB connection has an object cache that holds references to objects loaded into memory through the connection. At various times, objects in the cache are inspected to see if they are referenced only by the cache, or haven't been accessed for a period of time. Objects that haven't been accessed in a long time are deactivated so that their state is freed. Objects referenced only by the cache are removed from memory. Cache parameters can be set to control how aggressively objects are inspected and to control how recently objects must be accessed before they are deactivated.

3.3 Undo

Transactions may be undone, or rolled back after they are committed if the underlying storage supports "undo" by storing multiple object revisions. The file storage provided with ZODB is an example of a storage that supports undo. When an object is modified, a new object record is appended to the data file and old object revisions are retained.

Old revisions are removed when a database is packed.

3.4 Garbage collection

Circular references among persistent objects do not cause memory leaks. Objects that are no longer accessed are deactivated. When an object is deactivated, any references to sub-objects are released, thus breaking circular references in memory.

A pack operation is provided on the database to remove objects that are no longer referenced from the database root objects. The pack operation also removes unwanted object revisions when storages that keep multiple revisions are used.

1. Zope re-executes transactions up to three times when conflict errors are raised.

4. Status

ZODB 3.0 was released as part of the Zope 2.0 release in September of 1999. ZODB 3.0 added a number of significant features over earlier ZODB releases, most notably:

- Support for concurrent threads of execution[1],
- Well-defined storage interface with integrated transaction support,
- Two-phase commit,
- Integrated versions and sub-transactions,

A number of features are planned for future releases, including:

Auditing Storages that support undo can provide auditing information for object revisions that haven't been removed by packing. This information includes transaction meta data such as who performed an operation, what action they were taking, and other descriptive information. Interfaces need to be provided for storages that keep auditing information independent of undo.

New storages New storages are expected, such as storages that store data in relational databases.

There may also be storages that augment existing storages, such as storages that store data compressed or that reduce storage when updated records are identical with existing records.

A utility is needed to copy data between storages without resorting to object export and import.

Optimization Over time, parts of the database will be re-implemented in C to provide greater performance or to reduce memory usage.

Locking As mentioned earlier, objects are not locked and conflicts are only detected when objects are written. If object data is read and used to update another object, the other object may be updated from stale data. A protocol will be added for verifying objects that are read. Objects will not be locked, but object time stamps will be verified at commit time. If an object time stamp is no longer valid, then a `ConflictError` exception will be raised, so that the transaction may be resubmitted with up-to-date data.

Similarly, content management protocols, such as WebDAV require the ability to lock objects for editing. The locks needed by these protocols span many transactions. It is likely that this need will be satisfied by creating versions associated with locks and create version locks for the affected objects within the versions without actually writing object data.

Cross-database access Currently, although an application can use multiple ZODB databases, objects in one database cannot access objects in another. The ability to support cross-database accesses will make it feasible to build object systems that support multiple storage semantics and more flexible storage management.

Transaction processing monitor support The ZODB uses it's own transaction manager to implement two-phase commit. In the future, it should be possible to use an external transaction processing monitor.

1. Support is included for multiple threads of execution within a process. Protocols are available to implement support for multiple processes.

Application-level conflict resolution protocols	The optimistic time-stamp protocol used by the ZODB is well suited to design environments and other environments where there are complex data structures and in which reads are far more common that writes. Applications with much higher write to read ratios are likely to encounter frequent conflict errors which can seriously affect performance. An approach for coping with conflicts is to provide conflict resolution at the application level. To see a simple example of this, consider a transaction that increments a counter. The transaction reads the counter's old value, increments it, and writes it back. Two transactions that try to increment the counter at the same time will conflict. With some help from the designer of the counter class, we can arrange for increment operations to be non-conflicting.
Query language	Python provides the query language for the ZODB, however, it is desirable to provide industry standard query languages, such as the Object Query Language (OQL). It would be desirable to have an OQL implementation for Python, that could be layered over the ZODB.

5. Summary

The ZODB provides an object-oriented database for Python that provides a high-degree of transparency. Applications can take advantage of object database features with few, if any, changes to application logic. With the exception of “root” objects, it isn’t necessary to query or update objects through database interactions. Objects are obtained and updated through normal object interactions. A plug-able storage interface provides a great deal of flexibility for managing data. Transactions can be undone in a way that maintains transaction integrity. An object cache provides high-performance, efficient memory usage, and protection from memory leaks due to circular references.

6. References

Fulton96, Fulton, James L., 1996, Extension Classes, Python Extension Types Become Classes, http://www.digicool.com/releases/ExtensionClass.

Fulton99, Fulton, James L., 1999, Zope Object Database Version 3 UML model, http://www.zope.org/Documentation/Models/ZODB.

van Rossum99, vanRossum, Guido, 1999, Python Library Reference, Corporation for National Research Initiatives (CNRI), 1895 Preston White Drive, Reston, Va 20191, USA.