ZODB Storage API

Release 1.00

Jeremy Hylton

January 17, 2003

jeremy@zope.com

Abstract

A ZODB storage provides the low-level storage for ZODB transactions. Examples include FileStorage, OracleStorage, and bsddb3Storage. The storage API handles storing and retrieving individual objects in a transaction-specifc way. It also handles operations like pack and undo. This document describes the interface implemented by storages.

Contents

1 Concepts

1.1 Versions

Versions provide support for long-running transactions. They extend transaction semantics, such as atomicity and serializability, to computation that involves many basic transactions, spread over long periods of time, which may be minutes, or years.

Versions were motivated by a common problem in website management, but may be useful in other domains as well. Often, a website must be changed in such a way that changes, which may require many operations over a period of time, must not be visible until completed and approved. Typically this problem is solved through the use of *staging servers*. Essentially, two copies of a website are maintained. Work is performed on a staging server. When work is completed, the entire site is copied from the staging server to the production server. This process is too resource intensive and too monolithic. It is not uncommon for separate unrelated changes to be made to a website and these changes will need to be copied to the production server independently. This requires an unreasonable amount of coordination, or multiple staging servers.

ZODB addresses this problem through long-running transactions, called *versions*. Changes made to a website can be made to a version (of the website). The author sees the version of the site that reflects the changes, but people working outside of the version cannot see the changes. When the changes are completed and approved, they can be saved, making them visible to others, almost instantaneously.

Versions require support from storage managers. Version support is an optional feature of storage managers and support in a particular database will depend on support in the underlying storage manager.

2 Storage Interface

General issues:

The objects are stored as Python pickles. The pickle format is important, because various parts of ZODB depend on it, e.g. pack.

Conflict resolution

Various versions of the interface.

Concurrency and transactions.

The various exceptions that can be raised.

An object that implements the `Storage` interface must support the following methods:

`tpc_begin`(*transaction*[, *tid*[, *status*]])

Begin the two-phase commit for *transaction*.

This method blocks until the storage is in the not committing state, and then places the storage in the committing state. If the storage is in the committing state and the given transaction is the transaction that is already being committed, then the call does not block and returns immediately without any effect.

The optional *tid* argument specifies the timestamp to be used for the transaction ID and the new object serial numbers. If it is not specified, the implementation chooses the timestamp.

The optional *status* argument, which has a default value of `' '`, has something to do with copying transactions.

`store`(*oid*, *serial*, *data*, *version*, *transaction*)

Store *data*, a Python pickle, for the object ID identified by *oid*. A Storage need not and often will not write data immediately. If data are written, then the storage should be prepared to undo the write if a transaction is aborted.

The value of *serial* is opaque; it should be the value returned by the `load()` call that read the object. *version* is a string that identifies the version or the empty string. *transaction*, an instance of `ZODB.Transaction.Transaction`, is the current transaction. The current transaction is the transaction passed to the most recent `tpc_begin()` call.

There are several possible return values, depending in part on whether the storage writes the data immediately. The return value will be one of:

- `None`, indicating the data has not been stored yet
- a string, containing the new serial number for the object
- a sequence of object ID, serial number pairs, containing the new serial numbers for objects updated by earlier `store()` calls that are part of this transaction. If the serial number is not a string, it is an exception object that should be raised by the caller. **Note:** This explanation is confusing; how to tell the sequence of pairs from the exception? Barry, Jeremy, please clarify here.

Several different exceptions can be raised when an error occurs.

- `ConflictError` is raised when *serial* does not match the most recent serial number for object *oid*.

- `VersionLockError` is raised when object *oid* is locked in a version and the *version* argument contains a different version name or is empty.
- `StorageTransactionError` is raised when *transaction* does not match the current transaction.
- `StorageError` or, more often, a subclass of it, is raised when an internal error occurs while the storage is handling the `store()` call.

`restore`(*oid, serial, data, version, transaction*)
A lot like `store()` but without all the consistency checks. This should only be used when we *know* the data is good, hence the method name. While the signature looks like `store()`, there are some differences:

- *serial* is the serial number of this revision, not of the previous revision. It is used instead of `self._serial`, which is ignored.
- Nothing is returned.
- *data* can be `None`, which indicates a George Bailey object (one who's creation has been transactionally undone).

`new_oid`()
XXX

`tpc_vote`(*transaction*)
XXX

`tpc_finish`(*transaction, func*)
Finish the transaction, making any transaction changes permanent. Changes must be made permanent at this point.

If *transaction* is not the current transaction, nothing happens.

func is called with no arguments while the storage lock is held, but possibly before the updated date is made durable. This argument exists to support the `Connection` object's invalidation protocol.

`abortVersion`(*version, transaction*)
Clear any changes made by the given version. *version* is the version to be aborted; it may not be the empty string. *transaction* is the current transaction.

This method is state dependent. It is an error to call this method if the storage is not committing, or if the given transaction is not the transaction given in the most recent `tpc_begin()`.

If undo is not supported, then version data may be simply discarded. If undo is supported, however, then the `abortVersion()` operation must be undoable, which implies that version data must be retained. Use the `supportsUndo()` method to determine if the storage supports the undo operation.

`commitVersion`(*source, destination, transaction*)
Store changes made in the *source* version into the *destination* version. A `VersionCommitError` is raised if the *source* and *destination* are equal or if *source* is an empty string. The *destination* may be an empty string, in which case the data are saved to non-version storage.

This method is state dependent. It is an error to call this method if the storage is not committing, or if the given transaction is not the transaction given in the most recent `tpc_begin()`.

If the storage doesn't support undo, then the old version data may be discarded. If undo is supported, then this operation must be undoable and old transaction data may not be discarded. Use the `supportsUndo()` method to determine if the storage supports the undo operation.

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

`lastSerial`(*oid*)
Returns the serial number for the last committed transaction for the object identified by *oid*. If there is no serial number for *oid* — which can only occur if it represents a new object — returns `None`. **Note:** This is not defined for `ZODB.BaseStorage`.

`lastTransaction``()`
Return transaction ID for last committed transaction. **Note:** This is not defined for `ZODB.BaseStorage`.

`getName``()`
Returns the name of the store. The format and interpretation of this name is storage dependent. It could be a file name, a database name, etc.

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

`getSerial``(`*oid*`)`
Return the serial number of the most recent version of the object identified by *oid*.

`load``(`*oid, version*`)`
Returns the pickle data and serial number for the object identified by *oid* in the version *version*.

`loadSerial``(`*oid, serial*`)`
Load a historical version of the object identified by *oid* having serial number *serial*.

`modifiedInVersion``(`*oid*`)`
Returns the version that the object with identifier *oid* was modified in, or an empty string if the object was not modified in a version.

`isReadOnly``()`
Returns true if the storage is read-only, otherwise returns false.

`supportsTransactionalUndo``()`
Returns true if the storage implementation supports transactional undo, or false if it does not. **Note:** This is not defined for `ZODB.BaseStorage`.

`supportsUndo``()`
Returns true if the storage implementation supports undo, or false if it does not.

`supportsVersions``()`
Returns true if the storage implementation supports versions, or false if it does not.

`transactionalUndo``(`*transaction_id, transaction*`)`
Undo a transaction specified by *transaction_id*. This may need to do conflict resolution. **Note:** This is not defined for `ZODB.BaseStorage`.

`undo``(`*transaction_id*`)`
Undo the transaction corresponding to the transaction ID given by *transaction_id*. If the transaction cannot be undone, then `UndoError` is raised. On success, returns a sequence of object IDs that were affected.

`undoInfo``(`*XXX*`)`
XXX

`undoLog``([`*first*`[`*, last*`[`*, filter*`]]])`
Returns a sequence of `TransactionDescription` objects for undoable transactions. *first* gives the index of the first transaction to be retured, with `0` (the default) being the most recent.

Note: *last* is confusing; can Barry or Jeremy try to explain this?

If *filter* is provided and not `None`, it must be a function which accepts a `TransactionDescription` object as a parameter and returns true if the entry should be reported. If omitted or `None`, all entries are reported.

`versionEmpty``(`*version*`)`
Return true if there are no transactions for the specified version.

`versions``([`*max*`])`
Return a sequence of the versions stored in the storage. If *max* is given, the implementation may choose not to return more than *max* version names.

`history``(`*oid*`[`*, version*`[`*, size*`[`*, filter*`]]])`

Return a sequence of `HistoryEntry` objects. The information provides a log of the changes made to the object. Data are reported in reverse chronological order. If *version* is given, history information is given with respect to the specified version, or only the non-versioned changes if the empty string is given. By default, all changes are reported. The number of history entries reported is constrained by *size*, which defaults to `1`. If *filter* is provided and not `None`, it must be a function which accepts a `HistoryEntry` object as a parameter and returns true if the entry should be reported. If omitted or `None`, all entries are reported.

`pack`(*t, referencesf*)
Remove transactions from the database that are no longer needed to maintain the current state of the database contents. Undo will not be restore objects to states from before the most recent call to `pack()`.

`copyTransactionsFrom`(*other*[, *verbose*])
Copy transactions from another storage, given by *other*. This is typically used when converting a database from one storage implementation to another. This will use `restore()` if available, but will use `store()` if `restore()` is not available. When `store()` is needed, this may fail with `ConflictError` or `VersionLockError`.

`iterator`([*start*[, *stop*]])
Return a iterable object which produces all the transactions from a range. If *start* is given and not `None`, transactions which occurred before the identified transaction are ignored. If *stop* is given and not `None`, transactions which occurred after the identified transaction are ignored; the specific transaction identified by *stop* will be included in the series of transactions produced by the iterator. **Note:** This is not defined for `ZODB.BaseStorage`.

`registerDB`(*db, limit*)
Register a database *db* for distributed storage invalidation messages. The maximum number of objects to invalidate is given by *limit*. If more objects need to be invalidated than this limit, then all objects are invalidated. This argument may be `None`, in which case no limit is set. Non-distributed storages should treat this is a null operation. Storages should work correctly even if this method is not called.

3 ZODB.BaseStorage Implementation

4 Notes for Storage Implementors

5 Distributed Storage Interface

Distributed storages support use with multiple application processes.

Distributed storages have a storage instance per application and some sort of central storage server that manages data on behalf of the individual storage instances.

When a process changes an object, the object must be invaidated in all other processes using the storage. The central storage sends a notification message to the other storage instances, which, in turn, send invalidation messages to their respective databases.