An introduction to the ZODB

Laurence Rowe (laurence@lrowe.co.uk), Plone Conference 2007, Naples.

Relational Databases are great at handling large quantities of homogenous data. If you're building a ledger system and RDB is a great fit. But RDBs 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 these systems do not really support transactional semantics.

ORMs such as SQLAlchemy make working with RDBs 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 is a transactional system to its core. Transactions provide concurrency control and atomicity.

Transactions are executed as if they have exclusive access to the data, so as an application developer you don't have to worry about threading. Of course there is nothing to prevent two simultaneous conflicting requests, So checks are made at transaction commit time to ensure consistency.

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

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

The hot points for ConflictErrors are the catalogue indexes. Some of the indexes do not support conflict resolution and you will see ConflictErrors under write-intensive loads. On solution is to defer catalogue updates using **QueueCatalog** (PloneQueueCatalog in the

collective), 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.

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 an SQLAlchemy wrapper such as collective.lead.

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** (svn.zope.org).

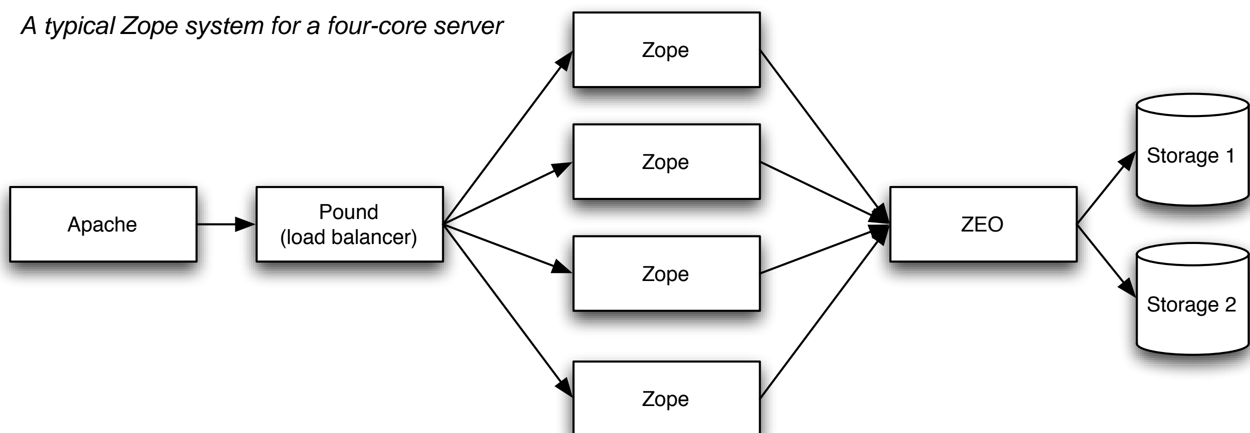
Scalability

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 a problem for it.

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 types

- **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** has a 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.
- **PGStorage** is new and interesting as it does away with the need for ZEO. Though the PostgreSQL network layer is likely faster than ZEO it moves conflict resolution to the application server, which could be bad for worst case performance. Not sure if anyone is using it in production.
- BDBStorage, OracleStorage and APE have all 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. 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 a VACUUM 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. Can be very convenient for getting things done when they are need now...
- Use scalable data structures such as BTrees.
- Keep your content objects simple, add functionality with adapters and views.

More documentation

- The ZODB Wiki: http://wiki.zope.org/ZODB/Documentation