

Versioning datastore design

Introduction

This little design documents talks about the software design of the versioned datastore, hoping to ease the job of whoever tried to hack against the versioning postgis datastore, maybe to port it on top of another jdbc datastore (db2, hsql, and even Oracle, if you don't want to use the workspace managed facilities directly).

The database structure and approach are already described [in a separate document](#), so the same information won't be repeated here.

Version enabling a JDBC datastore requires the following:

- have a set of create/alter table statements ready to create support structures and alter database tables;
- provide mapping between the real versioned data structures and the one shown to the user;
- provide mapping between the filters the user is providing, and those needed to query the versioned table;
- provide an identity concept that does not change because of versions.

All the above must be obtained without adding too much complexity to the datastore that's getting version enabled, in order to allow seamless maintenance of the original data store.

The solution

General design

The postgis datastore could be version enabled in a few ways:

1. by modifying it directly
2. by subclassing it
3. by wrapping it

The first solution would have been too risky, the Postgis datastore is our top notch datastore, so big changes are not welcomed.

The second one has been tried, but it has soon become evident that the JDBC datastore design does not allow for a split between internal and external vision, too many methods inherited from the base classes call other public methods, which is a problem because we do adversite user visible feature types that are not the ones available in the database.

So the implementation used the third solution, which has the following advantages:

- it reuses the postgis datastore with only minor modification (obtained by subclassing), keeping the postgis datastore maintenance constraint free;
- it clearly separates one datastore that knows about the real data structure and deals with actual persistence and data retrieval, from one that does just the mapping;
- it reduces the number of postgis specific knowledge, making it easier to move the code to other data stores as well. In fact, some classes (readers, writers, and feature store) do not use postgis specific knowledge at all, and good part of the datastore class itself does not need changes neither.

The mappings

The following mappings are performed on the fly by the versioned datastore:

- feature type, the internal one contains all the columns (fid mappers are instructed to include the primary key columns in the feature type), whilst the user is provided with the same columns as before (old primary key

- columns are not included in the external feature type at the moment);
- filters, each read/write filter must be augmented with version predicates, and FidFilter have to be split into normal comparison filters against the old primary key columns (the new primary key contains `revision` as well);
 - fid mappers have to map between an internal fid, which is always multicolumn, and an external one, that should not include the "&revision" final part.

Handling identity

Usual FidMappers do not work properly in the versioned environment because we do have two identity concepts:

- the internal one, based on the primary key, that changes for each revision
 - the external one, which is based on the old primary key columns, that must stay unchanged over versions
- Another key difference is that new record do not mean new FIDs, because the new record may just be a new revision of an existing feature (and thus the old primary key columns must have the same values).

A new interface, `VersionedFidMapper`, has been created with extra methods that do map between external and internal ids, plus implementors must handle `createld` properly, that is, avoid creating a new id if the primary key columns have already been populated (something that the `VersionedFeatureWriter` does).