

# Versioning datastore user guide

## Introduction

The versioning postgis datastore allows to version enable one or more feature types so that all edits are versioned the same way as in a source code version control system.

In fact, much of the external interface is inspired by Subversion operations, whilst the internal design is more in line with ArcSDE way of doing things.

The datastore has been designed in such a way that once a feature type has been versioned normal operations keep on working just like usual, that is, normal reads hit the last version, and normal writes do create new versions, so you can keep on using exactly the same code until you need to access some of the extra datastore capabilities, such as logging, diffing or rollbacks.

## Basic interaction with the datastore

VersionedPostgisDataStore works against a Postgis database. You can attach a normal postgis datastore, and it will work just fine in non versioned mode. The only difference is that a few extra tables will be created, and a new feature type, `changesets`, will appear, allowing to access the changelogs as normal features.

## Changesets

The `changesets` features are created at each transaction commit against versioned feature types, and contain the following information:

- `revision`: the revision number of the change;
- `author`: a string that identifies the change author;
- `message`: the commit message;
- `date`: date and time the revision has been created at;
- `bbox`: the lat/lon bbox that has been interested by the changes, represented as a polygon (it's always a rectangle).

## Version enabling features

A feature type can be version enabled by calling `setVersioned(typeName, versioned, author, message)` with `versioned=true`, provided the primary key of the feature type is supported: at the time of writing single string column, multiple columns and serial type are the only supported kind of primary keys.

Author and message should be provided for the changelog. The same command can be used to version disable a feature type.

`isVersioned(featureType)` can be used to check whether a feature type is versioned or not.

Once a feature type is versioned any write against it will be versioned.

## Reading data

Data reading against version enabled feature types can be performed as usual, and it will return the latest version unless a Query is build by populating the `version` parameter.

At the moment the only way to express the version is a revision number, but in future point in time extractions may be supported as well, and branches will have to be included in version specification once the datastore acquires branching capabilities:

```
DefaultQuery q = new DefaultQuery(myType,  
myFilter);  
q.setVersion("5");
```

## Writing

Writing again version enabled features can be performed as usual, and it will create new versions transparently, with two a drawback thought: no author and commit message will be included in the changelog.

To make sure these are populated, create a transaction and load these informations as transaction properties:

```
Transaction t = new DefaultTransaction();  
t.putProperty(VersionedPostgisDataStore.AUTH  
OR, author);  
t.putProperty(VersionedPostgisDataStore.MESS  
AGE, message);
```

The same transaction can be used to modify multiple feature types. Once the transaction is committed, the new revision number generated can be retrieved by querying a transaction property, as well as the version:

```
Long revision =  
t.getProperty(VersionedPostgisDataStore.REVI  
SION); // for example, 5  
String version =  
t.getProperty(VersionedPostgisDataStore.VERS  
ION); // for example, "5"
```

At the time of writing, revision and version are the same, but that will change once branching support is added (version will become something like "branchId:5").

## Advanced interactions: diffs, logs and rollbacks

The following are brand new operations that cannot be achieved with normal datastore API.

Only the most primitive operation, `getModifiedFeatureFIDs`, is provided at the datastore level, whilst the others can be thought as compositions of simpler operations, and have been included in the `VersionedPostgisFeatur`

eStore instead (at the time of writing, feature locking is not supported by the versioned data store).

## Finding features changed between two versions

The following operation returns a set of feature ids for features that were modified, created or deleted between version1 and version2 and that matched the specified filter at least in one revision between version1 and version2:

```
ModifiedFeatureIds getModifiedFeatureFIDs(String typeName, String version1, String version2, Filter filter, Transaction transaction)
```

This operation can be used to build filters and extract the state of the modified features at version1 or version2 and render both on a map to show changes, to perform diffs and to make rollbacks (though the latter are so common operations that are provided out of the box in the VersionedPostgisFeatureStore class).

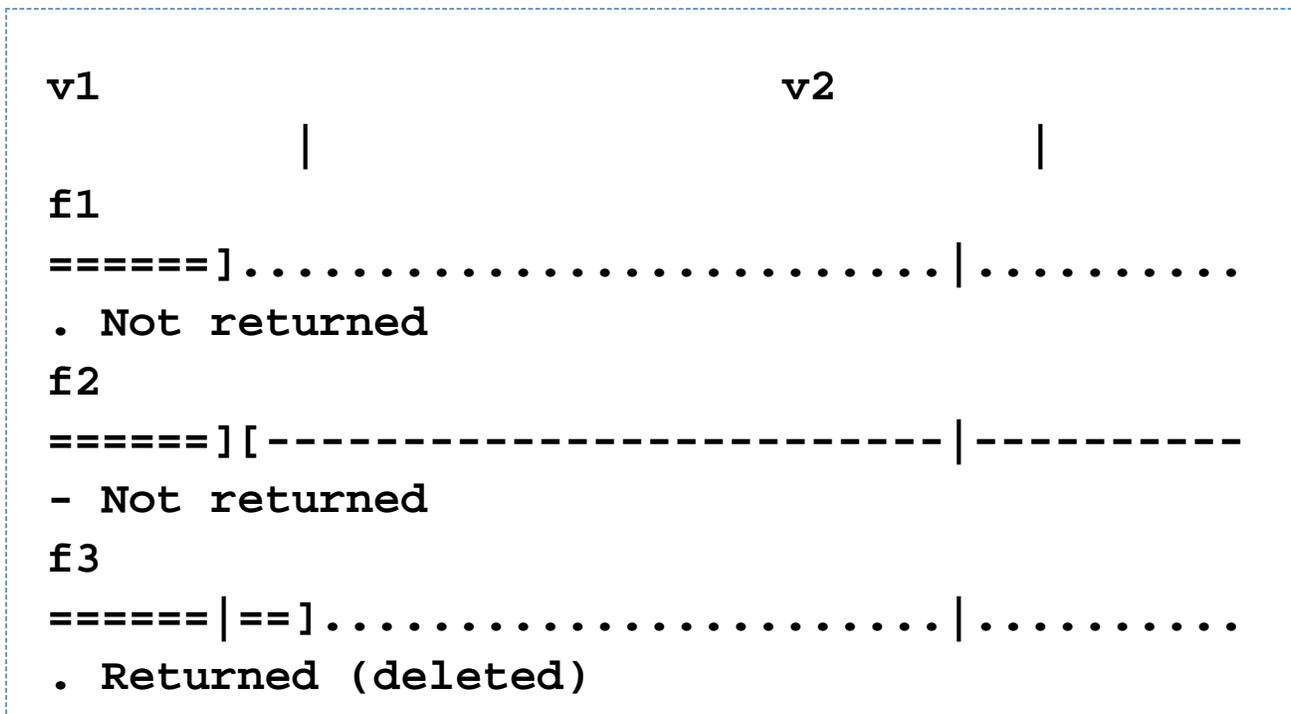
The feature matching semantics is a little complex, so here is a more detailed explanation:

- A feature is said to have been modified between version1 and version2 if a new state of it has been created after version1 and before or at version2 (included), or if it has been deleted between version1 and version2 (included).
- Filter is used to match every state between version1 and version2, so all new states after version1, but also the states existent at version1 provided they existed also at version1 + 1.
- If at least one state matches the filter, the feature id is returned.

The result is composed of three sets of feature ids:

- Features created after version1 are included in the created set
- Features deleted before or at version2 are included in the deleted set
- Features not included in the created/deleted sets are included in the modified set

The following graph illustrates feature matching and set destination. Each line represents a feature lifeline, with different symbols for filter matched states, unmatched states, state creation, expiration, and lack of feature existence.



f4

==== | == ] [ ----- | --- ] .....  
. Returned (modified)

f5

..... | . [ ===== ] ..... | .....  
. Returned (created/deleted)

f5

..... [ ===== ] ..... | .....  
. Returned (deleted)

f5

..... [ ----- ] [ ===== | ===== ] .....  
. Returned (modified)

f6

[ ----- | ----- ] [ ===== ] [ ----- | ----- ]  
- Returned (modified)

Legend:

-: unmatched state

=: matched state

```
.: no state (feature has ben deleted)
[: creation of a state
]: expiration of a state
```

## Rolling back

Rolling back changes can be performed by calling the `VersionedPostgisFeatureStore.rollback` method:

```
{{ public void rollback(String toVersion, Filter filter)}}
```

and specifying the target rollback revision, and which feature have to be included in the rollback operation.

This can be useful to roll back changes performed in a certain area, or to roll back specific feature changes.

Rollback creates a new revision, does not remove rolled back changes from history, just like when rolling back with Subversion (which is done by merging a reverse diff and committing).

## Logging

An equivalent of `svn log` can be performed with the following method on `VersionedPostgisFeatureStore`:

```
public FeatureCollection getLog(String fromVersion, String toVersion, Filter filter)
```

The returned `FeatureCollection` contains all `changeset` feature that are associated to feature changes between the specified versions and matching the specified filter (with the same matching semantics described for `getModifiedFeatureFIDs`).

Returned features are sorted on revision number, descending (just like `svn log`).

## Diffing

Difference computation can be performed calling the following method:

```
FeatureDiffReader getDifferences(String  
fromVersion, String toVersion, Filter  
filter)
```

The returned reader allows to gather a stream of `FeatureDiff` objects describing differences between the features at the two revision that do match the filter (with the same matching semantics described for `getModifiedFeatureFIDs`).

A `FeatureDiff` object provides:

- The feature ID
- A state change, which can be `CREATED`, `DELETED` or `MODIFIED`, depending on how the feature changed between `fromVersion` and `toVersion`;
- A map from changed attributes to their value at `toVersion`.  
If the state is `MODIFIED`, the map contains only the modified attributes, whilst if it's `CREATED` it contains all

attributes.

This is the only command that does not swap versions before executing if fromVersion > toVersion. If a feature is created between fromVersion and toVersion, a swapped request will mark it as deleted, and vice versa. Values for modified features will always be those found at toVersion.

## Some under the hood information

Each time a feature type is version enabled, the underlying table structure gets modified. In particular:

- a new column, `revision`, is added to track the revision in which the record has been created;
- another one, `expired`, is added to track the revision in which the record expired (either a new version of the record has been created, or the feature has been deleted);
- the primary key is altered, if (pk1, ..., pkN) are the old primary key columns, the new primary key is (revision, pk1, ..., pkN)
- a new index, (expired, pk1, ..., pkN) is created as well.

Each time a feature gets modified, the current record is marked as expired and a new one is created, thus, the database contains a full copy of each feature state.

In the future the versioned datastore may learn to shave off some of the old states, preserving only tagger revisions, but for the moment it does not seem necessary to, performance is still good even with lots of revisions inside the db.

For further details, see the [datastore preliminary study and benchmark](#).

Of course this way of doing things prevents other applications to keep on working transparently against the data. If reading is all that's needed, the following view may help:

```
select [all attributes besides revision and  
expired]  
from myVersionedFeatureType  
where revisionExpired = 9223372036854775807
```

Allowing writing could be done as well using "instead of" rules against the above view, but that would replicate lots of the functionality already available in the datastore (which has been designed this way to make the creation of versioned datastores against other databases easy).

If there is sufficient interest and someone sponsoring the work, this could be indeed turned into a PostGIS extension, in this case the datastore could become a tiny wrapper above functionality provided by triggers and stored procedures.