SQLAlchemy-Continuum Documentation

Release 1.3.6

Konsta Vesterinen

Nov 18, 2018

Contents

SQLAlchemy-Continuum is a versioning extension for SQLAlchemy.

CHAPTER 1

Introduction

1.1 Why?

SQLAlchemy already has a versioning extension. This extension however is very limited. It does not support versioning entire transactions.

Hibernate for Java has Envers, which had nice features but lacks a nice API. Ruby on Rails has papertrail, which has very nice API but lacks the efficiency and feature set of Envers.

As a Python/SQLAlchemy enthusiast I wanted to create a database versioning tool for Python with all the features of Envers and with as intuitive API as papertrail. Also I wanted to make it _fast_ keeping things as close to the database as possible.

1.2 Features

- Does not store updates which don't change anything
- Supports alembic migrations
- Can revert objects data as well as all object relations at given transaction even if the object was deleted
- Transactions can be queried afterwards using SQLAlchemy query syntax
- Querying for changed records at given transaction
- Querying for versions of entity that modified given property
- Querying for transactions, at which entities of a given class changed
- History models give access to parent objects relations at any given point in time

1.3 Installation

```
pip install SQLAlchemy-Continuum
```

1.4 Basics

In order to make your models versioned you need two things:

1. Call make_versioned() before your models are defined.
2. Add __versioned__ to all models you wish to add versioning to

```
import sqlalchemy as sa
from sqlalchemy_continuum import make_versioned


make_versioned(user_cls=None)


class Article(Base):
    __versioned__ = {}
    __tablename__ = 'article'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
    content = sa.Column(sa.UnicodeText)


# after you have defined all your models, call configure_mappers:
sa.orm.configure_mappers()
```

After this setup SQLAlchemy-Continuum does the following things:

1. It creates ArticleHistory model that acts as version history for Article model
2. Creates TransactionLog and TransactionChanges models for transactional history tracking
3. Adds couple of listeners so that each Article object insert, update and delete gets recorded

When the models have been configured either by calling configure_mappers() or by accessing some of them the first time, the following things become available:

```
from sqlalchemy_continuum import version_class, parent_class


version_class(Article)  # ArticleHistory class

parent_class(version_class(Article))  # Article class
```

1.5 Versions and transactions

At the end of each transaction SQLAlchemy-Continuum gathers all changes together and creates version objects for each changed versioned entity. Continuum also creates one TransactionLog entity and N number of Transac-

tionChanges entities per transaction (here N is the number of affected classes per transaction). TransactionLog and TransactionChanges entities are created for transaction tracking.

```
article = Article(name=u'Some article')
session.add(article)
session.commit()

article.versions[0].name == u'Some article'

article.name = u'Some updated article'

session.commit()

article.versions[1].name == u'Some updated article'
```

CHAPTER 2

Version objects

2.1 Operation types

When changing entities and committing results into database Continuum saves the used operations (INSERT, UPDATE or DELETE) into version entities. The operation types are stored by default to a small integer field named 'operation_type'. Class called 'Operation' holds convenient constants for these values as shown below:

```
from sqlalchemy_continuum import Operation

article = Article(name=u'Some article')
session.add(article)
session.commit()

article.versions[0].operation_type == Operation.INSERT

article.name = u'Some updated article'
session.commit()
article.versions[1].operation_type == Operation.UPDATE

session.delete(article)
session.commit()
article.versions[2].operation_type == Operation.DELETE
```

2.2 Version traversal

```
first_version = article.versions[0]
first_version.index
# 0


second_version = first_version.next
```

(continues on next page)

(continued from previous page)

```
assert second_version == article.versions[1]

second_version.previous == first_version
# True

second_version.index
# 1
```

2.3 Changeset

Continuum provides easy way for getting the changeset of given version object. Each version contains a changeset property which holds a dict of changed fields in that version.

```
article = Article(name=u'New article', content=u'Some content')
session.add(article)
session.commit(article)

version = article.versions[0]
version.changeset
# {
#    'id': [None, 1],
#    'name': [None, u'New article'],
#    'content': [None, u'Some content']
# }
article.name = u'Updated article'
session.commit()

version = article.versions[1]
version.changeset
# {
#    'name': [u'New article', u'Updated article'],
# }

session.delete(article)
version = article.versions[1]
version.changeset
# {
#    'id': [1, None]
#    'name': [u'Updated article', None],
#    'content': [u'Some content', None]
# }
```

SQLAlchemy-Continuum also provides a utility function called changeset. With this function you can easily check the changeset of given object in current transaction.

```
from sqlalchemy_continuum import changeset


article = Article(name=u'Some article')
changeset(article)
# {'name': [u'Some article', None]}
```

2.4 Version relationships

Each version object reflects all parent object relationships. You can think version object relations as 'relations of parent object in given point in time'.

Lets say you have two models: Article and Category. Each Article has one Category. In the following example we first add article and category objects into database.

Continuum saves new ArticleVersion and CategoryVersion records in the background. After that we update the created article entity to use another category. Continuum creates new version objects accordingly.

Lastly we check the category relations of different article versions.

```
category = Category(name=u'Some category')
article = Article(
    name=u'Some article',
    category=category
)
session.add(article)
session.commit()

article.category = Category(name=u'Some other category')
session.commit()


article.versions[0].category.name  # u'Some category'
article.versions[1].category.name  # u'Some other category'
```

The logic how SQLAlchemy-Continuum builds these relationships is within the RelationshipBuilder class.

2.4.1 Relationships to non-versioned classes

Let's take previous example of Articles and Categories. Now consider that only Article model is versioned:

```
class Article(Base):
    __tablename__ = 'article'
    __versioned__ = {}

    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255), nullable=False)


class Category(Base):
    __tablename__ = 'tag'

    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    article_id = sa.Column(sa.Integer, sa.ForeignKey(Article.id))
    article = sa.orm.relationship(
        Article,
        backref=sa.orm.backref('categories')
    )
```

Here Article versions will still reflect the relationships of Article model but they will simply return Category objects instead of CategoryVersion objects:

```
category = Category(name=u'Some category')
article = Article(
    name=u'Some article',
    category=category
)
session.add(article)
session.commit()

article.category = Category(name=u'Some other category')
session.commit()

version = article.versions[0]
version.category.name                    # u'Some other category'
isinstance(version.category, Category)   # True
```

2.4.2 Dynamic relationships

If the parent class has a dynamic relationship it will be reflected as a property which returns a query in the associated version class.

```
class Article(Base):
    __tablename__ = 'article'
    __versioned__ = {}

    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255), nullable=False)


class Tag(Base):
    __tablename__ = 'tag'
    __versioned__ = {}

    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    article_id = sa.Column(sa.Integer, sa.ForeignKey(Article.id))
    article = sa.orm.relationship(
        Article,
        backref=sa.orm.backref(
            'tags',
            lazy='dynamic'
        )
    )

article = Article()
article.name = u'Some article'
article.content = u'Some content'
session.add(article)
session.commit()

tag_query = article.versions[0].tags
tag_query.all()  # return all tags for given version

tag_query.count()  # return the tag count for given version
```

CHAPTER 3

Reverting changes

One of the major benefits of SQLAlchemy-Continuum is its ability to revert changes.

3.1 Revert update

```
article = Article(name=u'New article', content=u'Some content')
session.add(article)
session.commit(article)

version = article.versions[0]
article.name = u'Updated article'
session.commit()

version.revert()
session.commit()

article.name
# u'New article'
```

3.2 Revert delete

```
article = Article(name=u'New article', content=u'Some content')
session.add(article)
session.commit(article)

version = article.versions[0]
session.delete(article)
session.commit()

version.revert()
```

(continues on next page)

(continued from previous page)

```
session.commit()

# article lives again!
session.query(Article).first()
```

3.3 Revert relationships

Sometimes you may have cases where you want to revert an object as well as some of its relation to certain state. Consider the following model definition:

```
class Article(Base):
    __tablename__ = 'article'
    __versioned__ = {}

    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255))


class Tag(Base):
    __tablename__ = 'tag'
    __versioned__ = {}

    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    article_id = sa.Column(sa.Integer, sa.ForeignKey(Article.id))
    article = sa.orm.relationship(Article, backref='tags')
```

Now lets say some user first adds an article with couple of tags:

```
article = Article(
    name=u'Some article',
    tags=[Tag(u'Good'), Tag(u'Interesting')]
)

session.add(article)
session.commit()
```

Then lets say another user deletes one of the tags:

```
tag = session.query(Tag).filter_by(name=u'Interesting')

session.delete(tag)
session.commit()
```

Now the first user wants to set the article back to its original state. It can be achieved as follows (notice how we use the relations parameter):

```
article = session.query(Article).get(1)
article.versions[0].revert(relations=['tags'])
session.commit()
```

CHAPTER 4

Queries

You can query history models just like any other sqlalchemy declarative model.

```
from sqlalchemy_continuum import version_class


ArticleVersion = version_class(Article)

session.query(ArticleVersion).filter_by(name=u'some name').all()
```

4.1 How many transactions have been executed?

```
from sqlalchemy_continuum import transaction_class


Transaction = transaction_class(Article)


Transaction.query.count()
```

4.2 Querying for entities of a class at a given revision

In the following example we find all articles which were affected by transaction 33.

```
session.query(ArticleVersion).filter_by(transaction_id=33)
```

4.3 Querying for transactions, at which entities of a given class changed

In this example we find all transactions which affected any instance of 'Article' model. This query needs the TransactionChangesPlugin.

```
TransactionChanges = Article.__versioned__['transaction_changes']


entries = (
    session.query(Transaction)
    .innerjoin(Transaction.changes)
    .filter(
        TransactionChanges.entity_name.in_(['Article'])
    )
)
```

4.4 Querying for versions of entity that modified given property

In the following example we want to find all versions of Article class which changed the attribute 'name'. This example assumes you are using PropertyModTrackerPlugin.

```
ArticleVersion = version_class(Article)

session.query(ArticleHistory).filter(ArticleVersion.name_mod).all()
```

CHAPTER 5

Transactions

5.1 Transaction

For each committed transaction SQLAlchemy-Continuum creates a new Transaction record.

Transaction can be queried just like any other sqlalchemy declarative model.

```
from sqlalchemy_continuum import transaction_class
Transaction = transaction_class(Article)

# find all transactions
session.query(Transaction).all()
```

5.2 UnitOfWork

For each database connection SQLAlchemy-Continuum creates an internal UnitOfWork object. Normally these objects are created at before flush phase of session workflow. However you can also force create unit of work before this phase.

```
uow = versioning_manager.unit_of_work(session)
```

Transaction objects are normally created automatically at before flush phase. If you need access to transaction object before the flush phase begins you can do so by calling the create_transaction method of the UnitOfWork class.

```
transaction = uow.create_transaction(session)
```

The version objects are normally created during the after flush phase but you can also force create those at any time by calling make_versions method.

```
uow.make_versions(session)
```

5.3 Workflow internals

Consider the following code snippet where we create a new article.

```
article = Article()
article.name = u'Some article'
article.content = u'Some content'
session.add(article)
session.commit()
```

This would execute the following SQL queries (on PostgreSQL)

1. **INSERT INTO article (name, content) VALUES (?, ?)** params: ('Some article', 'Some content')
2. **INSERT INTO transaction (issued_at) VALUES (?)** params: (datetime.utcnow())
3. **INSERT INTO article_version (id, name, content, transaction_id) VALUES (?, ?, ?, ?)** params: (<article id from query 1>, 'Some article', 'Some content', <transaction id from query 2>)

CHAPTER 6

Native versioning

As of version 1.1 SQLAlchemy-Continuum supports native versioning for PostgreSQL dialect. Native versioning creates SQL triggers for all versioned models. These triggers keep track of changes made to versioned models. Compared to object based versioning, native versioning has

- Much faster than regular object based versioning
- Minimal memory footprint when used alongside *create_tables=False* and *create_models=False* configuration options.
- More cumbersome database migrations, since triggers need to be updated also.

6.1 Usage

For enabling native versioning you need to set *native_versioning* configuration option as *True*.

```
make_versioned(options={'native_versioning': True})
```

6.2 Schema migrations

When making schema migrations (for example adding new columns to version tables) you need to remember to call sync_trigger in order to keep the version trigger up-to-date.

```
from sqlalchemy_continuum.dialects.postgresql import sync_trigger


sync_trigger(conn, 'article_version')
```

CHAPTER 7

Plugins

7.1 Using plugins

```
from sqlalchemy.continuum.plugins import PropertyModTrackerPlugin


versioning_manager.plugins.append(PropertyModTrackerPlugin())


versioning_manager.plugins  # <PluginCollection [...]>

# You can also remove plugin

del versioning_manager.plugins[0]
```

7.2 Activity

The ActivityPlugin is the most powerful plugin for tracking changes of individual entities. If you use ActivityPlugin you probably don't need to use TransactionChanges nor TransactionMeta plugins.

You can initalize the ActivityPlugin by adding it to versioning manager.

```
activity_plugin = ActivityPlugin()

make_versioned(plugins=[activity_plugin])
```

ActivityPlugin uses single database table for tracking activities. This table follows the data structure in activity stream specification, but it comes with a nice twist:

Column	Type	Description
id	BigInteger	The primary key of the activity
verb	Unicode	Verb defines the action of the activity
data	JSON	Additional data for the activity in JSON format
transaction_id	BigInteger	The transaction this activity was associated with
object_id	BigInteger	The primary key of the object. Object can be any entity which has an integer as primary key.
object_type	Unicode	The type of the object (class name as string)
object_tx_id	BigInteger	The last transaction_id associated with the object. This is used for efficiently fetching the object version associated with this activity.
target_id	BigInteger	The primary key of the target. Target can be any entity which has an integer as primary key.
target_type	Unicode	The of the target (class name as string)
target_tx_id	BigInteger	The last transaction_id associated with the target.

Each Activity has relationships to actor, object and target but it also holds information about the associated transaction and about the last associated transactions with the target and object. This allows each activity to also have object_version and target_version relationships for introspecting what those objects and targets were in given point in time. All these relationship properties use generic relationships of the SQLAlchemy-Utils package.

7.2.1 Limitations

Currently all changes to parent models must be flushed or committed before creating activities. This is due to a fact that there is still no dependency processors for generic relationships. So when you create activities and assign objects / targets for those please remember to flush the session before creating an activity:

```
article = Article(name=u'Some article')
session.add(article)
session.flush()  # <- IMPORTANT!
first_activity = Activity(verb=u'create', object=article)
session.add(first_activity)
session.commit()
```

Targets and objects of given activity must have an integer primary key column id.

7.2.2 Create activities

Once your models have been configured you can get the Activity model from the ActivityPlugin class with activity_cls property:

```
Activity = activity_plugin.activity_cls
```

Now let's say we have model called Article and Category. Each Article has one Category. Activities should be created along with the changes you make on these models.

```
article = Article(name=u'Some article')
session.add(article)
session.flush()
first_activity = Activity(verb=u'create', object=article)
session.add(first_activity)
session.commit()
```

Current transaction gets automatically assigned to activity object:

```
first_activity.transaction  # Transaction object
```

7.2.3 Update activities

The object property of the Activity object holds the current object and the object_version holds the object version at the time when the activity was created.

```
article.name = u'Some article updated!'
session.flush()
second_activity = Activity(verb=u'update', object=article)
session.add(second_activity)
session.commit()

second_activity.object.name  # u'Some article updated!'
first_activity.object.name  # u'Some article updated!'

first_activity.object_version.name  # u'Some article'
```

7.2.4 Delete activities

The version properties are especially useful for delete activities. Once the activity is fetched from the database the object is no longer available (since its deleted), hence the only way we could show some information about the object the user deleted is by accessing the object_version property.

```
session.delete(article)
session.flush()
third_activity = Activity(verb=u'delete', object=article)
session.add(third_activity)
session.commit()

third_activity.object_version.name  # u'Some article updated!'
```

7.2.5 Local version histories using targets

The target property of the Activity model offers a way of tracking changes of given related object. In the example below we create a new activity when adding a category for article and then mark the article as the target of this activity.

```
session.add(Category(name=u'Fist category', article=article))
session.flush()
activity = Activity(
    verb=u'create',
    object=category,
    target=article
)
session.add(activity)
session.commit()
```

Now if we wanted to find all the changes that affected given article we could do so by searching through all the activities where either the object or target is the given article.

```
import sqlalchemy as sa


activities = session.query(Activity).filter(
    sa.or_(
        Activity.object == article,
        Activity.target == article
    )
)
```

7.3 Flask

FlaskPlugin offers way of integrating Flask framework with SQLAlchemy-Continuum. Flask-Plugin adds two columns for Transaction model, namely *user_id* and *remote_addr*.

These columns are automatically populated when transaction object is created. The *remote_addr* column is populated with the value of the remote address that made current request. The *user_id* column is populated with the id of the current_user object.

```
from sqlalchemy_continuum.plugins import FlaskPlugin
from sqlalchemy_continuum import make_versioned


make_versioned(plugins=[FlaskPlugin()])
```

7.4 PropertyModTracker

The PropertyModTrackerPlugin offers a way of efficiently tracking individual property modifications. With PropertyModTrackerPlugin you can make efficient queries such as:

Find all versions of model X where user updated the property A or property B.

Find all versions of model X where user didn't update property A.

PropertyModTrackerPlugin adds separate modified tracking column for each versioned column. So for example if you have versioned model called Article with columns *name* and *content*, this plugin would add two additional boolean columns *name_mod* and *content_mod* for the version model. When user commits transactions the plugin automatically updates these boolean columns.

7.5 TransactionChanges

TransactionChanges provides way of keeping track efficiently which declarative models were changed in given transaction. This can be useful when transactions need to be queried afterwards for problems such as:

1. Find all transactions which affected *User* model.
2. Find all transactions which didn't affect models *Entity* and *Event*.

The plugin works in two ways. On class instrumentation phase this plugin creates a special transaction model called *TransactionChanges*. This model is associated with table called *transaction_changes*, which has only only two fields: transaction_id and entity_name. If for example transaction consisted of saving 5 new User entities and 1 Article entity, two new rows would be inserted into transaction_changes table.

transaction_id	entity_name
233678	User
233678	Article

7.6 TransactionMeta

TransactionMetaPlugin offers a way of saving key-value data for transations. You can use the plugin in same way as other plugins:

```
meta_plugin = TransactionMetaPlugin()

versioning_manager.plugins.add(meta_plugin)
```

TransactionMetaPlugin creates a simple model called TransactionMeta. This class has three columns: transaction_id, key and value. TransactionMeta plugin also creates an association proxy between TransactionMeta and Transaction classes for easy dictionary based access of key-value pairs.

You can easily 'tag' transactions with certain key value pairs by giving these keys and values to the meta property of Transaction class.

```
from sqlalchemy_continuum import versioning_manager


article = Article()
session.add(article)

uow = versioning_manager.unit_of_work(session)
tx = uow.create_transaction(session)
tx.meta = {u'some_key': u'some value'}
session.commit()

TransactionMeta = meta_plugin.model_class
Transaction = versioning_manager.transaction_cls

# find all transactions with 'article' tags
query = (
    session.query(Transaction)
    .join(Transaction.meta_relation)
    .filter(
        db.and_(
            TransactionMeta.key == 'some_key',
```

(continues on next page)

(continued from previous page)

```
            TransactionMeta.value == 'some value'
        )
    )
)
```

CHAPTER 8

Configuration

8.1 Global and class level configuration

All Continuum configuration parameters can be set on global level (manager level) and on class level. Setting an option at manager level affects all classes within the scope of the manager's class instrumentation listener (by default all SQLAlchemy declarative models).

In the following example we set 'transaction_column_name' configuration option to False at the manager level.

```
make_versioned(options={'transaction_column_name': 'my_tx_id'})
```

As the name suggests class level configuration only applies to given class. Class level configuration can be passed to __versioned__ class attribute.

```
class User(Base):
    __versioned__ = {
        'transaction_column_name': 'tx_id'
    }
```

8.2 Versioning strategies

Similar to Hibernate Envers SQLAlchemy-Continuum offers two distinct versioning strategies 'validity' and 'subquery'. The default strategy is 'validity'.

8.2.1 Validity

The 'validity' strategy saves two columns in each history table, namely 'transaction_id' and 'end_transaction_id'. The names of these columns can be configured with configuration options *transaction_column_name* and *end_transaction_column_name*.

As with 'subquery' strategy for each inserted, updated and deleted entity Continuum creates new version in the history table. However it also updates the end_transaction_id of the previous version to point at the current version. This creates a little bit of overhead during data manipulation.

With 'validity' strategy version traversal is very fast. When accessing previous version Continuum tries to find the version record where the primary keys match and end_transaction_id is the same as the transaction_id of the given version record. When accessing the next version Continuum tries to find the version record where the primary keys match and transaction_id is the same as the end_transaction_id of the given version record.

Pros:

- Version traversal is much faster since no correlated subqueries are needed

Cons:

- Updates, inserts and deletes are little bit slower

8.2.2 Subquery

The 'subquery' strategy uses one column in each history table, namely 'transaction_id'. The name of this column can be configured with configuration option *transaction_column_name*.

After each inserted, updated and deleted entity Continuum creates new version in the history table and sets the 'transaction_id' column to point at the current transaction.

With 'subquery' strategy the version traversal is slow. When accessing previous and next versions of given version object needs correlated subqueries.

Pros:

- Updates, inserts and deletes little bit faster than in 'validity' strategy

Cons:

- Version traversel much slower

8.3 Column exclusion and inclusion

With *exclude* configuration option you can define which entity attributes you want to get versioned. By default Continuum versions all entity attributes.

```
class User(Base):
    __versioned__ = {
        'exclude': ['picture']
    }

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    picture = sa.Column(sa.LargeBinary)
```

8.4 Basic configuration options

Here is a full list of configuration options:

- **base_classes (default: None)** A tuple defining history class base classes.
- **table_name (default: '%s_version')** The name of the history table.

- **transaction_column_name (default: 'transaction_id')** The name of the transaction column (used by history tables).
- **end_transaction_column_name (default: 'end_transaction_id')** The name of the end transaction column in history table when using the validity versioning strategy.
- **operation_type_column_name (default: 'operation_type')** The name of the operation type column (used by history tables).
- **strategy (default: 'validity')** The versioning strategy to use. Either 'validity' or 'subquery'

Example

```
class Article(Base):
    __versioned__ = {
        'transaction_column_name': 'tx_id'
    }
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
    content = sa.Column(sa.UnicodeText)
```

8.5 Customizing transaction user class

By default Continuum tries to build a relationship between 'User' class and Transaction class. If you have differently named user class you can simply pass its name to make_versioned:

```
make_versioned(user_cls='MyUserClass')
```

If you don't want transactions to contain any user references you can also disable this feature.

```
make_versioned(user_cls=None)
```

8.6 Customizing versioned mappers

By default SQLAlchemy-Continuum versions all mappers. You can override this behaviour by passing the desired mapper class/object to make_versioned function.

```
make_versioned(mapper=my_mapper)
```

8.7 Customizing versioned sessions

By default SQLAlchemy-Continuum versions all sessions. You can override this behaviour by passing the desired session class/object to make_versioned function.

```
make_versioned(session=my_session)
```

CHAPTER 9

Continuum Schema

9.1 Version tables

By default SQLAlchemy-Continuum creates a version table for each versioned entity table. The version tables are suffixed with '_version'. So for example if you have two versioned tables 'article' and 'category', SQLAlchemy-Continuum would create two version tables 'article_version' and 'category_version'.

By default the version tables contain these columns:

- id of the original entity (this can be more then one column in the case of composite primary keys)
- transaction_id - an integer that matches to the id number in the transaction_log table.
- end_transaction_id - an integer that matches the next version record's transaction_id. If this is the current version record then this field is null.
- operation_type - a small integer defining the type of the operation
- versioned fields from the original entity

If you are using *PropertyModTracker* Continuum also creates one boolean field for each versioned field. By default these boolean fields are suffixed with '_mod'.

The primary key of each version table is the combination of parent table's primary key + the transaction_id. This means there can be at most one version table entry for a given entity instance at given transaction.

9.2 Transaction tables

By default Continuum creates one transaction table called *transaction*. Many continuum plugins also create additional tables for efficient transaction storage. If you wish to query efficiently transactions afterwards you should consider using some of these plugins.

The transaction table only contains two fields by default: id and issued_at.

9.3 Using vacuum

sqlalchemy_continuum.**vacuum**(*session*, *model*, *yield_per=1000*)

When making structural changes to version tables (for example dropping columns) there are sometimes situations where some old version records become futile.

Vacuum deletes all futile version rows which had no changes compared to previous version.

```
from sqlalchemy_continuum import vacuum


vacuum(session, User)  # vacuums user version
```

Parameters

- **session** – SQLAlchemy session object
- **model** – SQLAlchemy declarative model class
- **yield_per** – how many rows to process at a time

9.4 Schema tools

sqlalchemy_continuum.schema.**update_end_tx_column**(*table*, *end_tx_column_name='end_transaction_id'*, *tx_column_name='transaction_id'*, *conn=None*)

Calculates end transaction columns and updates the version table with the calculated values. This function can be used for migrating between subquery versioning strategy and validity versioning strategy.

Parameters

- **table** – SQLAlchemy table object
- **end_tx_column_name** – Name of the end transaction column
- **tx_column_name** – Transaction column name
- **conn** – Either SQLAlchemy Connection, Engine, Session or Alembic Operations object. Basically this should be an object that can execute the queries needed to update the end transaction column values.

  If no object is given then this function tries to use alembic.op for executing the queries.

sqlalchemy_continuum.schema.**update_property_mod_flags**(*table*, *tracked_columns*, *mod_suffix='_mod'*, *end_tx_column_name='end_transaction_id'*, *tx_column_name='transaction_id'*, *conn=None*)

Update property modification flags for given table and given columns. This function can be used for migrating an existing schema to use property mod flags (provided by PropertyModTracker plugin).

Parameters

- **table** – SQLAlchemy table object
- **mod_suffix** – Modification tracking columns suffix
- **end_tx_column_name** – Name of the end transaction column

- **tx_column_name** – Transaction column name
- **conn** – Either SQLAlchemy Connection, Engine, Session or Alembic Operations object. Basically this should be an object that can execute the queries needed to update the property modification flags.

  If no object is given then this function tries to use alembic.op for executing the queries.

CHAPTER 10

Alembic migrations

Each time you make changes to database structure you should also change the associated history tables. When you make changes to your models SQLAlchemy-Continuum automatically alters the history model definitions, hence you can use *alembic revision –autogenerate* just like before. You just need to make sure *make_versioned* function gets called before alembic gathers all your models.

Pay close attention when dropping or moving data from parent tables and reflecting these changes to history tables.

CHAPTER 11

Utilities

11.1 changeset

sqlalchemy_continuum.utils.**changeset**(*obj*)

Return a humanized changeset for given SQLAlchemy declarative object. With this function you can easily check the changeset of given object in current transaction.

```
from sqlalchemy_continuum import changeset


article = Article(name=u'Some article')
changeset(article)
# {'name': [u'Some article', None]}
```

Parameters **obj** – SQLAlchemy declarative model object

11.2 count_versions

sqlalchemy_continuum.utils.**count_versions**(*obj*)

Return the number of versions given object has. This function works even when obj has *create_models* and *create_tables* versioned settings disabled.

```
article = Article(name=u'Some article')

count_versions(article)  # 0

session.add(article)
session.commit()

count_versions(article)  # 1
```

Parameters **obj** – SQLAlchemy declarative model object

11.3 get_versioning_manager

sqlalchemy_continuum.utils.**get_versioning_manager**(*obj_or_class*)

Return the associated SQLAlchemy-Continuum VersioningManager for given SQLAlchemy declarative model class or object.

Parameters **obj_or_class** – SQLAlchemy declarative model object or class

11.4 is_modified

sqlalchemy_continuum.utils.**is_modified**(*obj*)

Return whether or not the versioned properties of given object have been modified.

```
article = Article()

is_modified(article)  # False

article.name = 'Something'

is_modified(article)  # True
```

Parameters **obj** – SQLAlchemy declarative model object

See also:

is_modified_or_deleted()

See also:

is_session_modified()

11.5 is_modified_or_deleted

sqlalchemy_continuum.utils.**is_modified_or_deleted**(*obj*)

Return whether or not some of the versioned properties of given SQLAlchemy declarative object have been modified or if the object has been deleted.

Parameters **obj** – SQLAlchemy declarative model object

11.6 is_session_modified

sqlalchemy_continuum.utils.**is_session_modified**(*session*)

Return whether or not any of the versioned objects in given session have been either modified or deleted.

Parameters **session** – SQLAlchemy session object

See also:

is_versioned()

See also:

versioned_objects()

11.7 is_versioned

sqlalchemy_continuum.utils.**is_versioned**(*obj_or_class*)

Return whether or not given object is versioned.

```
is_versioned(Article)  # True

article = Article()

is_versioned(article)  # True
```

Parameters **obj_or_class** – SQLAlchemy declarative model object or SQLAlchemy declarative model class.

See also:

versioned_objects()

11.8 parent_class

sqlalchemy_continuum.utils.**parent_class**(*version_cls*)

Return the parent class for given version model class.

```
parent_class(ArticleVersion)  # Article class
```

Parameters **model** – SQLAlchemy declarative version model class

See also:

version_class()

11.9 transaction_class

sqlalchemy_continuum.utils.**transaction_class**(*cls*)

Return the associated transaction class for given versioned SQLAlchemy declarative class or version class.

```
from sqlalchemy_continuum import transaction_class


transaction_class(Article)  # Transaction class
```

Parameters **cls** – SQLAlchemy versioned declarative class or version model class

11.10 version_class

sqlalchemy_continuum.utils.**version_class**(*model*)

Return the version class for given SQLAlchemy declarative model class.

```
version_class(Article)  # ArticleVersion class
```

Parameters **model** – SQLAlchemy declarative model class

See also:

parent_class()

11.11 versioned_objects

sqlalchemy_continuum.utils.**versioned_objects**(*session*)

Return all versioned objects in given session.

Parameters **session** – SQLAlchemy session object

See also:

is_versioned()

11.12 version_table

sqlalchemy_continuum.utils.**version_table**(*table*)

Return associated version table for given SQLAlchemy Table object.

Parameters **table** – SQLAlchemy Table object

CHAPTER 12

License

Python Module Index

S

Index