# SQLAlchemy-ORM-tree Documentation

***Release 0.2.0***

**RokuSigma Inc. and contributors**

July 05, 2016

# Contents

Contents:

CHAPTER 1

# Installation

At the command line:

```
$ easy_install sqlalchemy-orm-tree
```

Or from pip:

```
$ pip install sqlalchemy-orm-tree
```

CHAPTER 2

# API

## 2.1 Managers

**class** `sqlalchemy_tree.manager.`**`TreeClassManager`**(*node_class*, *options*, *mapper_extension*, *session_extension*)

Node class manager, which handles tree-wide operations such as insertion, deletion, and moving nodes around. No need to create it by hand: it is created by :class:`TreeManager`.

**Parameters**

- **`node_class`** – class which was mapped to tree table.
- **`options`** – instance of :class:`TreeOptions`

**`all_ancestors_of`**(*descendant*, **args*, ***kwargs*)
Return *False* unless every one of the remaining positional arguments is a ancestor of the first.

**`all_child_nodes`**(**args*)
Return *False* unless every one of the positional arguments is a child node.

**`all_children_of`**(*parent*, **args*)
Return *False* unless every one of the remaining positional arguments is a child of the first.

**`all_descendants_of`**(*ancestor*, **args*, ***kwargs*)
Return *False* unless every one of the remaining positional arguments is a descendant of the first.

**`all_leaf_nodes`**(**args*)
Return *False* unless every one of the positional arguments is a leaf node.

**`all_root_nodes`**(**args*)
Return *False* unless every one of the positional arguments is a root node.

**`all_siblings_of`**(*sibling*, **args*, ***kwargs*)
Return *False* unless every one of the remaining positional arguments is a sibling of the first.

**`any_ancestors_of`**(*descendant*, **args*, ***kwargs*)
Return *True* if the first positional argument is a descendant of any of the positional arguments that follow.

**`any_child_nodes`**(**args*)
Return *True* if any of the positional arguments are child nodes.

**`any_children_of`**(*parent*, **args*)
Return *True* if the first positional argument is the parent of any of the positional arguments that follow.

**`any_descendants_of`**(*ancestor*, **args*, ***kwargs*)
Return *True* if the first positional argument is a ancestor of any of the positional arguments that follow.

**any_leaf_nodes**(**args*)
Return *True* if any of the positional arguments are leaf nodes.

**any_root_nodes**(**args*)
Return *True* if any of the positional arguments are root nodes.

**any_siblings_of**(*sibling*, **args*, ***kwargs*)
Return *True* if the first positional argument is a sibling of any of the positional arguments that follow.

**filter_ancestors_of_node**(**args*, ***kwargs*)
Returns a filter condition for the ancestors of passed-in nodes.

**filter_children_of_node**(**args*)
Returns a filter condition for the children of passed-in nodes.

**filter_descendants_of_node**(**args*, ***kwargs*)
Returns a filter condition for the descendants of passed-in nodes.

**filter_leaf_nodes**()
Creates a filter condition containing all leaf nodes.

**filter_leaf_nodes_by_tree_id**(**args*)
Creates a filter condition containing all leaf nodes of the tree(s) specified through the positional arguments (interpreted as tree ids).

**filter_leaf_nodes_of_node**(**args*, ***kwargs*)
Get a filter condition returning the leaf nodes of the descendants of the passed-in nodes.

**filter_next_siblings_of_node**(**args*, ***kwargs*)
Returns a filter condition identifying siblings to the right of passed-in nodes.

**filter_parent_of_node**(**args*)
Get a filter condition for the parents of passed-in nodes.

**filter_previous_siblings_of_node**(**args*, ***kwargs*)
Returns a filter condition identifying siblings to the left of passed-in nodes.

**filter_root_node_by_tree_id**(**args*)
Get a filter condition returning root nodes of the tree specified through the positional arguments (interpreted as tree ids).

**filter_root_node_of_node**(**args*)
Get a filter condition returning the root nodes of the trees which include the passed-in nodes.

**filter_root_nodes**()
Get a filter condition for all root nodes.

**filter_siblings_of_node**(**args*, ***kwargs*)
Returns a filter condition identifying siblings of passed-in nodes.

**query_ancestors_of_node**(**args*, ***kwargs*)
Returns a query containing the ancestors of passed-in nodes.

**query_children_of_node**(**args*, ***kwargs*)
Returns a query containing the children of passed-in nodes.

**query_descendants_of_node**(**args*, ***kwargs*)
Returns a query containing the descendants of passed-in nodes.

**query_leaf_nodes**(*session=None*, **args*, ***kwargs*)
Returns a query containing all leaf nodes.

**query_leaf_nodes_by_tree_id**(**args*, ***kwargs*)
Returns a query containing all leaf nodes of the tree(s) specified through the positional arguments (interpreted as tree ids) using `filter_leaf_nodes_by_tree_id` and the session associated with this node. The session must be passed explicitly if called from a class manager.

**query_leaf_nodes_of_node**(**args*, ***kwargs*)
Returns the leaf nodes of the descendants of the passed-in nodes, using `filter_leaf_nodes_by_tree_id`. The session used to perform the query is either a) the session explicitly passed in, b) the session associated with the first bound positional parameter, or c) the session associated with the instance manager's node.

**query_next_siblings_of_node**(**args*, ***kwargs*)
Returns a query containing siblings to the right of passed-in nodes.

**query_parent_of_node**(**args*, ***kwargs*)
Returns a query containing the parents of passed-in nodes.

**query_previous_siblings_of_node**(**args*, ***kwargs*)
Returns a query containing siblings to the left of passed-in nodes.

**query_root_node_by_tree_id**(**args*, ***kwargs*)
Returns the root nodes of the trees specified through the positional arguments (interpreted as tree ids) using `filter_root_node_by_tree_id` and the session associated with this node. The session must be passed explicitly if called from a class manager.

**query_root_node_of_node**(**args*, ***kwargs*)
Returns the root nodes of the trees which contain the passed in nodes, using `filter_root_node_by_tree_id`. The session used to perform the query is either a) the session explicitly passed in, b) the session associated with the first bound positional parameter, or c) the session associated with the instance manager's node.

**query_root_nodes**(*session=None*, **args*, ***kwargs*)
Convenience method that gets a query for all root nodes using `filter_root_nodes` and the session associated with this node. The session must be passed explicitly if called from a class manager.

**query_siblings_of_node**(**args*, ***kwargs*)
Returns a query containing the siblings of passed-in nodes.

**rebuild**(**args*, ***kwargs*)
Rebuild tree parameters on the basis of adjacency relations for all nodes under the subtrees rooted by the nodes passed as positional arguments. Specifying no positional arguments performs a complete rebuild of all trees.

**Parameters** **order_by** – an "order by clause" for sorting children nodes of each subtree.

TODO: Support order_by. What about the rest of sqlalchemy_tree. Is any order_by used when inserting a new node?

**class** `sqlalchemy_tree.manager.`**TreeInstanceManager**(*class_manager*, *obj*, **args*, ***kwargs*)
A node manager, unique for each node instance. Created on first access to `TreeManager` descriptor from instance. Implements API to query nodes related somehow to a particular node: descendants, ancestors, etc.

**Parameters**

- **class_manager** – the `TreeClassManager` associated with the node class, which is used to perform tree-altering behaviors.
- **obj** – particular node instance.

**filter_ancestors**(*include_self=False*)
The same as `filter_descendants()` but filters ancestor nodes.

**filter_children**()
: The same as `filter_descendants()` but filters direct children only and does not accept an `include_self` parameter.

**filter_descendants**(*include_self=False*)
: Get a filter condition for node's descendants.

  Requires that node has *tree_id*, *left*, *right* and *depth* values available (that means it has "persistent version" even if the node itself is in "detached" state or it is in "pending" state in *autoflush*-enabled session).

  Usage example:

```
session.query(Node).filter(root.mp.filter_descendants()) \
                    .order_by(Node.mp)
```

  This example is silly and meant only to illustrate the syntax for using *filter_descendants*, don't use it for such purpose as there is a better way for such simple queries: `query_descendants()`.

  **Parameters** **include_self** – *bool*, if set to *True*, include this node in the filter as well.

  **Returns** a filter clause applicable as argument for *sqlalchemy.orm.Query.filter()* and others.

**filter_leaf_nodes**(*include_self=False*)
: Creates a filter containing leaf nodes of this node instance.

  Requires that node has *tree_id*, *left*, *right* and *depth* values available (that means it has "persistent version" even if the node itself is in "detached" state or it is in "pending" state in *autoflush*-enabled session).

  **Parameters** **include_self** – *bool*, if set to *True*, the filter will also include this node (if it is a leaf node).

**filter_next_siblings**(*include_self=False*)
: Get a filter condition for the siblings of a node which occur subsequent to it in tree ordering.

**filter_parent**()
: Get a filter condition for a node's parent.

**filter_previous_siblings**(*include_self=False*)
: Get a filter condition for the siblings of a node which occur prior to it in tree ordering.

**filter_root_node**()
: Return a filter condition identifying the root node of the tree which includes this node.

**filter_siblings**(*include_self=False*)
: Get a filter condition for a node's siblings.

**get_descendant_count**()
: Returns the number of descendants this node has.

**is_ancestor_of**(*descendant*, *include_self=False*)
: Returns *True* if the passed-in node is a descendant of this node.

**is_child_node**
: Returns *True* if the node has a parent.

**is_child_of**(*parent*)
: Returns *True* if the passed-in node is parent to this node.

**is_descendant_of**(*ancestor*, *include_self=False*)
: Returns *True* if the passed-in node is an ancestor of this node.

**is_leaf_node**
: Returns *True* if the node has no children.

**is_root_node**
Returns *True* if the node has no parent.

**is_sibling_of**(*sibling*, *include_self=True*)
Returns *True* if the passed-in node is a sibling to this node.

**next_sibling**
Returns the next sibling with respect to tree ordering, or *None*.

**previous_sibling**
Returns the previous sibling with respect to tree ordering, or *None*.

**query_ancestors**(*session=None*, *include_self=False*)
The same as `query_descendants()` but queries node's ancestors.

**query_children**(*session=None*)
The same as `query_descendants()` but queries direct children only and does not accept an `include_self` parameter.

**query_descendants**(*session=None*, *include_self=False*)
Get a query for node's descendants.

Requires that node is in "persistent" state or in "pending" state in *autoflush*-enabled session.

**Parameters**

- **session** – session object for query. If not provided, node's session is used. If node is in "detached" state and `session` is not provided, query will be detached too (will require setting *session* attribute to execute).
- **include_self** – *bool*, if set to *True* self node will be selected by query.

**Returns** a *sqlalchemy.orm.Query* object which contains only node's descendants.

**query_leaf_nodes**(*session=None*, *include_self=False*)
Returns a query containing leaf nodes of this node instance.

Requires that node has *tree_id*, *left*, *right* and *depth* values available (that means it has "persistent version" even if the node itself is in "detached" state or it is in "pending" state in *autoflush*-enabled session).

**Parameters**

- **session** – session object for query. If not provided, node's session is used. If node is in "detached" state and `session` is not provided, query will be detached too (will require setting *session* attribute to execute).
- **include_self** – *bool*, if set to *True*, the filter will also include this node (if it is a leaf node).

**Returns** a *sqlalchemy.orm.Query* object which contains only node's descendants which are themselves leaf nodes.

**query_next_siblings**(*session=None*, *include_self=False*)
Get a query containing the siblings of a node which occur subsequent to it in tree ordering.

**query_previous_siblings**(*session=None*, *include_self=False*)
Get a query containing the siblings of a node which occur prior to it in tree ordering.

**query_root_node**()
Return a query containing the root node of the tree which includes this node.

**query_siblings**(*session=None*, *include_self=False*)
Get a query containing a nodes siblings.

**root_node**
Return the root node of the tree which includes this node.

**class** sqlalchemy_tree.manager.**TreeManager**(**args*, ***kwargs*)
Extension class to create required fields and access class-level and instance-level API based on context.

Basic usage is simple:

```
class Node(object):
  tree = sqlalchemy_tree.TreeManager(node_table)

# After Node is mapped:
Node.tree.register()
```

Now there is an ability to get an instance manager or class manager via the property *'mp'* depending on the way in which it is accessed. *Node.mp* will return the mapper extension until the class is mapped (useful for setting up parameters to pass to the mapper function itself), the class manager `TreeClassManager` after mapping, and *instance_node.mp* will return instance_node's `TreeInstanceManager`. See those classes for more details about their public API's.

**Parameters**

- **table** – instance of *sqlalchemy.Table*. A table that will be mapped to the node class and will hold tree nodes in its rows. The adjacency-list link/self- referential foreign-key will be automatically determined, and the additional four columns "tree_id", "left", "right" and "depth" will automatically be added if necessary. `table` is the only one strictly required argument.
- **parent_id_field=None** – a self-referencing foreign key field containing the parent node's primary key. If this parameter is omitted, it will be guessed joining a *table* with itself and using the right part of join's *ON* clause as parent id field.
- **tree_id_field='tree_id'** – the name of the tree id field, or the field object itself. The field will be created if the actual parameter value is a string and there is no such column in the table `table`. If the value provided is or names an existing SQLAlchemy column object, that object must pass some sanity checks: it must be in `table`, it should have *nullable=False*, and be of type `TreeIdField`.
- **left_field='tree_left'** – the same as for `tree_id_field`, except that the type of this column should be `TreeLeftField`.
- **right_field='tree_right'** – the same as for `tree_id_field`, except that the type of this column should be `TreeRightField`.
- **depth_field='tree_depth'** – the same as for `tree_id_field`, except that the type of this column should be `TreeDepthField`.
- **instance_manager_attr='_tree_instance_manager'** – name for node instance's attribute to cache node's instance manager.

**Warning:** Do not change the values of *TreeManager* constructor's arguments after saving a first tree node. Doing so will corrupt the tree.

## 2.2 ORM Extensions

**class** sqlalchemy_tree.orm.**TreeMapperExtension**(*options*)
An extension to a node class' mapper, handling insertion, deletion, and updates of tree nodes. This class is

instantiated by the manager object, and the average developer need not bother himself with it.

**Parameters** **options** – instance of TreeOptions

**after_delete** (*mapper*, *connection*, *node*)
Just after an existent node is updated.

**after_insert** (*mapper*, *connection*, *node*)
Just after a previously non-existent node is inserted into the tree.

**after_update** (*mapper*, *connection*, *node*)
Just after an existent node is updated.

**before_delete** (*mapper*, *connection*, *node*)
Just prior to an existent node being deleted.

**before_insert** (*mapper*, *connection*, *node*)
Just prior to a previously non-existent node being inserted into the tree.

Sets up the tree state (tree_id, left, right and depth) for node (which has not yet been inserted into in the database) so it will be positioned relative to a given target node in the manner specified by position (the insertion parameters), with any necessary space already having been made for it.

target and position are stored on a hidden attribute of node, having been set when TreeManager.insert was called by the user, or otherwise auto-generated by the session's before_flush handler.

A target of None indicates that node should become the last root node, which is a constant-time insertion operation. (Positioning root nodes with respect to other root nodes can be accomplished by using the POSITION_LEFT or POSITION_RIGHT constants and specifying the neighboring root node as target.)

Accepted values for position are POSITION_FIRST_CHILD, POSITION_LAST_CHILD, POSITION_LEFT or POSITION_RIGHT. POSITION_LAST_CHILD is likely to cause the least number of row updates, so therefore it is the default behavior if position is not specified.

**before_update** (*mapper*, *connection*, *node*)
Called just prior to an existent node being updated.

Possibly moves node relative to a given target node as specified by position (when appropriate), by examining both nodes and calling the appropriate method to perform the move.

target and position are stored on a hidden attribute of node, having been set when TreeManager.insert was called by the user, or otherwise auto-generated by the session's before_flush handler upon detection of an adjacency-list change.

A target of None indicates that node should be made into the last root node. (Positioning root nodes with respect to other root nodes can be accomplished by using the POSITION_LEFT or POSITION_RIGHT constants and specifying the neighboring root node as target.)

Valid values for position are POSITION_LEFT, POSITION_RIGHT, POSITION_FIRST_CHILD or POSITION_LAST_CHILD.

node will be modified to reflect its new tree state in the database. Depending on the type of the move, a good many other nodes might be modified as well.

This method explicitly checks for node being made a sibling of a root node, as this is a special case due to our use of tree ids to order root nodes.

**class** sqlalchemy_tree.orm.**TreeSessionExtension** (*options*, *node_class*)
An session extension handling insertion, deletion, and updates of tree nodes. This class is instantiated by the manager object, and the average developer need not bother himself with it.

**Parameters**

- **`options`** – instance of `TreeOptions`
- **`node_class`** – the mapped object class for tree nodes

**`before_flush`**(*session*, *flush_context*, *instances*)
Just prior to a flush event, while we still have time to modify the flush plan.

## 2.3 Types

**class** `sqlalchemy_tree.types.`**`TreeDepthType`**(**args*, ***kwargs*)
Integer field subtype representing an node's depth level.

**class** `sqlalchemy_tree.types.`**`TreeEndpointType`**(**args*, ***kwargs*)
Abstract base class of an integer implementing either a "left" or "right" field of a node.

**class** `sqlalchemy_tree.types.`**`TreeIdType`**(**args*, ***kwargs*)
Integer field subtype representing an node's tree identifier.

**class** `sqlalchemy_tree.types.`**`TreeIntegerType`**(**args*, ***kwargs*)
Abstract base class implementing an integer type.

**`impl`**
alias of `Integer`

**class** `sqlalchemy_tree.types.`**`TreeLeftType`**(**args*, ***kwargs*)
Integer field subtype representing an node's "left" field.

**class** `sqlalchemy_tree.types.`**`TreeRightType`**(**args*, ***kwargs*)
Integer field subtype representing an node's "right" level.

## 2.4 Options

**class** `sqlalchemy_tree.options.`**`TreeOptions`**(*table*, *instance_manager_attr*, *parent_id_field=None*, *tree_id_field=None*, *left_field=None*, *right_field=None*, *depth_field=None*, *_attach_columns=True*)
A container for options for one tree.

**Parameters** see `TreeManager`.

**`order_by_clause`**()
Get an object applicable for usage as an argument for *Query.order_by()*. Used to sort subtree query by *tree_id* then *left*.

## 2.5 Exceptions

**class** `sqlalchemy_tree.exceptions.`**`InvalidMoveError`**
An invalid node move was attempted. For example, attempting to make a node a child of itself.

CHAPTER 3

# Authors

The primary author of SQLAlchemy-ORM-tree is Mark Friedenbach <mark@monetize.io>.

Others who have contributed to the application:

- Jonathan Buchanan et al. (authors of Django MPTT)
- Anton Gritsay <anton@angri.ru> (author of SQLAMP)
- Tony Narlock <tony@git-pull.com>
- Michael Elsdörfer <michael@elsdoerfer.com>

# Roadmap

## 4.1 dev

- Python 2+3 support (Issue #2)
- Farey fractions isntead of integer intervals (Issue #4)
- SQLAlchemy 0.9 Declarative syntax support (Issue #3)

For more, see the issues on github:

CHAPTER 5

# History

## 5.1 0.2.0-dev

Released: Ongoing

### 5.1.1 tests

- **[tests]**
  - Update testsuite to werkzeug/flask format.
  - Split tests into multiple files
  - Coveralls.io and Travis support

  ¶ References: #11, #5, pull request 8

### 5.1.2 docs

- **[docs]**
  - `__future__` imports for `.py` files.
  - Replaces some instances of `filter` with list comprehensions.
  - Created a `py2map` inside of `_compat` to import into code, which preserves python 2.x's `map` behavior in python 3.
  - Update README.rst for python 3 support
  - Update `setup.py` classifier data
  - Update Travis for python 3.3 support (they don't have 3.4 support yet, tox passes 3.4 for me though)
  - Import `reduce` from `functools`.
  - Add `tox.ini` file.

  ¶ References: #2, pull request 20

- **[docs]** SQLAlchemy-ORM-tree now has a ReadTheDocs page at http://sqlalchemy-orm-tree.readthedocs.org/.

  Changelog

  TODO ¶ References: #7

### 5.1.3 internals

- **[internals]**
  - PEP8
  - Package modernization

  ¶ References: #5, #6, pull request 8, pull request 10

An implementation for SQLAlchemy-based applications of the nested-sets / modified-pre-order-tree-traversal technique for storing hierarchical data in a relational database.

| | |
|---|---|
| Python support | Python 2.6+, 3.3+ |
| SQLAlchemy | SQLAlchemy >=0.7.5, >=0.8, >=0.9 |
| Source | https://github.com/monetizeio/sqlalchemy-orm-tree |
| Issues | https://github.com/monetizeio/sqlalchemy-orm-tree/issues |
| Docs | https://sqlalchemy-orm-tree.readthedocs.org/ |
| API | https://sqlalchemy-orm-tree.readthedocs.org/api.html |
| Travis | http://travis-ci.org/monetizeio/sqlalchemy-orm-tree |
| Test coverage | https://coveralls.io/r/monetizeio/sqlalchemy-orm-tree |
| pypi | https://pypi.python.org/pypi/sqlalchemy-orm-tree |
| ohloh | http://www.ohloh.net/p/sqlalchemy-orm-tree |
| License | BSD. |
| git repo | `$ git clone https://github.com/monetizeio/sqlalche` |
| install | `$ pip install sqlalchemy-orm-tree` |
| install dev | `$ git clone https://github.com/monetizeio/sqlalche`<br>`$ cd ./sqlalchemy-orm-tree`<br>`$ virtualenv .env`<br>`$ source .env/bin/activate`<br>`$ pip install -e .` |
| tests | `$ python setup.py test` |

# Simple Example

```
import sqlalchemy_tree
Model = declarative_base(metaclass=sqlalchemy_tree.DeclarativeMeta)

class Page(Model):

    # This activates sqlalchemy-orm-tree.
    __tree_manager__ = 'tree'
```

Page.tree.register()

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search