sqlamp Documentation

Release 0.6

Anton Gritsay

September 29, 2014

author Anton Gritsay, http://angri.ru

version 0.6

license 2-clause BSD (see LICENSE)

download http://sqlamp.angri.ru/sqlamp-0.6.tar.gz

hack on hg clone https://bitbucket.org/angri/sqlamp

`sqlamp` is an implementation of an efficient algorithm for working with hierarchical data structures — `Materialized Path`. `sqlamp` uses (and depends of) SQLAlchemy.

`Materialized Path` is a way to store (and fetch) a trees in a relational databases. It is the compromise between `Nested Sets` and `Adjacency Relations` in respect to simplicity and efficiency. Method was promoted by Vadim Tropashko in his book SQL Design Patterns. Vadim's description of the method can be read in his article Trees in SQL: Nested Sets and Materialized Path (by Vadim Tropashko).

Implemented features:

- Setting up with `declarative.ext` or without it.
- Saving node roots — if no parent set for node. The tree will have a new `tree_id`.
- Saving child nodes — if node has some parent. The whole dirty job of setting values in `tree_id`, `path` and `depth` fields is done by `sqlamp`.
- Fetching node's descendants, ancestors and children using the most efficient way available (see `MPInstanceManager`).
- Autochecking exhaustion of tree size limits — maximum number of children and maximum nesting level (see `MPManager` to learn more about limits fine-tuning) is done during session flush.
- Rebuilding all trees (see `MPClassManager.rebuild_all_trees()`) on the basis of Adjacency Relations.
- Collapsing flat tree returned from query to recursive structure (see `tree_recursive_iterator()`).
- Node classes may use polymorphic inheritance.
- Nodes and whole trees/subtrees can be moved around or removed entirely. See moving nodes.

Known-to-work supported DBMS include sqlite (tested with 3.6.14), MySQL (tested using both MyISAM and InnoDB with server version 5.1.34) and PostgreSQL (tested with 8.3.7), but sqlamp should work with any other DBMS supported by SQLAlchemy.

Supported versions of SQLAlchemy include current minor versions of branches 0.5 and 0.6 as well as 0.7 since 0.7.2.

CHAPTER 1

Quickstart

Note: Code examples here are all runable by copy-paste to interactive interpreter.

```
import sqlalchemy, sqlalchemy.orm
engine = sqlalchemy.create_engine('sqlite:///:memory:', echo=False)
metadata = sqlalchemy.MetaData(engine)

node_table = sqlalchemy.Table('node', metadata,
    sqlalchemy.Column('id', sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column('parent_id', sqlalchemy.ForeignKey('node.id')),
    sqlalchemy.Column('name', sqlalchemy.String)
)
```

There is nothing special to `sqlamp` here. Note self-reference "child to parent" ('parent_id' is foreign key to table's primary key) just as in any other implementation of adjacency relations.

```
import sqlamp
class Node(object):
    mp = sqlamp.MPManager(node_table)
    def __init__(self, name, parent=None):
        self.name = name
        self.parent = parent
    def __repr__(self):
        return '<Node %r>' % self.name
```

Attach instance of `MPManager` to class that represents node. The only required argument for `MPManager` constructor is the table object.

Now we can create the table and define the mapper (it is important to create table *after* `MPManager` was created as created `MPManager` appends three new columns and one index to the table):

```
node_table.create()
```

Setting up the mapper requires only one extra step — providing `Node.mp` as mapper extension:

```
mapper = sqlalchemy.orm.mapper(
    Node, node_table,
    extension=[Node.mp],
    properties={
        'parent': sqlalchemy.orm.relation(
            Node, remote_side=[node_table.c.id]
        )
    }
)
```

You may see value provided as `properties` argument: this is a way recommended by the official SQLAlchemy documentation to set up an adjacency relation.

Alternative way to set up: ext.declarative

Starting from version 0.5 it is able and convenient to use declarative approach to set your trees up:

```
import sqlalchemy, sqlalchemy.orm
from sqlalchemy.ext.declarative import declarative_base
import sqlamp

engine = sqlalchemy.create_engine('sqlite:///:memory:', echo=False)
metadata = sqlalchemy.MetaData(engine)

BaseNode = declarative_base(metadata=metadata,
                            metaclass=sqlamp.DeclarativeMeta)

class Node(BaseNode):
    __tablename__ = 'node'
    __mp_manager__ = 'mp'
    id = sqlalchemy.Column(sqlalchemy.Integer, primary_key=True)
    parent_id = sqlalchemy.Column(sqlalchemy.ForeignKey('node.id'))
    parent = sqlalchemy.orm.relation("Node", remote_side=[id])
    name = sqlalchemy.Column(sqlalchemy.String())
    def __init__(self, name, parent=None):
        self.name = name
        self.parent = parent
    def __repr__(self):
        return '<Node %r>' % self.name

Node.__table__.create()
```

As you can see it is pretty much the same as usual for sqlalchemy's "declarative" extension. Only two things here are sqlamp-specific: `metaclass` argument provided to `declarative_base()` factory function should be `DeclarativeMeta` and the node class should have an __mp_manager__ property with string value. See `DeclarativeMeta` for more information about options.

Now all the preparation steps are done. Lets try to use it!

```
session = sqlalchemy.orm.sessionmaker(engine)()
root = Node('root')
child1 = Node('child1', parent=root)
child2 = Node('child2', parent=root)
grandchild = Node('grandchild', parent=child1)
session.add_all([root, child1, child2, grandchild])
session.flush()
```

We have just created a sample tree. This is all about `AL`, nothing special to `sqlamp` here. The interesting part is fetching trees:

```
root.mp.query_children().all()
# should print [<Node 'child1'>, <Node 'child2'>]

root.mp.query_descendants().all()
# [<Node 'child1'>, <Node 'grandchild'>, <Node 'child2'>]

grandchild.mp.query_ancestors().all()
# [<Node 'root'>, <Node 'child1'>]
```

```
Node.mp.query(session).all()
# [<Node 'root'>, <Node 'child1'>, <Node 'grandchild'>, <Node 'child2'>]

for node in root.mp.query_descendants(and_self=True):
    print '  ' * node.mp_depth, node.name
# root
#   child1
#     grandchild
#   child2
```

As you can see all `sqlamp` functionality is accessible via `MPManager` descriptor (called `'mp'` in this example).

Note: `Node.mp` (a so-called "class manager") is not the same as `node.mp` ("instance manager"). Do not confuse them as they are for different purposes and their APIs has no similar. Class manager (see `MPClassManager`) exists for features that are not intended to particular node but for the whole tree: basic setup (mapper extension) and tree-maintenance functions. And an instance managers (`MPInstanceManager`) are each unique to and bounded to a node. They allow to make queries for related nodes and other things specific to concrete node. There is also third kind of values that `MPManager` descriptor may return, see `its reference` for more info.

CHAPTER 2

Implementation details

sqlamp borrowed some implementation ideas from django-treebeard. In particular, sqlamp uses the same alphabet (which consists of numeric digits and latin-letters in upper case), sqlamp as like as django-treebeard doesn't use path parts delimiter — path parts has fixed adjustable length. But unlike django-treebeard sqlamp stores each tree absolutely stand-alone — two or more trees may (and will) have identical values in path and depth fields and be different only by values in tree_id field. This is the way that can be found in django-mptt.

sqlamp works *only* on basis of Adjacency Relations. This solution makes data more denormalized but more fault-tolerant. It makes possible rebuilding all paths for all trees using only AL data. Also it makes applying sqlamp on existing project easier.

2.1 Limits

sqlamp imposes some limits on the amount of data in the tree. Those limits are configurable for any specific application. Here is the list of limits:

Number of trees: Imposed by TreeIdField, which is of type INTEGER. Therefore the limit of number of trees is defined by the highest integer for DBMS in use. This is not configurable. If you expect to have more than 2**31 trees you might want to use custom field for tree id with a different numeric type (supposedly BIGINT).

Number of children in each node: Imposed by the length of one path segment. Can be configured using the "steplen" parameter (see MPManager). The number of children in each node is equal to "36 ** steplen" and with default "steplen=3" is equal to 46656. Note that it is not the total maximum number of nodes in a tree. Each node can have that much children and each of it children can have that much children, and so on.

Maximum nesting depth: Imposed by length of path field and length of one path segment. Generally speaking the deepest nesting is equal to maximum possible number of path segments in a whole path. So it can be expressed as "pathlen // steplen + 1" (see MPManager). The default value for "pathlen" is for historical reason 255, so together with default steplen it sets the maximum depth to 86. Nowadays all major DBMS support a higher length for VARCHAR so you can freely increase "pathlen" to, say, 10240 and "steplen" to 4. These values would limit your tree to 1679616 maximum children and 2561 maximum depth.

Total number of nodes in a tree: There is no such limit. Not any that you can hit even with (extremely low) default path length. The total number of nodes in a tree is equal to "36 ** pathlen" and with "pathlen=255" it is something around 7.2e+397.

2.2 Moving nodes

There are several things to consider on moving nodes with materialized path. First of all, it can never be as efficient in terms of execution speed as plain adjacency lists for obvious reason. It can also be slower than nested sets, because

DBMS needs to do more work on rebuilding indices. Keep in mind that every time you move a subtree, for instance, from one parent to another, path, tree_id and depth fields have to be updated in each and every node in the whole tree/subtree you're moving and also in all subtrees that start from (both old and new) following siblings. If your application relies on extensive moving of nodes it might be better to stay with AL.

There is also some points to note in a way the moving of nodes is implemented. In order to achieve the best performance moving of nodes is not working on ORM level, instead it uses bulk update queries. The most important implication of that is that node objects which are stored in session **do not get updated** after moving is performed. Using them in new queries will inevitably produce wrong results. Therefore you need to make sure that you expire all (not only ones that belong to a moved tree!) node objects from the session after performing any moving or deleting operation. This implementation detail is reflected in fact that moving operations are incorporated to `MPClassManager` API (not instance managers) and accept primary keys instead of node objects.

Another caveat is in limits check. If you put a subtree deep inside another one it may be possible to overlook that for some nodes the result path can be longer than accepted value by the path field (see limits for details). Unfortunately it is impossible to check in advance without doing (probably expensive) queries to find out the deepest path in the subtree that is been moving.

The API for moving nodes comprises the following operations (all of them are methods of `MPClassManager`):

- `detach_subtree()` – for creating new distinct tree from a part of another tree;
- `delete_subtree()` – for deleting subtree or whole tree without leaving gaps in paths;
- `move_subtree_before()` and `move_subtree_after()` – for moving a tree/subtree basing on siblings (useful also for changing the child nodes order);
- `move_subtree_to_top()` and `move_subtree_to_bottom()` – for moving nodes based on specified new parent node.

The last four methods raise `TooManyChildrenError` if new parent node already has `36 ** steplen` children and can not accept one more child node. They also raise `MovingToDescendantError` if a new parent node is one of descendants of moved node.

CHAPTER 3

Support

You can either email author directly to send bugreports, feature requests and patches or use issue tracking on bitbucket project's page.

CHAPTER 4

API Reference

exception sqlamp.**PathOverflowError**

Base class for exceptions in calculations of node's path.

exception sqlamp.**TooManyChildrenError**

Maximum children limit is exceeded. Raised during flush.

exception sqlamp.**PathTooDeepError**

Maximum depth of nesting limit is exceeded. Raised during flush.

exception sqlamp.**MovingToDescendantError**

An attempt to move a tree inside of one of its descendants was made.

See moving nodes.

class sqlamp.**MPManager**(*table*, *parent_id_field=None*, *path_field='mp_path'*, *depth_field='mp_depth'*, *tree_id_field='mp_tree_id'*, *steplen=3*, *instance_manager_key='_mp_instance_manager'*)

Descriptor for access class-level and instance-level API.

Basic usage is simple:

```
class Node(object):
    mp = sqlamp.MPManager(node_table)
```

Now there is an ability to get instance manager or class manager via property `'mp'` depending on way to access it. `Node.mp` will return mapper extension till class is mapped, class manager `MPClassManager` after that and `instance_node.mp` will return instance_node's `MPInstanceManager`. See that classes for more details about their public API.

Changed in version 0.5.1: Previously mapper extension was accessible via class manager's property.

Parameters

- **table** – instance of `sqlalchemy.Table`. A table that will be mapped to node class and will hold tree nodes in its rows. It is the only one strictly required argument.
- **parent_id_field=None** – a foreign key field that is reference to 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 onclause as parent id field.
- **path_field='mp_path'** – the name for the path 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 value provided is an object column some sanity checks will be performed with the column object: it should have `nullable=False` and have `PathField` type.

- **depth_field='mp_depth'** – the same as for `path_field`, except that the type of this column should be `DepthField`.
- **tree_id_field='mp_tree_id'** – the same as for `path_field`, except that the type of this column should be `TreeIdField`.
- **pathlen=255** – an integer, the length for path field. See limits for details.
- **steplen=3** – an integer, the number of characters in each part of the path. See limits.
- **instance_manager_key='_mp_instance_manager'** – name for node instance's attribute to cache node's instance manager.

Warning: Do not change the values of `MPManager` constructor's attributes after saving a first tree node. Doing this will corrupt the tree.

`__get__`(*obj*, *objtype*)

There may be three kinds of return values from this getter.

The first one is used when the class which this descriptor is attached to is not yet mapped to any table. In that case the return value is an instance of `MPMapperExtension`. which is intended to be used as mapper extension.

The second scenario is access to `MPManager` via mapped class. The corresponding `MPClassManager` instance is returned.

Note: If the nodes of your tree use polymorphic inheritance it is important to know that class manager is accessible only via the base class of inheritance hierarchy.

And the third way is accessing it from the node instance. Attached to that node `instance manager` is returned then.

class `sqlamp.`**`DeclarativeMeta`**(*name*, *bases*, *dct*)

Metaclass for declaratively defined node model classes.

New in version 0.5.

See *usage example* above in Quickstart.

All options that accepts `MPManager` can be provided with declarative definition. To provide an option you can simply assign value to class' property with name like `__mp_tree_id_field__` (for `tree_id_field` parameter) and so forth. See the complete list of options in `MPManager`'s constructor parameters. Note that you can use only string options for field names, not the column objects.

A special class variable `__mp_manager__` should exist and hold a string name which will be used as `MPManager` descriptor property.

class `sqlamp.`**`MPClassManager`**(*node_class*, *opts*)

Node class manager. No need to create it by hand: it's created by `MPManager`.

Parameters

- **node_class** – class which was mapped to the tree table.
- **opts** – instance of `MPOptions`.

Changed in version 0.6: Previously existing method `__clause_element__` which used to allow using the instances of `MPClassManager` as arguments to methods `query.order_by()` was removed in 0.6. Use `query()` instead.

`create_indices`(*session*)

Create mp-related indices.

Note that needed indices are created by default if you use sqlalchemy's DDL facility (like `table.create()`) on mp-armed table. This method is useful after call to `rebuild_all_trees()`, or when you're setting up sqlamp on an existing table.

Parameters **session** – sqlalchemy `Session` object to bind DDL queries.

New in version 0.6.

`delete_subtree`(*session*, *node_id*)

Delete a whole tree/subtree starting from root `node_id`.

Parameters

- **session** – session object for DML queries.
- **node_id** – primary key of root of tree/subtree to be deleted.

This method differs from performing something like

```
# leaves gaps!
node.mp.query_descendants(session, and_self=True).delete()
```

because it updates the paths of following siblings in order to make sure that the tree limits are not lowered artificially no matter how many deletion operations have been performed. Therefore the shown code is not recommended way for deleting nodes. Use `delete_subtree()` instead.

Note: If you use MySQL InnoDB you need to set child-to-parent foreign key as `ON DELETE CASCADE` in order for this method to work for non-empty trees/subtrees. This is because of InnoDB's inability to defer constraint check to the end of statement execution. See relevant innodb docs.

See also general notes on moving nodes.

`detach_subtree`(*session*, *node_id*)

Create a new distinct tree with root `node_id`.

Expects that new root node is a part of another tree (has non-empty parent id). New root node and all its descendants will have new `tree_id` and adjusted paths.

Parameters

- **session** – session object for DML queries.
- **node_id** – primary key of to-be-new-root node.

See also general notes on moving nodes.

`drop_indices`(*session*)

Drop mp-related indices.

Note that in general you need this only in conjunction with `rebuild_all_trees()`, which this method used to be a part of.

Parameters **session** – sqlalchemy `Session` object to bind DDL queries.

New in version 0.6.

`max_children`

The maximum number of children in each node, readonly.

`max_depth`

The maximum level of nesting in this tree, readonly.

`move_subtree_after`(*session*, *node_id*, *anchor_id*)

The same as `move_subtree_before()` but makes target tree/subtree root the immediately following sibling of anchor node.

`move_subtree_before`(*session*, *node_id*, *anchor_id*)

Move tree/subtree starting from `node_id` to make it preceding sibling of node with pk `anchor_id`. Anchor node is expected not to be a root of own tree.

Parameters

- **session** – session object for DML queries.
- **node_id** – primary key of root of tree/subtree to be moved.
- **anchor_id** – primary key of a node which target node should become previous sibling to.

See also general notes on moving nodes.

`move_subtree_to_bottom`(*session*, *node_id*, *new_parent_id*)

The same as `move_subtree_before()` but makes target tree/subtree root the last child of anchor node.

`move_subtree_to_top`(*session*, *node_id*, *new_parent_id*)

Move tree/subtree starting from `node_id` to make it the first child of node with pk `anchor_id`.

Parameters

- **session** – session object for DML queries.
- **node_id** – primary key of root of tree/subtree to be moved.
- **anchor_id** – primary key of a node which should become a new parent for target node.

See also general notes on moving nodes.

`query`(*session*)

Query all stored trees.

Parameters **session** – a sqlalchemy `Session` object to bind a query.

Returns `Query` object with all nodes of all trees sorted as usual by `(tree_id, path)`.

Changed in version 0.6: Before 0.6 this method was called `query_all_trees`. The old name still works for backward compatibility.

`rebuild_all_trees`(*session*, *order_by=None*)

Perform a complete rebuild of all trees on the basis of adjacency relations.

Parameters

- **session** – a session object which will be used for DML-queries. The session's transaction gets commited when rebuilding is done.
- **order_by** – an "order by clause" for sorting root nodes and children nodes in each subtree. By default ordering by primary key is used.

Changed in version 0.6: `rebuild_all_trees()` didn't receive `session` parameter prior to 0.6.

Warning: This method no longer drops/creates indices!

Changed in version 0.6: Before 0.6 this method was dropping mp-related indices before starting to modify table content and recreating them afterwards. Now these parts are factored out to `drop_indices()` and `create_indices()` respectively.

class `sqlamp.`**`MPInstanceManager`**(*opts*, *root_node_class*, *obj*)

A node instance manager, unique for each node. First created on access to `MPManager` descriptor from instance. Implements API to query nodes related somehow to particular node: descendants, ancestors, etc.

Parameters

- **opts** – instance of `MPOptions`.
- **root_node_class** – the root class in the node class' polymorphic inheritance hierarchy. This class will be used to perform queries.
- **obj** – particular node instance.

`filter_ancestors`(*and_self=False*)

The same as `filter_descendants()` but filters ancestor nodes.

`filter_children`()

The same as `filter_descendants()` but filters children nodes and does not accepts `and_self` parameter.

`filter_descendants`(*and_self=False*)

Get a filter condition for node's descendants.

Requires that node has `path`, `tree_id` 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:

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

This example is silly and only shows an approach of using `filter_descendants`. Don't use it for such purpose as there is a better way for such simple queries: `query_descendants()`.

Parameters **and_self** – `bool`, if set to `True` self node will be selected by filter.

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

`query_ancestors`(*session=None*, *and_self=False*)

The same as `query_descendants()` but queries node's ancestors.

`query_children`(*session=None*)

The same as `query_descendants()` but queries children nodes and does not accepts `and_self` parameter.

`query_descendants`(*session=None*, *and_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).
- **and_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 and is ordered by `path`.

`sqlamp.`**`tree_recursive_iterator`**(*flat_tree*, *class_manager*)

Make a recursive iterator from plain tree nodes sequence (`Query` instance for example). Generates two-item tuples: node itself and it's children collection (which also generates two-item tuples...) Children collection evaluates to `False` if node has no children (it is zero-length tuple for leaf nodes), else it is a generator object.

Parameters

- **flat_tree** – plain sequence of tree nodes.
- **class_manager** – instance of `MPClassManager`.

Can be used when it is simpler to process tree structure recursively. Simple usage example:

```
def recursive_tree_processor(nodes):
    print '<ul>'
    for node, children in nodes:
        print '<li>%s' % node.name,
        if children:
            recursive_tree_processor(children)
        print '</li>'
    print '</ul>'

query = root_node.mp.query_descendants(and_self=True)
recursive_tree_processor(
    sqlamp.tree_recursive_iterator(query, Node.mp)
)
```

Changed in version 0.6: Before this function was sorting `flat_tree` if it was a query-object. Since 0.6 it doesn't do it, so make sure that `flat_tree` is properly sorted. The best way to achieve this is using queries returned from public API methods of `MPClassManager` and `MPInstanceManager`.

Warning: Process `flat_tree` items once and sequentially so works right only if used in depth-first recursive consumer.

class sqlamp.**PathField**
Varchar field subtype representing node's path.

class sqlamp.**DepthField**
Integer field subtype representing node's depth level.

class sqlamp.**TreeIdField**
Integer field subtype representing node's tree identifier.

CHAPTER 5

Changelog

5.1 0.6: released 2012-01-12

The most exciting things in 0.6 are the moving nodes feature, support of SQLAlchemy 0.5, 0.6 and 0.7 as well as python 2.4 to 3.

Adding support of 0.7 required some backward-incompatible changes in public API, from which the biggest one is inability to use `MPClassManager` instances as arguments for `order_by()`. If you use `query*` methods from `MPClassManager` and `MPInstanceManager` you don't need to worry — query objects which come from there are properly ordered. If you use old semantic like `session.query(Node).order_by(Node.mp)` you will need to convert such code to use `MPClassManager.query()`, like `Node.mp.query(session)`.

Another similar change is that `tree_recursive_iterator()` doesn't reorder the argument provided, so if you construct your queries by hand (which is not recommended) — make sure that they're properly ordered.

- Moving nodes support, see Moving nodes.
- Custom path field length can now be used easily, see Limits.
- `MPClassManager.rebuild_subtree()` was dropped altogether in favor of real moving nodes API and real `maintenance` functionality.
- `MPClassManager.rebuild_all_trees()` now accepts required session parameter. Original patch by Uriy Zhuravlev.
- Pickling and unpickling of node instances now works.
- Python branches 2.4 to 2.7 as well as python3 are supported now.
- `MPClassManager` can not be used as an argument for `order_by()`. Instead use method `MPClassManager.query()` for constructing queries.
- Method `MPClassManager.query_all_trees()` was renamed to `query()`. The old name still works though.
- `tree_recursive_iterator()` doesn't reorder query argument anymore.
- Added support of SQLAlchemy 0.7.2.
- Documentation was cleaned up and updated.
- Workaround for bug in sqlite 3.6.x (problems with binding two integer attributes). Initial patch by Josip Delic.

5.2 0.5.2: released 2010-09-19

- SQLAlchemy of versions 0.6.x is now supported as well as 0.5.x.
- `tree_recursive_iterator()` does not require python 2.6+ anymore, python 2.5 is now fully supported.

5.3 0.5.1: released 2009-11-29

- `mapper_extension` property now removed from class manager, descriptor `MPManager` returns it instead of class manager if the class is not mapped yet.
- Joined table inheritance now supported both in imperative and declarative ways: thanks to Laurent Rahuel for testing.

5.4 0.5: released 2009-09-05

This release contains some backward-incompatible changes in setup facilities. The main highlights are support of `declarative` SQLAlchemy extension and some cleaning up in `MPManager`'s constructor options.

- Index name now includes table name as prefix so there is an ability to have two or more mp-driven tables in the same database.
- There is only one strictly required option for `MPManager` now: the table object. `pk_field` option removed at all (can be safely determined from the table) and `parent_id_field` could be guessed for almost every simple table (if yours are not so simple you can provide this option as it was with 0.4.x).
- changed names of `path_field`, `depth_field` and `tree_id_field` parameters of `MPManager`'s constructor: removed `_name` suffix: now the values can be column objects and they are not redefined if such a column exists already.

5.5 0.4.1: released 2009-07-16

- Fixed another bug in `MPClassManager.rebuild_all_trees()`: tree_id and depth for root nodes were not updated. Method also was slightly optimized to do less queries.
- Small fixes in documentation.

5.6 0.4: released 2009-06-11

- Small fixes in documentation: actually Tropashko was not the first who introduced MP, he only promoted it.
- Implemented `MPClassManager.query_all_trees()`.
- Fixed a bug of `MPClassManager.rebuild_all_trees()` did not reset path for root nodes.
- Implemented `tree_recursive_iterator()`.
- Changed the value of path field for a root nodes. Previously they used to had `'0' * steplen` path and so first-level children gain `'0' * steplen * 2`, but now new roots will have an empty string in their path field. This change should be backward-compatible as it touches only new trees. But if you want to have no difference between two identical old and new trees in your table you can rebuild all your trees by `Node.mp.rebuild_all_trees()` or use sql query like this:

```
UPDATE <table> SET mp_path = substr(mp_path, <steplen> + 1);
```

This will remove `steplen` characters from the beginning of each node's path. **Do not forget to backup your data!**

5.7 0.3: released 2009-05-23

The first public release.