libtree Documentation

Release 1.0.0

Fabian Kochem

October 28, 2015

1 Contents **3**
1.1 Installation . . . 3
1.2 Quickstart . . . 3
1.3 User Guide . . . 4
1.4 Tree object . . . 6
1.5 Transaction object . . . 7
1.6 Node object . . . 8
1.7 Core functions . . . 9
1.8 Benchmarks . . . 15
1.9 Database Model . . . 15

Python Module Index **19**

libtree is a Python library which assists you in dealing with **large, hierarchical data sets**. It runs on top of **PostgreSQL 9.4** and is compatible with **all major Python interpreters** (2.7, 3.3-3.5, PyPy2 and PyPy3).

Why use **libtree**? Because...

- the usage is **super simple** (see *Quickstart*)
- it scales up to **billions of nodes** (see *Database Model*)
- the reads and writes are **blazingly fast** (*Benchmarks* will be available soon)
- it supports **attribute inheritance** (see *Property functions*)

But there's even more, **libtree**...

- offers **thread-safety** by working inside transactions
- enforces **integrity** by moving tree logic to inside the database
- provides a **convenient** high level API and **fast** low level functions
- core is **fully integration tested**, the testsuite covers >90% of the code

CHAPTER 1

Contents

1.1 Installation

Install libtree directly via:

```
$ pip install libtree
```

in your virtualenv.

1.2 Quickstart

Install `libtree` via `pip install libtree`. Then start the interactive Python interpreter of your choice to start working with `libtree`:

```
# Imports
from libtree import Tree
import psycopg2

# Connect to PostgreSQL
connection = psycopg2.connect("dbname=test_tree user=vortec")
tree = Tree(connection)

# Start working with libtree inside a database transaction
with tree() as transaction:

    # Create tables
    transaction.install()

    # Create nodes
    root = transaction.insert_root_node()
    binx = root.insert_child({'title': 'Binary folder'})
    bash = binx.insert_child({'title': 'Bash executable', 'chmod': 755})
    etc = root.insert_child({'title': 'Config folder'})
    hosts = etc.insert_child({'title': 'Hosts file'})
    passwd = etc.insert_child({'title': 'Password file', 'chmod': 644})

    # Direct attribute access
    root.children  # => binx, etc
    len(root)  # => 2
    binx.parent  # => root
    bash.ancestors  # => binx, root
    root.descendants  # => binx, bash, etc, hosts, passwd

    # Query by property
    transaction.get_nodes_by_property_key('chmod')  # bash, passwd
    transaction.get_nodes_by_property_dict({'chmod': 644})  # passwd

    # Move bash node into etc node
    bash.move(etc)
    etc.children  # => hosts, passwd, bash
    bash.set_position(1)
    etc.children  # => hosts, bash, passwd

    # Print entire tree
    transaction.print_tree()
    # Output:
    # <NodeData id=1, parent=None>
    #   <NodeData id=2, title='Binary folder'>
    #   <NodeData id=4, title='Config folder'>
    #     <NodeData id=5, title='Hosts file'>
    #     <NodeData id=6, title='Password file'>
    #     <NodeData id=3, title='Bash executable'>
```

1.3 User Guide

1.3.1 Database Connection

To start working with *libtree*, make sure PostgreSQL 9.4 is running. If you don't have a database yet, create one now:

```
$ createdb libtree
```

Next, start a Python interpreter, import libtree and create a *Tree object* object. To make it connect to PostgreSQL, you must create a *psycopg2* connection. After that, you can install libtree:

```
$ python
Python 3.4.2 (default, Nov 12 2014, 10:43:21)
>>> from libtree import Tree
>>> import psycopg2
>>> connection = psycopg2.connect("dbname=test_tree user=vortec")
>>> transaction = Tree(connection).make_transaction()
>>> transaction.install()
>>> transaction.commit()
```

The `transaction` object represents a database transaction and must be passed to every function when you want to query or modify the tree. Running `install()` creates the SQL tables and must only be executed if you haven't done so before. Executing `commit()` writes the changes you made to the database. If you want to discard the changes, run `transaction.rollback()`.

For more convenience, you can use the auto-committing context manager:

```
>>> tree = Tree(connection)
>>> with tree() as transaction:
...     transaction.install()
```

When the context manager leaves it will commit the transaction to the database. If an exception occurs, it will rollback automatically.

1.3.2 Modify the tree

Now, you can create some nodes:

```
>>> html = transaction.insert_root_node()
>>> title = html.insert_child({'title': 'title', 'content': 'libtree'})
>>> head = html.insert_child({'title': 'head'})
>>> body = html.insert_child({'title': 'body'})
>>> h2 = body.insert_child({'title': 'h2', 'content': 'to libtree'})
>>> transaction.commit()
```

This should render as a nice, DOM-like tree:

```
>>> transaction.print_tree()
<NodeData id=1>
  <NodeData id=2, title='title'>
  <NodeData id=3, title='head'>
  <NodeData id=4, title='body'>
    <NodeData id=5, title='h2'>
```

But do you spot the mistake? In HTML, a `<title>` tag goes beneath the `<head>` tag, so let's move it:

```
>>> title.move(head)
>>> transaction.print_tree()
<NodeData id=1>
  <NodeData id=3, title='head'>
    <NodeData id=2, title='title'>
  <NodeData id=4, title='body'>
    <NodeData id=5, title='h2'>
```

And you also forgot the `<h1>` node, let's insert it before `<h2>`:

```
>>> body.insert_child({'title': 'h1', 'content': 'Welcome'}, position=0)
<Node id=6, title='h1'>
>>> transaction.print_tree()
<NodeData id=1>
  <NodeData id=3, title='head'>
    <NodeData id=2, title='title'>
  <NodeData id=4, title='body'>
    <NodeData id=6, title='h1'>
    <NodeData id=5, title='h2'>
```

Since you know the ID, you can easily delete nodes without a `Node` object:

```
>>> h2.delete()
>>> transaction.print_tree()
<NodeData id=1>
  <NodeData id=3, title='head'>
    <NodeData id=2, title='title'>
  <NodeData id=4, title='body'>
    <NodeData id=6, title='h1'>
>>> transaction.commit()
```

1.3.3 Query the tree

If you want to get a `Node` object, you can easily get one by querying for the ID:

```
>>> title.properties
{'content': 'libtree', 'title': 'title'}
```

You can get the immediate children of a node:

```
>>> html.children
[<Node id=3, title='head'>, <Node id=4, title='body'>]
```

You can get all nodes that have a certain property key set:

```
>>> transaction.get_nodes_by_property_key('content')
{<Node id=6, title='h1'>, <Node id=2, title='title'>}
```

Or ask for nodes that have a certain property value set:

```
>>> transaction.get_nodes_by_property_value('content', 'Welcome')
{<Node id=6, title='h1'>}
```

If you have a node, you can output the path from the root node to it too:

```
>>> h1.ancestors
[<Node id=1>, <Node id=4, title='body'>]
```

1.4 Tree object

class Tree (*connection=None*, *pool=None*, *node_factory=<class 'libtree.node.Node'>*)

Context manager for creating thread-safe transactions in which libtree operations can be executed.

It yields a *libtree.transaction.Transaction* object which can be used for accessing the tree. When the context manager gets exited, all changes will be committed to the database. If an exception occured, the transaction will be rolled back.

It requires either a `connection` or `pool` object from the psycopg2 package.

When libtree is used in a threaded environment (usually in production), it's recommended to use a pool object.

When libtree is used in a single-threaded environment (usually during development), it's enough to pass a standard connection object.

By default the built-in *libtree.node.Node* class is used to create node objects, but it's possible to pass a different one via `node_factory`.

Parameters

- **connection** (*psycopg2.connection*) – psycopg2 connection object
- **pool** (*psycopg2.pool.ThreadedConnectionPool*) – psycopg2 pool object
- **node_factory** (*object*) – Factory class for creating node objects (default: *libtree.node.Node*)

close ()

Close all connections in pool or the manually assigned one.

make_transaction ()

Get a new transaction object using a connection from the pool or the manually assigned one.

1.5 Transaction object

class **`Transaction`**(*connection*, *node_factory*)
Representation of a database transaction and entrypoint for global tree operations.

Parameters

- **`connection`** (*Connection*) – Postgres connection object. Its `autocommit` attribute will be set to `False`.
- **`node_factory`** (*object*) – Factory class for creating node objects

`clear`()
Empty database tables.

`commit`()
Write changes to databases. See commit() .

`get_node`(*node_id*)
Get node with given database ID.

Parameters **`node_id`** (*int*) – Database ID

`get_nodes_by_property_dict`(*query*)
Get a set of nodes which have all key/value pairs of `query` in their properties. Inherited properties are not considered.

Parameters **`query`** (*dict*) – The dictionary to search for

`get_nodes_by_property_key`(*key*)
Get a set of nodes which have a property named `key` in their properties. Inherited properties are not considered.

Parameters **`key`** (*str*) – The key to search for

`get_nodes_by_property_value`(*key*, *value*)
Get a set of nodes which have a property `key` with value `value`. Inherited properties are not considered.

Parameters

- **`key`** (*str*) – The key to search for
- **`value`** (*object*) – The exact value to sarch for

`get_root_node`()
Get root node if exists, other `None`.

`get_tree_size`()
Get amount of nodes inside the tree.

`insert_root_node`(*properties=None*)
Create root node, then get it.

Parameters **`properties`** (*dict*) – Inheritable key/value pairs (see *Property functions*)

`install`()
Create tables and trigger functions in database.

`is_compatible_postgres_version`()
Determine whether PostgreSQL server version is compatible with libtree.

`print_tree`()
Simple function to print tree structure to stdout.

rollback()
Discard changes. See rollback() .

uninstall()
Remove libtree tables from database.

1.6 Node object

class **Node**(*transaction*, *id*)
Representation of a tree node and entrypoint for local tree operations.

It's a thin wrapper around the underlaying core functions. It does not contain any data besides the database ID and must therefore query the database every time the value of an attribute like `parent` has been requested. This decision has been made to avoid race conditions when working in concurrent or distributed environments, but comes at the cost of slower runtime execution speeds. If this becomes a problem for you, grab the the corresponding *`libtree.core.node_data.NodeData`* object via *`libtree.node.Node.node_data`*.

This object is tightly coupled to a *`libtree.transaction.Transaction`* object. It behaves like a partial which passes a database cursor and node ID into every `libtree.core` function. It also has a few convenience features like attribute access via Python properties and shorter method names.

Parameters

- **transaction** (Transaction) – Transaction object
- **id** (*int*) – Database node ID

__len__()
Return amount of child nodes.

__eq__(*other*)
Determine if this node is equal to `other`.

ancestors
Get list of ancestor nodes.

children
Get list of immediate child nodes.

delete()
Delete node and its subtree.

descendants
Get list of descendant nodes.

get_child_at_position(*position*)
Get child node at certain position.

Parameters **position** (*int*) – Position to get the child node from

id
Database ID

inherited_properties
Get inherited property dictionary.

insert_child(*properties=None*, *position=-1*)
Create a child node and return it.

Parameters

- **properties** (*dict*) – Inheritable key/value pairs (see *Property functions*)

- **position** (*int*) – Position in between siblings. If 0, the node will be inserted at the beginning of the parents children. If -1, the node will be inserted the the end of the parents children.

move (*target*, *position=-1*)
Move node and its subtree from its current to another parent node. Raises `ValueError` if `target` is inside this nodes' subtree.

Parameters

- **target** (Node) – New parent node
- **position** (*int*) – Position in between siblings. If 0, the node will be inserted at the beginning of the parents children. If -1, the node will be inserted the the end of the parents children.

node_data
Get a *libtree.core.node_data.NodeData* object for current node ID from database.

parent
Get parent node.

position
Get position in between sibling nodes.

properties
Get property dictionary.

set_position (*new_position*)
Set position.

Parameters **position** (*int*) – Position in between siblings. If 0, the node will be inserted at the beginning of the parents children. If -1, the node will be inserted the the end of the parents children.

set_properties (*properties*)
Set properties.

Parameters **properties** (*dict*) – Property dictionary

swap_position (*other*)
Swap position with `other` position.

Parameters **other** (Node) – Node to swap the position with

1.7 Core functions

1.7.1 Database functions

create_schema (*cur*)
Create table schema.

create_triggers (*cur*)
Create triggers.

drop_tables (*cur*)
Drop all tables.

flush_tables (*cur*)
Empty all tables.

is_compatible_postgres_version(*cur*)

Determine whether PostgreSQL server version is compatible with libtree.

1.7.2 NodeData class

class **NodeData**(*id=None*, *parent=None*, *position=None*, *properties=None*)

Immutable data-holding object which represents tree node data. Its attributes are identical to the columns in the `nodes` table (see *Database Model*).

Since the object is immutable, you must retrieve a new instance of the same node using *`libtree.core.query.get_node()`* to get updated values.

To manipulate the values, you must use one of the following functions:

- *`libtree.core.tree.change_parent()`*
- *Positioning functions*
- *Property functions*

Most `libtree` functions need a database ID in order to know on which data they should operate, but also accept `Node` objects to make handling with them easier.

All parameters are optional and default to `None`.

Parameters

- **id** (*int*) – ID of the node as returned from the database
- **parent** (*Node or int*) – Reference to a parent node
- **position** (*int*) – Position in between siblings (see *Positioning functions*)
- **properties** (*dict*) – Inheritable key/value pairs (see *Property functions*)

id

Node ID

parent

Parent ID

position

Position in between its siblings

properties

Node properties

to_dict()

Return dictionary containing all values of the object.

1.7.3 Positioning functions

Auto position

`libtree` has a feature called *auto position* which is turned on by default and makes sure that whenever you insert, move or delete a node its siblings stay correctly ordered.

Let's assume you have a node sequence like this:

```
position | 0 | 1
node     | A | B
```

If you now insert a new node without any further arguments, auto position will insert it at the end of the sequence:

```
position | 0 | 1 | 2
node     | A | B | C
```

But if you insert the node at a certain position (1 in this example), auto position will free the desired spot and shift the following siblings to the right like this:

```
position | 0 | 1 | 2
node     | A | C | B
```

Likewise, if you want to delete the node at position 1, auto position will left-shift all following nodes, so you end up with the same sequence as at the beginning again.

This is default behaviour because most users expect a tree implementation to behave like this.

Disable auto position

If you're working on a dataset in which you know the final positions of your nodes before feeding them into `libtree`, you can disable auto position altogether. This means lesser queries to the database and thus, faster insert speeds. On the other hand this means that no constraint checks are being made and you could end up with non-continuative position sequences, multiple nodes at the same position or no position at all. Don't worry - libtree supports those cases perfectly well - but it might be confusing in the end.

To disable auto position you must pass `auto_position=False` to any function that manipulates the tree (see *Tree functions*).

API

Related: `libtree.query.get_node_at_position()`

`ensure_free_position`(*cur*, *node*, *position*)

Move siblings away to have a free slot at `position` in the children of `node`.

Parameters

- **`node`** (*Node or int*) –
- **`position`** (*int*) –

`find_highest_position`(*cur*, *node*)

Return highest, not occupied position in the children of `node`.

Parameters **`node`** (*Node or int*) –

`set_position`(*cur*, *node*, *position*, *auto_position=True*)

Set `position` for `node`.

Parameters

- **`node`** (*Node or int*) –
- **`position`** (*int*) – Position in between siblings. If 0, the node will be inserted at the beginning of the parents children. If -1, the node will be inserted the the end of the parents children. If *auto_position* is disabled, this is just a value.
- **`auto_position`** (*bool*) – See *Positioning functions*

`shift_positions`(*cur*, *node*, *position*, *offset*)

Shift all children of `node` at `position` by `offset`.

Parameters

- **`node`** (*Node or int*) –

- **`position`** (*int*) –
- **`offset`** (*int*) – Positive value for right shift, negative value for left shift

`swap_node_positions`(*cur*, *node1*, *node2*)
Swap positions of `node1` and `node2`.

Parameters

- **`node1`** (*Node or int*) –
- **`node2`** (*Node or int*) –

1.7.4 Property functions

`get_inherited_properties`(*cur*, *node*)
Get the entire inherited property dictionary.

To calculate this, the trees path from root node till `node` will be traversed. For each level, the property dictionary will be merged into the previous one. This is a simple merge, only the first level of keys will be combined.

Parameters **`node`** (*Node or int*) –

Return type dict

`get_inherited_property_value`(*cur*, *node*, *key*)
Get the inherited value for a single property key.

Parameters

- **`node`** (*Node or int*) –
- **`key`** – str

`get_nodes_by_property_dict`(*cur*, *query*)
Return an iterator that yields a `NodeData` object of every node which contains all key/value pairs of `query` in its property dictionary. Inherited keys are not considered.

Parameters **`query`** (*dict*) – The dictionary to search for

`get_nodes_by_property_key`(*cur*, *key*)
Return an iterator that yields a `NodeData` object of every node which contains `key` in its property dictionary. Inherited keys are not considered.

Parameters **`key`** (*str*) – The key to search for

`get_nodes_by_property_value`(*cur*, *key*, *value*)
Return an iterator that yields a `NodeData` object of every node which has `key` exactly set to `value` in its property dictionary. Inherited keys are not considered.

Parameters

- **`key`** (*str*) – The key to search for
- **`value`** (*object*) – The exact value to sarch for

`set_properties`(*cur*, *node*, *new_properties*)
Set the property dictionary to `new_properties`. Return `NodeData` object with updated properties.

Parameters

- **`node`** (*Node or int*) –
- **`new_properties`** – dict

set_property_value (*cur*, *node*, *key*, *value*)

Set the value for a single property key. Return `NodeData` object with updated properties.

Parameters

- **node** (*Node or int*) –
- **key** – str
- **value** – object

update_properties (*cur*, *node*, *new_properties*)

Update existing property dictionary with another dictionary. Return `NodeData` object with updated properties.

Parameters

- **node** (*Node or int*) –
- **new_properties** – dict

1.7.5 Query functions

get_ancestor_ids (*cur*, *node*)

Return an iterator that yields the ID of every element while traversing from `node` to the root node.

Parameters **node** (*Node or int*) –

get_ancestors (*cur*, *node*, *sort=True*)

Return an iterator that yields a `NodeData` object of every element while traversing from `node` to the root node.

Parameters

- **node** (*Node or int*) –
- **sort** (*bool*) – Start with closest node and end with root node. (default: True)

get_child_ids (*cur*, *node*)

Return an iterator that yields the ID of every immediate child.

Parameters **node** (*Node or int*) –

get_children (*cur*, *node*)

Return an iterator that yields a `NodeData` object of every immediate child.

Parameters **node** (*Node or int*) –

get_children_count (*cur*, *node*)

Get amount of immediate children.

Parameters **node** (*Node or int*) – Node

get_descendant_ids (*cur*, *node*)

Return an iterator that yields a `NodeData` object of each element in the nodes subtree. Be careful when converting this iterator to an iterable (like list or set) because it could contain billions of objects.

Parameters **node** (*Node or int*) –

get_descendants (*cur*, *node*)

Return an iterator that yields the ID of every element while traversing from `node` to the root node.

Parameters **node** (*Node or int*) –

get_node (*cur*, *id*)

Return `NodeData` object for given `id`. Raises `ValueError` if ID doesn't exist.

Parameters **id** (*int*) – Database ID

get_node_at_position (*cur*, *node*, *position*)
Return node at `position` in the children of `node`.

Parameters

- **node** (*Node or int*) –
- **position** (*int*) –

get_root_node (*cur*)
Return root node. Raise `ValueError` if root node doesn't exist.

get_tree_size (*cur*)
Return the total amount of tree nodes.

1.7.6 Tree functions

change_parent (*cur*, *node*, *new_parent*, *position=None*, *auto_position=True*)
Move node and its subtree from its current to another parent node. Return updated `Node` object with new parent set. Raise `ValueError` if `new_parent` is inside `node` s subtree.

Parameters

- **node** (*Node or int*) –
- **new_parent** (*Node or int*) – Reference to the new parent node
- **position** (*int*) – Position in between siblings. If 0, the node will be inserted at the beginning of the parents children. If -1, the node will be inserted the the end of the parents children. If *auto_position* is disabled, this is just a value.
- **auto_position** (*bool*) – See *Positioning functions*.

delete_node (*cur*, *node*, *auto_position=True*)
Delete node and its subtree.

Parameters

- **node** (*Node or int*) –
- **auto_position** (*bool*) – See *Positioning functions*

insert_node (*cur*, *parent*, *properties=None*, *position=None*, *auto_position=True*)
Create a `Node` object, insert it into the tree and then return it.

Parameters

- **parent** (*Node or int*) – Reference to its parent node. If *None*, this will be the root node.
- **properties** (*dict*) – Inheritable key/value pairs (see *Property functions*)
- **position** (*int*) – Position in between siblings. If 0, the node will be inserted at the beginning of the parents children. If -1, the node will be inserted the the end of the parents children. If *auto_position* is disabled, this is just a value.
- **auto_position** (*bool*) – See *Positioning functions*

print_tree (*cur*, *start_node=None*, *indent=' '*, *_level=0*)
Print tree to stdout.

Parameters

- **start_node** (*int, Node, NodaData or None*) – Starting point for tree output. If `None`, start at root node.

- **indent** (*str*) – String to print per level (default: ' ')

1.8 Benchmarks

Available soon.

1.9 Database Model

libtree aims to support billions of nodes while guaranteeing fast reads and fast writes. Well-known SQL solutions like Adjacency List or Nested Set have drawbacks which hinder performance in either direction. A very good model to achieve high performance is called *Closure Table*, which is explained here.

1.9.1 Closure Table

In Closure Table, you have two tables. One contains the node metadata, the other one contains every possible ancestor/descendant combination. In libtree, here's what they look like:

```
CREATE TABLE nodes
(
  id serial NOT NULL,
  parent integer,
  "position" smallint DEFAULT NULL,
  properties jsonb NOT NULL,
  CONSTRAINT "primary" PRIMARY KEY (id)
)
```

This is pretty simple and should be self-explanatory. Note that libtree uses the Adjacency List-style `parent` column, even though it's possible to drag this information out of the ancestor table (see below). This is mainly for speed reasons as it avoids a JOIN operation onto a huge table.

The more interesting bit is the ancestor table:

```
CREATE TABLE ancestors
(
  node integer NOT NULL,
  ancestor integer NOT NULL,
  CONSTRAINT idx UNIQUE (node, ancestor)
)
```

In this table, every tree relation is stored. This means not only child/parent, but also grandparent/grandchild relations. So if A is a parent of B, and B is a parent of C and C is a parent of D, we need to store the following relations:

node	ancestor
A	B
A	C
A	D
B	C
B	D
C	D

(in the real implementation integers are being used)

This information enables us to query the tree quickly without any form of recursion. To get the entire subtree of a node, you'd execute `SELECT ancestor FROM ancestors WHERE node='B'`. Likewise, to get all ancestors of a node, you'd execute `SELECT node FROM ancestors WHERE ancestor='D'`. In both queries you can simply JOIN the nodes table to retrieve the corresponding metadata. In the second query, you might notice that the output comes in no particular order, because there is no column to run SORT BY on. This is an implementation detail of libtree in order to save disk space and might change at a later point.

Manipulating the tree is somewhat more complex. When inserting a node, the ancestor information of its parent must be copied and completed. When deleting a node, all traces of it and its descendants must be deleted from both tables. When moving a node, first all outdated ancestor information must be found and deleted. Then the new parents ancestor information must be copied for the node (and its descendants) that is being moved and completed.

There are different ways to implement Closure Table. Some people store the depth of each ancestor/descendant combination to make sorting easier, some don't use the Adjacency List-style *parent* column, and some even save paths of *length zero* to reduce the complexity of some queries.

1.9.2 Indices

Everything has tradeoffs; libtree trades speed for disk space. This means its indices are huge. Both columns in the ancestor table are indexed separately and together, resulting in index sizes that are twice the size of the actual data. In the nodes table the columns `id` and `parent` are indexed, resulting in index sizes that are roughly the same as the data.

Maybe it's possible to remove indices, this needs benchmarking. But RAM and disk space became very cheap and don't really matter these days, right? ... right?

1.9.3 Database Triggers

The ancestor calculation happens automatically inside PostgreSQL using trigger functions written in PL/pgSQL. This is great because it means the user doesn't *have* to use libtree to modify the tree. They can use their own scripts or manually execute queries from CLI. It's possible to insert nodes, delete nodes or change the parent attribute of nodes - the integrity stays intact without the user having to do anything. On the other hand this means that altering the ancestor table will very likely result in a broken data set (don't do it).

1.9.4 Referential Integrity

While one advantage of using Closure Table is the possibility to use the RDBMSs referential integrity functionality, libtree doesn't use it in order to get more speed out of inserts and updates. If the integrity gets broken somehow, it's simple to fix:

- export nodes table using pgAdmin or similar
- delete both tables
- install libtree again
- import saved nodes table

1.9.5 Boundaries

The `id` column is of type `serial` (32bit integer) and can therefore be as high as 2,147,483,647. When needed, changing it to `bigserial` (64bit integer) is simple but requires more space.

1.9.6 Model Comparison

Closure Table

As mentioned before, Closure Table is a great database model to handle tree-like data. Its advantages are both read and write performance and also ease of implementation. It's recursion free and allows you to use referential integrity. The most complex and slowest part is when changing parents. Its disadvantage is high disk usage.

Adjacency List

The naive and most simple model. All queries and writes are very simple and fast. It also is referential integrity compatible. However, querying for nodes any deeper than the immediate children is near impossible without using recursion on the script side or the rather new WITH RECURSIVE statement.

Path Enumeration

A very good model if you don't mind stringly typed integrity and tremendous use of string functions in SQL queries. It should be fast for all types of queries but is not RI-compatible.

Nested Sets

Compared to the others, it's very complex and although popular, the worst model in all ways. It's simple to query subtrees, but it's hard and slow to do anything else. If you want to insert a node at the top, you must rebalance the entire tree. If you get the balancing wrong, you have no chance to repair the hierarchy. Furthermore it's not RI-compatible.

- genindex

Python Module Index

l

libtree.core.database, 9
libtree.core.node_data, 10
libtree.core.positioning, 10
libtree.core.properties, 12
libtree.core.query, 13
libtree.core.tree, 14
libtree.node, 8
libtree.transaction, 7
libtree.tree, 6

Index

Symbols

__eq__() (Node method), 8
__len__() (Node method), 8

A

ancestors (Node attribute), 8

C

change_parent() (in module libtree.core.tree), 14
children (Node attribute), 8
clear() (Transaction method), 7
close() (Tree method), 6
commit() (Transaction method), 7
create_schema() (in module libtree.core.database), 9
create_triggers() (in module libtree.core.database), 9

D

delete() (Node method), 8
delete_node() (in module libtree.core.tree), 14
descendants (Node attribute), 8
drop_tables() (in module libtree.core.database), 9

E

ensure_free_position() (in module libtree.core.positioning), 11

F

find_highest_position() (in module libtree.core.positioning), 11
flush_tables() (in module libtree.core.database), 9

G

get_ancestor_ids() (in module libtree.core.query), 13
get_ancestors() (in module libtree.core.query), 13
get_child_at_position() (Node method), 8
get_child_ids() (in module libtree.core.query), 13
get_children() (in module libtree.core.query), 13
get_children_count() (in module libtree.core.query), 13
get_descendant_ids() (in module libtree.core.query), 13
get_descendants() (in module libtree.core.query), 13
get_inherited_properties() (in module libtree.core.properties), 12
get_inherited_property_value() (in module libtree.core.properties), 12
get_node() (in module libtree.core.query), 13
get_node() (Transaction method), 7
get_node_at_position() (in module libtree.core.query), 14
get_nodes_by_property_dict() (in module libtree.core.properties), 12
get_nodes_by_property_dict() (Transaction method), 7
get_nodes_by_property_key() (in module libtree.core.properties), 12
get_nodes_by_property_key() (Transaction method), 7
get_nodes_by_property_value() (in module libtree.core.properties), 12
get_nodes_by_property_value() (Transaction method), 7
get_root_node() (in module libtree.core.query), 14
get_root_node() (Transaction method), 7
get_tree_size() (in module libtree.core.query), 14
get_tree_size() (Transaction method), 7

I

id (Node attribute), 8
id (NodeData attribute), 10
inherited_properties (Node attribute), 8
insert_child() (Node method), 8
insert_node() (in module libtree.core.tree), 14
insert_root_node() (Transaction method), 7
install() (Transaction method), 7
is_compatible_postgres_version() (in module libtree.core.database), 9
is_compatible_postgres_version() (Transaction method), 7

L

libtree.core.database (module), 9
libtree.core.node_data (module), 10
libtree.core.positioning (module), 10
libtree.core.properties (module), 12
libtree.core.query (module), 13
libtree.core.tree (module), 14

libtree.node (module), 8
libtree.transaction (module), 7
libtree.tree (module), 6

M

make_transaction() (Tree method), 6
move() (Node method), 9

N

Node (class in libtree.node), 8
node_data (Node attribute), 9
NodeData (class in libtree.core.node_data), 10

P

parent (Node attribute), 9
parent (NodeData attribute), 10
position (Node attribute), 9
position (NodeData attribute), 10
print_tree() (in module libtree.core.tree), 14
print_tree() (Transaction method), 7
properties (Node attribute), 9
properties (NodeData attribute), 10

R

rollback() (Transaction method), 7

S

set_position() (in module libtree.core.positioning), 11
set_position() (Node method), 9
set_properties() (in module libtree.core.properties), 12
set_properties() (Node method), 9
set_property_value() (in module libtree.core.properties), 12
shift_positions() (in module libtree.core.positioning), 11
swap_node_positions() (in module libtree.core.positioning), 12
swap_position() (Node method), 9

T

to_dict() (NodeData method), 10
Transaction (class in libtree.transaction), 7
Tree (class in libtree.tree), 6

U

uninstall() (Transaction method), 8
update_properties() (in module libtree.core.properties), 13