# psqlgraph Documentation

***Release 0.0.2b***

**Center for Data Intensive Science**

May 13, 2015

CONTENTS

CHAPTER

ONE

# PSQLGRAPH

Postgresql Graph Represetation Library

If you're just a detailed explanation of querying, feel free to skip to the section on *Graph Queries*.

If you're just a detailed explanation of how to write nodes, feel free to skip to the section on *Using The Session*.

## 1.1 Creating a model

**Note:** When creating models, all **Edge** classes must be imported **BEFORE** the **Node** classes. This allows the library to link the edges to the nodes at module load.

### 1.1.1 Nodes

Creating a node model is straightforward. We specify the properties of the node with the `pg_property` method. Each pg_property needs a 'setter'. The setter is where any custom validation happens:

```
from psqlgraph import Node, Edge, pg_property

class Foo(Node):

    # Optional: specify a custom label (defaults to
    # lowercase of class name, e.g. foo)
    __label__ = 'foo_node'

    # Optional: specify a non-null constraint key list
    __nonnull_properties__ = ['key1']

    @pg_property
    def key1(self, value):
        # insert custom validation here!
        self._set_property('key1', value)

    @pg_property(int)
    def key2(self, value):
        # Attempting to do node.key2 = '10' will
        # raise a ValidationError
        self._set_property('key1', value)

    @pg_property(enum=('allowed_value_1', 'allowed_2'))
    def key2(self, value):
        # Attempting to do node.key2 = 'not_allowed' will
```

```
        # raise a ValidationError
        self._set_property('key1', value)
```

You can also provide a list of keys that are non-nullable. This will be checked when the node is flushed to the database (basically whenever you commit a session or query the database).

### 1.1.2 Edges

Creating an edge model is similarly simple. Again note that edges must be declared **BEFORE** nodes, this example is out of order for clarity's sake:

```
class Edge1(Edge):
    __src_class__ = 'Foo'
    __dst_class__ = 'Bar'
    __src_dst_assoc__ = 'foos'
    __dst_src_assoc__ = 'bars'
```

The above edge would join two node classes Foo and Bar. Edges are direction. This allows a constitent source-to-destination relationship. The namnes of the source and destination classes are specified in `__src_class__` and `__dst_class__` respectively. When these node classes are instantiated, they will get a `foo._Edge1_out` and `bar._Edge1_in` attributes which are SQLAlchemy relationships specifying the connected edge objects. You are required to specify the `__src_dst_assoc__` and `__dst_src_assoc__` association attributes as well (though you can set them to None, if so, they will be ignored). These attributes specify what the AssociationProxy will be called. You will then be able to refer directly to the related objects, e.g. `foo.bars` will be a list of `Bar` objects related to your `Foo` object.

## 1.2 Using your models

Basic usage would be something like the following:

```
# IMPORTANT: You must import your models module.  This is how they are
# registered as models with psqlgraph.
from models import Test, Foo

g = PsqlGraphDriver(host, user, password, database)
with g.session_scope() as session:
    # create some nodes
    a = Test('a')
    b = Foo('b')

    # update our nodes
    a.key1 = 'Our first property!'
    a.foos.append(b)

    # b will now be a Foo in the list of a's Foos
    assert b in a.foos

    # actually add the nodes to the session
    session.add(a)
    session.merge(b)
# ^-- Here we exit the session scope  --^
#     anything you've added or merged
#     into the session will now be
#     flushed to the database.
```

```
# Now our changes above are stored in the database,
# let's look back at them with a new session.
with g.session_scope() as session:
    a = g.nodes(Test).props(key1='Our first property!').one()
    b = g.nodes(Foo).ids('b').one()
    assert b in a.foos

    excited = g.nodes(Test).filter(
        Test.key1.astext.endswith('Our first property!')).one()
    assert excited = a
```

Here we create nodes of types `Test, Foo` and add `b` to the list of `Foo` objects specified on the class `Test`. This is done behind the scenes with an AssociationProxy which represents a many-to-many relationship by creating a row in an edge table that specifies the source (`a`) and the destination (`b`). From a simplified perspect you can interact with association proxies like `Test.foos` as if they were python lists.

See sections *Using The Session* and *Querying* for more detailed usage.

### 1.2.1 Using The Session

**Note:** You cannot use psqlgraph without importing your models. In order to use any of your models, you must import them into the current runtime. For example, importing all classes from test/models.py will load the nodes and edges as subclasses.

Psqlgraph is basically a glorified session/model factory. It has a context scope function which is the bread and butter of dealing with the database. The session scope provides a SQLAlchemy session which is used as following (assuming we have classes `Foo, Bar, Baz` and edges `Edge1, Edge2` between `Foo, Bar` and `Bar, Baz` respectively:

```
g = PsqlGraphDriver(host, user, password, database)
with g.session_scope() as session:
     foo = Foo('1')
     bar = Bar('2')
     baz = Baz('3')
     edge1 = Edge1('1', '2')
     edge2 = Edge2('2', '3')
     session.add_all([foo, bar, edge1, edge2])
     ### We could call commit() here, but it will be done
     ### automatically as we exit the session scope context
     # session.commit()
```

As we exit the `with` clause, the session is automatically committed, which means that the changes are flushed to the database and written (at this point they also become visible to other sessions).

You can create a node by calling the model's constructor:

```
node1 = TestNode('id1')
```

You can update properties in one of many ways, all of them are equivalent:

```
node1.key1 = 'demonstration'
node1['key1'] = 'demonstration'
node1.properies['key1'] = 'demonstration'
```

Similarly, you can update the system annotations directly on the node. These will not be validated using any *hybrid_property* setter method that is specified as a `@pg_property`:

```
node.system_annotations['source'] = 'the moon'
```

When you are done updating your node, you can insert it or merge it into a session:

```
with g.session_scope() as session:
    session.insert(node1)
```

or:

```
with g.session_scope() as session:
    session.merge(node1)
```

the difference being that `insert()` will raise an exception if the node already exists in the database.

### 1.2.2 Querying

You can use the driver's `nodes()` function to produce a polymorphic query over your `Node` types. You can also specify a single node model to use: `.nodes(Foo)`. The following example follows the one above:

```
with g.session_scope() as session:
     # To get foo.bars to contain bar,
     # and bar.bazes to contian baz:
     foo = g.nodes().ids('1').one()
     bar = g.nodes(Bar).one()
     baz = g.nodes(Baz).ids('3').first()

     # To get all Foo nodes connected to Bar nodes,
     # connected to Baz nodes with the id '3':
     foo = g.nodes(Foo).path('bars.bazes').ids('3').all()
     assert foo.bars[0].node_id == bar.node_id
```

CHAPTER

# TWO

# PSQLGRAPHDRIVER

**class** psqlgraph.**PsqlGraphDriver**(*host*, *user*, *password*, *database*, ***kwargs*)

**nodes**(*query=<class 'psqlgraph.node.Node'>*)

**session_scope**(**args*, ***kwds*)

Provide a transactional scope around a series of operations.

This session scope has a deceptively complex behavior, so be careful when nesting sessions.

**Note:** A session scope that is not nested has the following properties:

1. Driver calls within the session scope will, by default, inherit the scope's session.
2. Explicitly passing a session as `session` will cause driver calls within the session scope to use the explicitly passed session.
3. Setting `can_inherit` to false will have no effect
4. Setting `must_inherit` to will raise a RuntimeError

**Note:** A session scope that is nested has the following properties given `driver` is a PsqlGraphDriver instance:

Example:

```
with driver.session_scope() as A:
    driver.node_insert()  # uses session A
    with driver.session_scope(A) as B:
        B == A  # is True
    with driver.session_scope() as C:
        C == A  # is True
    with driver.session_scope():
        driver.node_insert()  # uses session A still
    with driver.session_scope(can_inherit=False):
        driver.node_insert()  # uses new session D
    with driver.session_scope(can_inherit=False) as D:
        D != A  # is True
    with driver.session_scope() as E:
        E.rollback()  # rolls back session A
    with driver.session_scope(can_inherit=False) as F:
        F.rollback()  # does not roll back session A
    with driver.session_scope(can_inherit=False) as G:
```

```
G != A  # is True
driver.node_insert()  # uses session G
with driver.session_scope(A) as H:
    H == A  # true
    H != G  # true
    H.rollback()  # rolls back A but not G
with driver.session_scope(A):
    driver.node_insert()  # uses session A
```

**Parameters**

- **`session`** – The SQLAlchemy session to force the session scope to inherit
- **`can_inherit`** (*bool*) – The boolean value which determines whether the session scope inherits the session from any parent sessions in a nested context. The default behavior is to inherit the parent's session. If the session stack is empty for the driver, then this parameter is moot, there is no session to inherit, so one must be created.
- **`must_inherit`** (*bool*) – The boolean value which determines whether the session scope must inherit a session from a parent session. This parameter can be set to true to prevent session leaks from functions which return raw query objects

CHAPTER

THREE

# GRAPH QUERIES

**class** `psqlgraph.`**`GraphQuery`**(*entities*, *session=None*)

Query subclass implementing graph specific operations.

**`dst`**(*ids*)

Filter edges by dst_id

**Parameters** **`ids`** – A list of ids or single id to filter on Edge.dst_id == ids

**Returns** Returns a SQLAlchemy query object

```
g.nodes().dst('id1').filter(...
```

**`entity`**()

It is useful for us to be able to get the last entity in a chained join. Therfore, if there are _join_entities on the query, the entity will be the last one in the chain. If there is noo join in the query, then the entity is simply the specified entity.

**`has_sysan`**(*keys*)

Filter only entities that have a key *key* in system_annotations

**Parameters** **`key`** (*str*) – System annotation key

**`ids`**(*ids*)

Filter node by node_id

**Parameters** **`ids`** – A list of ids or single id to filter on Node.node_id == ids

**Returns** Returns a SQLAlchemy query object

```
g.nodes().ids('id1').filter(...
g.nodes().ids(['id1', 'id2']).filter(...
```

**`labels`**(*label*)

Filters on nodes that have certain labels.

**Parameters** **`labels`** – A single scalar string. The filtered results will have this label

**Returns** Returns a SQLAlchemy query object

**Note:** This is largely **deprecated**, rather you should specify the actual model class entity you want to return when you begin your query, e.g. driver.nodes(TestNode)

**`not_ids`**(*ids*)

Filter node such that returned nodes do not have node_id

**Parameters** **`ids`** – A list of ids or single id to filter on Node.node_id != ids

**Returns** Returns a SQLAlchemy query object

```
g.nodes().not_ids('id1').filter(...
g.nodes().not_ids('id1').filter(...
```

**not_props** (*props={}*, ***kwargs*)
Filter query results by property exclusion. See `props()` for usage.

**Parameters**

- **props** –

  ***Optional* A dictionary of properties which must not be a** subset of all result's properties.

- **kwargs** – This function also takes a list of key word arguments to include in the filter. This can be used in conjunction with *props*.

**Returns** Returns a SQLAlchemy query object

```
# Count the number of nodes with
# key1 != True and key2 != 'Yes'
g.props({'key1': True, 'key2': 'Yes'}).count()
g.props(key1=True, key2='Yes').count()
g.props({'key1': True}, key2='Yes').count()
```

**not_sysan** (*sysans={}*, ***kwargs*)
Filter query results by system_annotation exclusion. See `sysan()` for usage.

**Parameters**

- **sysans** –

  ***Optional* A dictionary of system_annotations which must not be a** subset of all result's system_annotations.

- **kwargs** – This function also takes a list of key word arguments to include in the filter. This can be used in conjunction with *props*.

**Returns** Returns a SQLAlchemy query object

**path** (**paths*)
Traverses a path in the graph given a list of AssociationProxy attributes

**Parameters paths** – Either list of paths or a string with paths each separated by '.'

**Returns** Returns a SQLAlchemy query object

```
# The following are identical and filter for Nations who
# have states who have cities who have streets named Main
# St.
g.nodes(Nation).path('states.cities.streets')\
               .props(name='Main St.')\
               .count()
g.nodes(Nation).path('states', 'cities', 'streets')\
               .props(name='Main St.')\
               .count()

# The following filters for nations that have states named
# Illinois and at least one city and at least one street
# named Main St.
g.nodes(Nation).path('states')\
               .props(name='Illinois')\
               .path('cities.streets')\
               .props(name='Main St.')\
```

```
                .count()

# The following filters on a forked path, i.e. a nation
# with a democratic government and also a state with a
# city with a street named Main St.
g.nodes(Nation).path('governments')\
                .props(type='Democracy')\
                .reset_joinpoin()\
                .path('states.cities.streets')\
                .props(name='Main St.')\
                .count()
```

**Note:** In order to fork a path, you can append .reset_joinpoint() to the returned query. This will set the join point back to the main entity, **but not** the filter point, i.e. any filter applied after .reset_joinpoint() will still attempt to filter on the last entity in .path(...). In order to filter at the beginning of the path, simply filter before path, i.e. query.filter().path() not query.path().reset_joinpoint().filter()

**path_via_assoc_proxy** (**entities*)

Similar to `path()`, but more cumbersome.

**Parameters** **entities** – A list of AssociationProxy entities to walk through.

**Returns** Returns a SQLAlchemy query object

```
# Filter for Nations who have states who have cities who
# have streets named Main St.
g.nodes(Nation).path_via_assoc_proxy(
                    Nation.states, States.cities, City.streets)\
                .props(name='Main St')\
                .count()
```

**prop** (*key*, *value*)

Filter query results by key value pair.

**Parameters**

- **key** (*str*) – Specifies which property to filter on.
- **value** – The value in property *key* must be equal to *value*

**Returns** Returns a SQLAlchemy query object

```
g.prop('key1', True).count()
```

**prop_in** (*key*, *values*)

Filter on entities that have a value corresponding to *key* that is in the list of keys *values*

**Parameters**

- **key** (*str*) – Specifies which property to filter on.
- **values** (*list*) – The value in property *key* must be in *list*

**Returns** Returns a SQLAlchemy query object

```
g.prop_in('key1', ['Yes', 'yes', 'True', 'true']).count()
```

**props** (*props={}*, ***kwargs*)

Filter query results by properties. Results in query will all contain given properties as a subset of _props.

**Parameters**

- **props** –

  *Optional* **A dictionary of properties which must be a** subset of all result's properties.
- **kwargs** – This function also takes a list of key word arguments to include in the filter. This can be used in conjunction with *props*.

**Returns** Returns a SQLAlchemy query object

```
# The following all count the number of nodes with
# key1 == True
# key2 == 'Yes'
g.props({'key1': True, 'key2': 'Yes'}).count()
g.props(key1=True, key2='Yes').count()
g.props({'key1': True}, key2='Yes').count()
```

**src**(*ids*)

Filter edges by src_id

**Parameters** **ids** – A list of ids or single id to filter on Edge.src_id == ids

**Returns** Returns a SQLAlchemy query object

```
g.nodes().src(node1.node_id).filter(...
```

**sysan**(*sysans={}*, ***kwargs*)

Filter query results by system_annotations. Results in query will all contain given properties as a subset of *system_annotations*.

**Parameters**

- **sysans** –

  *Optional* **A dictionary of annotations which must be a** subset of all result's system_annotations.
- **kwargs** – This function also takes a list of key word arguments to include in the filter. This can be used in conjunction with *sysans*.

**Returns** Returns a SQLAlchemy query object

```
g.sysan({'key1': True, 'key2': 'Yes'})
g.sysan(key1=True, key2='Yes')
g.sysan({'key1': True}, key2='Yes')
```

**with_edge_from_node**(*edge_type*, *source_node*)

Filter query to nodes with edges from a given node

**Parameters**

- **edge_type** – Edge model whose source is *target_node*
- **target_node** – The node that is a neighbor to other nodes through edge *edge_type*

**Returns** Returns a SQLAlchemy query object

```
g.nodes().with_edge_from_node(Edge1, node1).filter(...
```

**with_edge_to_node**(*edge_type*, *target_node*)

Filter query to nodes with edges to a given node

**Parameters**

- **edge_type** – Edge model whose destination is *target_node*

- **`target_node`** – The node that is a neighbor to other nodes through edge *edge_type*

**Returns** Returns a SQLAlchemy query object

```
g.nodes().with_edge_to_node(Edge1, node1).filter(...
```

CHAPTER

FOUR

# TESTS

```
$ python test/setup_test_psqlgraph.py
$ nosetest -v
```

CHAPTER

FIVE

# BUILDING DOCUMENTATION

```
$ python setup.py install # suggested to install using a virtualenv
$ cd doc
$ make latexpdf
```

CHAPTER

SIX

# UTILITY METHODS

psqlgraph.**retryable**(*func*)

This wrapper can be used to decorate a function to retry an operation in the case of an SQLalchemy IntegrityError. This error means that a race-condition has occured and operations that have occured within the session may no longer be valid.

You can set the number of retries by passing the keyword argument `max_retries` to the wrapped function. It's therefore important that `max_retries` is included as a kwarg in the definition of the wrapped function.

Setting `max_retries` to 0 will prevent retries upon failure; wrapped function will execute once.

Similar to `max_retries`, the kwarg `backoff` is a callback function that allows the user of the library to over-ride the default backoff function in the case of a retry. See *func default_backoff*

psqlgraph.**default_backoff**(*retries*, *max_retries*)

This is the default backoff function used in the case of a retry by and function wrapped with the `@retryable` decorator.

The behavior of the default backoff function is to sleep for a pseudo-random time between 0 and 2 seconds.

CHAPTER

SEVEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

## D



## E



## G



## H



## I



## L



## N



## P



## R



## S



## W