GeoAlchemy2 Documentation

Release 0.6.2

Eric Lemoine

Apr 29, 2019

Contents

Using SQLAlchemy with Spatial Databases.

GeoAlchemy 2 provides extensions to SQLAlchemy for working with spatial databases.

GeoAlchemy 2 focuses on PostGIS. PostGIS 1.5 and PostGIS 2 are supported.

SpatiaLite is also supported, but using GeoAlchemy 2 with SpatiaLite requires some specific configuration on the application side. GeoAlchemy 2 works with SpatiaLite 4.3.0 and higher.

GeoAlchemy 2 aims to be simpler than its predecessor, GeoAlchemy. Simpler to use, and simpler to maintain.

CHAPTER 1

Requirements

GeoAlchemy 2 requires SQLAlchemy 0.8. GeoAlchemy 2 does not work with SQLAlchemy 0.7 and lower.

CHAPTER 2

Installation

GeoAlchemy 2 is available on the Python Package Index. So it can be installed with the standard pip or easy_install tools.

CHAPTER 3

What's New in GeoAlchemy 2

- GeoAlchemy 2 supports PostGIS' `geometry` type, as well as the `geography` and `raster` types.
- The first series had its own namespace for spatial functions. With GeoAlchemy 2, spatial functions are called like any other SQLAlchemy function, using `func`, which is SQLAlchemy's standard way of calling SQL functions.
- GeoAlchemy 2 works with SQLAlchemy's ORM, as well as with SQLAlchemy's *SQL Expression Language* (a.k.a the SQLAlchemy Core). (This is thanks to SQLAlchemy's new type-level comparator system.)
- GeoAlchemy 2 supports reflection of geometry and geography columns.
- GeoAlchemy 2 adds `to_shape`, `from_shape` functions for a better integration with Shapely.

3.1 Migrate to GeoAlchemy 2

This section describes how to migrate an application from the first series of GeoAlchemy to GeoAlchemy 2.

3.1.1 Defining Geometry Columns

The first series has specific types like `Point`, `LineString` and `Polygon`. These are gone, the *geoalchemy2.types.Geometry* type should be used instead, and a `geometry_type` can be passed to it.

So, for example, a `polygon` column that used to be defined like this:

```
geom = Column(Polygon)
```

is now defined like this:

```
geom = Column(Geometry('POLYGON'))
```

This change is related to GeoAlchemy 2 supporting the *geoalchemy2.types.Geography* type.

3.1.2 Calling Spatial Functions

The first series has its own namespace/object for calling spatial functions, namely `geoalchemy.functions`. With GeoAlchemy 2, SQLAlchemy's `func` object should be used.

For example, the expression

```
functions.buffer(functions.centroid(box), 10, 2)
```

would be rewritten to this with GeoAlchemy 2:

```
func.ST_Buffer(func.ST_Centroid(box), 10, 2)
```

Also, as the previous example hinted it, the names of spatial functions are now all prefixed with `ST_`. (This is to be consistent with PostGIS and the `SQL-MM` standard.) The `ST_` prefix should be used even when applying spatial functions to columns, *geoalchemy2.elements.WKTElement*, or *geoalchemy2.elements.WKTElement* objects:

```
Lake.geom.ST_Buffer(10, 2)
lake_table.c.geom.ST_Buffer(10, 2)
lake.geom.ST_Buffer(10, 2)
```

3.1.3 WKB and WKT Elements

The first series has classes like `PersistentSpatialElement`, `PGPersistentSpatialElement`, `WKTSpatialElement`.

They're all gone, and replaced by two classes only: *geoalchemy2.elements.WKTElement* and *geoalchemy2.elements.WKBElement*.

geoalchemy2.elements.WKTElement is to be used in expressions where a geometry with a specific SRID should be specified. For example:

```
Lake.geom.ST_Touches(WKTElement('POINT(1 1)', srid=4326))
```

If no SRID need be specified, a string can used directly:

```
Lake.geom.ST_Touches('POINT(1 1)')
```

- *geoalchemy2.elements.WKTElement* literally replaces the first series' `WKTSpatialElement`.
- *geoalchemy2.elements.WKBElement* is the type into which GeoAlchemy 2 converts geometry values read from the database.

  For example, the `geom` attributes of `Lake` objects loaded from the database would be references to *geoalchemy2.elements.WKBElement* objects. This class replaces the first series' `PersistentSpatialElement` classes.

See the *Migrate to GeoAlchemy 2* page for details on how to migrate a GeoAlchemy application to GeoAlchemy 2.

CHAPTER 4

Tutorials

GeoAlchemy 2 works with both SQLAlchemy's *Object Relational Mapping* (ORM) and *SQL Expression Language*. This documentation provides a tutorial for each system. If you're new to GeoAlchemy 2 start with this.

4.1 ORM Tutorial

(This tutorial is greatly inspired by the SQLAlchemy ORM Tutorial, which is recommended reading, eventually.)

GeoAlchemy does not provide an Object Relational Mapper (ORM), but works well with the SQLAlchemy ORM. This tutorial shows how to use the SQLAlchemy ORM with spatial tables, using GeoAlchemy.

4.1.1 Connect to the DB

For this tutorial we will use a PostGIS 2 database. To connect we use SQLAlchemy's `create_engine()` function:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('postgresql://gis:gis@localhost/gis', echo=True)
```

In this example the name of the database, the database user, and the database password, is `gis`.

The `echo` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard logging module. With it is enabled, we'll see all the generated SQL produced.

The return value of `create_engine` is an `Engine` object, which represents the core interface to the database.

4.1.2 Declare a Mapping

When using the ORM, the configurational process starts by describing the database tables we'll be dealing with, and then by defining our own classes which will be mapped to those tables. In modern SQLAlchemy, these two tasks are usually performed together, using a system known as `Declarative`, which allows us to create classes that include directives to describe the actual database table they will be mapped to.

```
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy import Column, Integer, String
>>> from geoalchemy2 import Geometry
>>>
>>> Base = declarative_base()
>>>
>>> class Lake(Base):
...     __tablename__ = 'lake'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     geom = Column(Geometry('POLYGON'))
```

The `Lake` class establishes details about the table being mapped, including the name of the table denoted by `__tablename__`, and three columns `id`, `name`, and `geom`. The `id` column will be the primary key of the table. The `geom` column is a *`geoalchemy2.types.Geometry`* column whose `geometry_type` is `POLYGON`.

4.1.3 Create the Table in the Database

The `Lake` class has a corresponding `Table` object representing the database table. This `Table` object was created automatically by SQLAlchemy, it is referenced to by the `Lake.__table__` property:

```
>>> Lake.__table__
Table('lake', MetaData(bind=None), Column('id', Integer(), table=<lake>,
primary_key=True, nullable=False), Column('name', String(), table=<lake>),
Column('geom', Polygon(srid=4326), table=<lake>), schema=None)
```

To create the `lake` table in the database:

```
>>> Lake.__table__.create(engine)
```

If we wanted to drop the table we'd use:

```
>>> Lake.__table__.drop(engine)
```

4.1.4 Create an Instance of the Mapped Class

With the mapping declared, we can create a `Lake` object:

```
>>> lake = Lake(name='Majeur', geom='POLYGON((0 0,1 0,1 1,0 1,0 0))')
>>> lake.geom
'POLYGON((0 0,1 0,1 1,0 1,0 0))'
>>> str(lake.id)
'None'
```

A WKT is passed to the `Lake` constructor for its geometry. This WKT represents the shape of our lake. Since we have not yet told SQLAlchemy to persist the `lake` object, its `id` is `None`.

The EWKT (Extended WKT) format is also supported. So, for example, if the spatial reference system for the geometry column were `4326`, the string `SRID=4326;POLYGON((0 0,1 0,1,0 1,0 0))` could be used as the geometry representation.

4.1.5 Create a Session

The ORM interacts with the database through a `Session`. Let's create a `Session` class:

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=engine)
```

This custom-made `Session` class will create new `Session` objects which are bound to our database. Then, whenever we need to have a conversation with the database, we instantiate a `Session`:

```
>>> session = Session()
```

The above `Session` is associated with our PostgreSQL `Engine`, but it hasn't opened any connection yet.

4.1.6 Add New Objects

To persist our `Lake` object, we `add()` it to the `Session`:

```
>>> session.add(lake)
```

At this point the `lake` object has been added to the `Session`, but no SQL has been issued to the database. The object is in a *pending* state. To persist the object a *flush* or *commit* operation must occur (commit implies flush):

```
>>> session.commit()
```

We can now query the database for `Majeur`:

```
>>> our_lake = session.query(Lake).filter_by(name='Majeur').first()
>>> our_lake.name
u'Majeur'
>>> our_lake.geom
<WKBElement at 0x9af594c;
↪'0103000000010000000500000000000000000000000000000000000000000000000000000000000000000000f03f000000000000000000000000000000
↪'>
>>> our_lake.id
1
```

`our_lake.geom` is a *geoalchemy2.elements.WKBElement*, which a type provided by GeoAlchemy. *geoalchemy2.elements.WKBElement* wraps a WKB value returned by the database.

Let's add more lakes:

```
>>> session.add_all([
...     Lake(name='Garde', geom='POLYGON((1 0,3 0,3 2,1 2,1 0))'),
...     Lake(name='Orta', geom='POLYGON((3 0,6 0,6 3,3 3,3 0))')
... ])
>>> session.commit()
```

4.1.7 Query

A `Query` object is created using the `query()` function on `Session`. For example here's a `Query` that loads `Lake` instances ordered by their names:

```
>>> query = session.query(Lake).order_by(Lake.name)
```

Any `Query` is iterable:

```
>>> for lake in query:
...     print lake.name
...
Garde
Majeur
Orta
```

Another way to execute the query and get a list of `Lake` objects involves calling `all()` on the `Query`:

```
>>> lakes = session.query(Lake).order_by(Lake.name).all()
```

The SQLAlchemy ORM Tutorial's Querying section provides more examples of queries.

4.1.8 Make Spatial Queries

Using spatial filters in SQL SELECT queries is very common. Such queries are performed by using spatial relationship functions, or operators, in the `WHERE` clause of the SQL query.

For example, to find the `Lake` s that contain the point `POINT(4 1)`, we can use this `Query`:

```
>>> from sqlalchemy import func
>>> query = session.query(Lake).filter(
...             func.ST_Contains(Lake.geom, 'POINT(4 1)'))
...
>>> for lake in query:
...     print lake.name
...
Orta
```

GeoAlchemy allows rewriting this `Query` more concisely:

```
>>> query = session.query(Lake).filter(Lake.geom.ST_Contains('POINT(4 1)'))
>>> for lake in query:
...     print lake.name
...
Orta
```

Here the `ST_Contains` function is applied to the `Lake.geom` column property. In that case the column property is actually passed to the function, as its first argument.

Here's another spatial filtering query, based on `ST_Intersects`:

```
>>> query = session.query(Lake).filter(
...             Lake.geom.ST_Intersects('LINESTRING(2 1,4 1)'))
...
>>> for lake in query:
...     print lake.name
...
Garde
Orta
```

We can also apply relationship functions to *`geoalchemy2.elements.WKBElement`*. For example:

```
>>> lake = session.query(Lake).filter_by(name='Garde').one()
>>> print session.scalar(lake.geom.ST_Intersects('LINESTRING(2 1,4 1)'))
True
```

`session.scalar` allows executing a clause and returning a scalar value (a boolean value in this case).

The GeoAlchemy functions all start with `ST_`. Operators are also called as functions, but the function names don't include the `ST_` prefix. As an example let's use PostGIS' `&&` operator, which allows testing whether the bounding boxes of geometries intersect. GeoAlchemy provides the `intersects` function for that:

```
>>> query = session.query
>>> query = session.query(Lake).filter(
...             Lake.geom.intersects('LINESTRING(2 1,4 1)'))
...
>>> for lake in query:
...     print lake.name
...
Garde
Orta
```

4.1.9 Set Spatial Relationships in the Model

Let's assume that in addition to `lake` we have another table, `treasure`, that includes treasure locations. And let's say that we are interested in discovering the treasures hidden at the bottom of lakes.

The `Treasure` class is the following:

```
>>> class Treasure(Base):
...      __tablename__ = 'treasure'
...      id = Column(Integer, primary_key=True)
...      geom = Column(Geometry('POINT'))
```

We can now add a `relationship` to the `Lake` table to automatically load the treasures contained by each lake:

```
>>> from sqlalchemy.orm import relationship, backref
>>> class Lake(Base):
...     __tablename__ = 'lake'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     geom = Column(Geometry('POLYGON'))
...     treasures = relationship(
...         'Treasure',
...         primaryjoin='func.ST_Contains(foreign(Lake.geom), Treasure.geom).as_
↪comparison(1, 2)',
...         backref=backref('lake', uselist=False),
...         viewonly=True,
...         uselist=True,
...     )
```

Note the use of the `as_comparison` function. It is required for using an SQL function (`ST_Contains` here) in a `primaryjoin` condition. This only works with SQLAlchemy 1.3, as the `as_comparison` function did not exist before that version. See the Custom operators based on SQL function section of the SQLAlchemy documentation for more information.

Some information on the parameters used for configuring this `relationship`:

- `backref` is used to provide the name of property to be placed on the class that handles this relationship in the other direction, namely `Treasure`;
- `viewonly=True` specifies that the relationship is used only for loading objects, and not for persistence operations;

- `uselist=True` indicates that the property should be loaded as a list, as opposed to a scalar.

Also, note that the `treasures` property on `lake` objects (and the `lake` property on `treasure` objects) is loaded "lazily" when the property is first accessed. Another loading strategy may be configured in the `relationship`. For example you'd use `lazy='joined'` for related items to be loaded "eagerly" in the same query as that of the parent, using a `JOIN` or `LEFT OUTER JOIN`.

See the Relationships API section of the SQLAlchemy documention for more detail on the `relationship` function, and all the parameters that can be used to configure it.

4.1.10 Use Other Spatial Functions

Here's a `Query` that calculates the areas of buffers for our lakes:

```
>>> from sqlalchemy import func
>>> query = session.query(Lake.name,
...                       func.ST_Area(func.ST_Buffer(Lake.geom, 2)) \
...                           .label('bufferarea'))
>>> for row in query:
...     print '%s: %f' % (row.name, row.bufferarea)
...
Majeur: 21.485781
Garde: 32.485781
Orta: 45.485781
```

This `Query` applies the PostGIS `ST_Buffer` function to the geometry column of every row of the `lake` table. The return value is a list of rows, where each row is actually a tuple of two values: the lake name, and the area of a buffer of the lake. Each tuple is actually an SQLAlchemy `KeyedTuple` object, which provides property type accessors.

Again, the `Query` can written more concisely:

```
>>> query = session.query(Lake.name,
...                       Lake.geom.ST_Buffer(2).ST_Area().label('bufferarea'))
>>> for row in query:
...     print '%s: %f' % (row.name, row.bufferarea)
...
Majeur: 21.485781
Garde: 32.485781
Orta: 45.485781
```

Obviously, processing and measurement functions can also be used in `WHERE` clauses. For example:

```
>>> lake = session.query(Lake).filter(
...             Lake.geom.ST_Buffer(2).ST_Area() > 33).one()
...
>>> print lake.name
Orta
```

And, like any other functions supported by GeoAlchemy, processing and measurement functions can be applied to *`geoalchemy2.elements.WKBElement`*. For example:

```
>>> lake = session.query(Lake).filter_by(name='Majeur').one()
>>> bufferarea = session.scalar(lake.geom.ST_Buffer(2).ST_Area())
>>> print '%s: %f' % (lake.name, bufferarea)
Majeur: 21.485781
```

4.1.11 Further Reference

- Spatial Functions Reference: *Spatial Functions*
- Spatial Operators Reference: *Spatial Operators*
- Elements Reference: *Elements*

4.2 Core Tutorial

(This tutorial is greatly inspired from the SQLAlchemy SQL Expression Language Tutorial, which is recommended reading, eventually.)

This tutorial shows how to use the SQLAlchemy Expression Language (a.k.a. SQLAlchemy Core) with GeoAlchemy. As defined by the SQLAlchemy documentation itself, in contrast to the ORM's domain-centric mode of usage, the SQL Expression Language provides a schema-centric usage paradigm.

4.2.1 Connect to the DB

For this tutorial we will use a PostGIS 2 database. To connect we use SQLAlchemy's `create_engine()` function:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('postgresql://gis:gis@localhost/gis', echo=True)
```

In this example the name of the database, the database user, and the database password, is `gis`.

The `echo` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard logging module. With it is enabled, we'll see all the generated SQL produced.

The return value of `create_engine` is an `Engine` object, which respresents the core interface to the database.

4.2.2 Define a Table

The very first object that we need to create is a `Table`. Here we create a `lake_table` object, which will correspond to the `lake` table in the database:

```
>>> from sqlalchemy import Table, Column, Integer, String, MetaData
>>> from geoalchemy2 import Geometry
>>>
>>> metadata = MetaData()
>>> lake_table = Table('lake', metadata,
...     Column('id', Integer, primary_key=True),
...     Column('name', String),
...     Column('geom', Geometry('POLYGON'))
... )
```

This table is composed of three columns, `id`, `name` and `geom`. The `geom` column is a *`geoalchemy2.types.Geometry`* column whose `geometry_type` is `POLYGON`.

Any `Table` object is added to a `MetaData` object, which is a catalog of `Table` objects (and other related objects).

4.2.3 Create the Table

With our `Table` being defined we're ready (to have SQLAlchemy) create it in the database:

```
>>> lake_table.create(engine)
```

Calling `create_all()` on `metadata` would have worked equally well:

```
>>> metadata.create_all(engine)
```

In that case every `Table` that's referenced to by `metadata` would be created in the database. The `metadata` object includes one `Table` here, our now well-known `lake_table` object.

4.2.4 Insertions

We want to insert records into the `lake` table. For that we need to create an `Insert` object. SQLAlchemy provides multiple constructs for creating an `Insert` object, here's one:

```
>>> ins = lake_table.insert()
>>> str(ins)
INSERT INTO lake (id, name, geom) VALUES (:id, :name, ST_GeomFromEWKT(:geom))
```

The `geom` column being a `Geometry` column, the `:geom` bind value is wrapped in a `ST_GeomFromEWKT` call.

To limit the columns named in the `INSERT` query the `values()` method can be used:

```
>>> ins = lake_table.insert().values(name='Majeur',
...                                  geom='POLYGON((0 0,1 0,1 1,0 1,0 0))')
...
>>> str(ins)
INSERT INTO lake (name, geom) VALUES (:name, ST_GeomFromEWKT(:geom))
```

Tip: The string representation of the SQL expression does not include the data placed in `values`. We got named bind parameters instead. To view the data we can get a compiled form of the expression, and ask for its `params`:

```
>>> ins.compile.params()
{'geom': 'POLYGON((0 0,1 0,1 1,0 1,0 0))', 'name': 'Majeur'}
```

Up to now we've created an `INSERT` query but we haven't sent this query to the database yet. Before being able to send it to the database we need a database `Connection`. We can get a `Connection` from the `Engine` object we created earlier:

```
>>> conn = engine.connect()
```

We're now ready to execute our `INSERT` statement:

```
>>> result = conn.execute(ins)
```

This is what the logging system should output:

```
INSERT INTO lake (name, geom) VALUES (%(name)s, ST_GeomFromEWKT(%(geom)s)) RETURNING␣
↪lake.id
{'geom': 'POLYGON((0 0,1 0,1 1,0 1,0 0))', 'name': 'Majeur'}
COMMIT
```

The value returned by `conn.execute()`, stored in `result`, is a `sqlalchemy.engine.ResultProxy` object. In the case of an `INSERT` we can get the primary key value which was generated from our statement:

```
>>> result.inserted_primary_key
[1]
```

Instead of using `values()` to specify our `INSERT` data, we can send the data to the `execute()` method on `Connection`. So we could rewrite things as follows:

```
>>> conn.execute(lake_table.insert(),
...              name='Majeur', geom='POLYGON((0 0,1 0,1 1,0 1,0 0))')
```

Now let's use another form, allowing to insert multiple rows at once:

```
>>> conn.execute(lake_table.insert(), [
...     {'name': 'Garde', 'geom': 'POLYGON((1 0,3 0,3 2,1 2,1 0))'},
...     {'name': 'Orta', 'geom': 'POLYGON((3 0,6 0,6 3,3 3,3 0))'}
...     ])
...
```

Tip: In the above examples the geometries are specified as WKT strings. Specifying them as EWKT strings is also supported.

4.2.5 Selections

Inserting involved creating an `Insert` object, so it'd come to no surprise that Selecting involves creating a `Select` object. The primary construct to generate `SELECT` statements is SQLAlchemy's `select()` function:

```
>>> from sqlalchemy.sql import select
>>> s = select([lake_table])
>>> str(s)
SELECT lake.id, lake.name, ST_AsEWKB(lake.geom) AS geom FROM lake
```

The `geom` column being a `Geometry` it is wrapped in a `ST_AsEWKB` call when specified as a column in a `SELECT` statement.

We can now execute the statement and look at the results:

```
>>> result = conn.execute(s)
>>> for row in result:
...     print 'name:', row['name'], '; geom:', row['geom'].desc
...
name: Majeur ; geom: 0103...
name: Garde ; geom: 0103...
name: Orta ; geom: 0103...
```

`row['geom']` is a `geoalchemy2.types.WKBElement` instance. In this example we just get an hexadecimal representation of the geometry's WKB value using the `desc` property.

4.2.6 Spatial Query

As spatial database users executing spatial queries is of a great interest to us. There comes GeoAlchemy!

Spatial relationship

Using spatial filters in SQL SELECT queries is very common. Such queries are performed by using spatial relationship functions, or operators, in the `WHERE` clause of the SQL query.

For example, to find lakes that contain the point `POINT(4 1)`, we can use this:

```
>>> from sqlalchemy import func
>>> s = select([lake_table],
               func.ST_Contains(lake_table.c.geom, 'POINT(4 1)'))
>>> str(s)
SELECT lake.id, lake.name, ST_AsEWKB(lake.geom) AS geom FROM lake WHERE ST_
↪Contains(lake.geom, :param_1)
>>> result = conn.execute(s)
>>> for row in result:
...     print 'name:', row['name'], '; geom:', row['geom'].desc
...
name: Orta ; geom: 0103...
```

GeoAlchemy allows rewriting this more concisely:

```
>>> s = select([lake_table], lake_table.c.geom.ST_Contains('POINT(4 1)'))
>>> str(s)
SELECT lake.id, lake.name, ST_AsEWKB(lake.geom) AS geom FROM lake WHERE ST_
↪Contains(lake.geom, :param_1)
```

Here the `ST_Contains` function is applied to `lake.c.geom`. And the generated SQL the `lake.geom` column is actually passed to the `ST_Contains` function as the first argument.

Here's another spatial query, based on `ST_Intersects`:

```
   >>> s = select([lake_table],
   ...            lake_table.c.geom.ST_Intersects('LINESTRING(2 1,4 1)'))
   >>> result = conn.execute(s)
   >>> for row in result:
   ...     print 'name:', row['name'], '; geom:', row['geom'].desc
   ...
   name: Garde ; geom: 0103...
   name: Orta ; geom: 0103...

This query selects lakes whose geometries intersect ``LINESTRING(2 1,4 1)``.
```

The GeoAlchemy functions all start with `ST_`. Operators are also called as functions, but the names of operator functions don't include the `ST_` prefix.

As an example let's use PostGIS' `&&` operator, which allows testing whether the bounding boxes of geometries intersect. GeoAlchemy provides the `intersects` function for that:

```
>>> s = select([lake_table],
...            lake_table.c.geom.intersects('LINESTRING(2 1,4 1)'))
>>> result = conn.execute(s)
>>> for row in result:
...     print 'name:', row['name'], '; geom:', row['geom'].desc
...
name: Garde ; geom: 0103...
name: Orta ; geom: 0103...
```

Processing and Measurement

Here's a `Select` that calculates the areas of buffers for our lakes:

```
>>> s = select([lake_table.c.name,
                func.ST_Area(
                    lake_table.c.geom.ST_Buffer(2)).label('bufferarea')])
>>> str(s)
SELECT lake.name, ST_Area(ST_Buffer(lake.geom, %(param_1)s)) AS bufferarea FROM lake
>>> result = conn.execute(s)
>>> for row in result:
...     print '%s: %f' % (row['name'], row['bufferarea'])
Majeur: 21.485781
Garde: 32.485781
Orta: 45.485781
```

Obviously, processing and measurement functions can also be used in `WHERE` clauses. For example:

```
>>> s = select([lake_table.c.name],
               lake_table.c.geom.ST_Buffer(2).ST_Area() > 33)
>>> str(s)
SELECT lake.name FROM lake WHERE ST_Area(ST_Buffer(lake.geom, :param_1)) > :ST_Area_1
>>> result = conn.execute(s)
>>> for row in result:
...     print row['name']
Orta
```

And, like any other functions supported by GeoAlchemy, processing and measurement functions can be applied to *`geoalchemy2.elements.WKBElement`*. For example:

```
>>> s = select([lake_table], lake_table.c.name == 'Majeur')
>>> result = conn.execute(s)
>>> lake = result.fetchone()
>>> bufferarea = conn.scalar(lake[lake_table.c.geom].ST_Buffer(2).ST_Area())
>>> print '%s: %f' % (lake['name'], bufferarea)
Majeur: 21.485781
```

4.2.7 Further Reference

- Spatial Functions Reference: *Spatial Functions*
- Spatial Operators Reference: *Spatial Operators*
- Elements Reference: *Elements*

4.3 SpatiaLite Tutorial

GeoAlchemy 2's main target is PostGIS. But GeoAlchemy 2 also supports SpatiaLite, the spatial extension to SQLite. This tutorial describes how to use GeoAlchemy 2 with SpatiaLite. It's based on the *ORM Tutorial*, which you may want to read first.

4.3.1 Connect to the DB

Just like when using PostGIS connecting to a SpatiaLite database requires an `Engine`. This is how you create one for SpatiaLite:

```
>>> from sqlalchemy import create_engine
>>> from sqlalchemy.event import listen
>>>
>>> def load_spatialite(dbapi_conn, connection_record):
...     dbapi_conn.enable_load_extension(True)
...     dbapi_conn.load_extension('/usr/lib/x86_64-linux-gnu/mod_spatialite.so')
...
>>>
>>> engine = create_engine('sqlite:///gis.db', echo=True)
>>> listen(engine, 'connect', load_spatialite)
```

The call to `create_engine` creates an engine bound to the database file `gis.db`. After that a `connect` listener is registered on the engine. The listener is responsible for loading the SpatiaLite extension, which is a necessary operation for using SpatiaLite through SQL.

At this point you can test that you are able to connect to the database:

```
>> conn = engine.connect()
2018-05-30 17:12:02,675 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain␣
↪returns' AS VARCHAR(60)) AS anon_1
2018-05-30 17:12:02,676 INFO sqlalchemy.engine.base.Engine ()
2018-05-30 17:12:02,676 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode␣
↪returns' AS VARCHAR(60)) AS anon_1
2018-05-30 17:12:02,676 INFO sqlalchemy.engine.base.Engine ()
```

You can also check that the `gis.db` SQLite database file was created on the file system.

One additional step is required for using SpatiaLite: create the `geometry_columns` and `spatial_ref_sys` metadata tables. This is done by calling SpatiaLite's `InitSpatialMetaData` function:

```
>>> from sqlalchemy.sql import select, func
>>>
>>> conn.execute(select([func.InitSpatialMetaData()]))
```

Note that this operation may take some time the first time it is executed for a database. When `InitSpatialMetaData` is executed again it will report an error:

```
InitSpatiaMetaData() error:"table spatial_ref_sys already exists"
```

You can safely ignore that error.

Before going further we can close the current connection:

```
>>> conn.close()
```

4.3.2 Declare a Mapping

Now that we have a working connection we can go ahead and create a mapping between a Python class and a database table.

```
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy import Column, Integer, String
>>> from geoalchemy2 import Geometry
>>>
>>> Base = declarative_base()
>>>
>>> class Lake(Base):
...     __tablename__ = 'lake'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     geom = Column(Geometry(geometry_type='POLYGON', management=True))
```

This basically works in the way as with PostGIS. The difference is the `management` argument that must be set to `True`.

Setting `management` to `True` indicates that the `AddGeometryColumn` and `DiscardGeometryColumn` management functions will be used for the creation and removal of the geometry column. This is required with SpatiaLite.

4.3.3 Create the Table in the Database

We can now create the `lake` table in the `gis.db` database:

```
>>> Lake.__table__.create(engine)
```

If we wanted to drop the table we'd use:

```
>>> Lake.__table__.drop(engine)
```

There's nothing specific to SpatiaLite here.

4.3.4 Create a Session

When using the SQLAlchemy ORM the ORM interacts with the database through a `Session`.

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=engine)
>>> session = Session()
```

The session is associated with our SpatiaLite `Engine`. Again, there's nothing specific to SpatiaLite here.

4.3.5 Add New Objects

We can now create and insert new `Lake` objects into the database, the same way we'd do it using GeoAlchemy 2 with PostGIS.

```
>>> lake = Lake(name='Majeur', geom='POLYGON((0 0,1 0,1 1,0 1,0 0))')
>>> session.add(lake)
>>> session.commit()
```

We can now query the database for `Majeur`:

```
>>> our_lake = session.query(Lake).filter_by(name='Majeur').first()
>>> our_lake.name
u'Majeur'
>>> our_lake.geom
<WKBElement at 0x9af594c;
↪'0103000000010000000500000000000000000000000000000000000000000000000000000000000000000f03f000000000000000000000000000000
↪'>
>>> our_lake.id
1
```

Let's add more lakes:

```
>>> session.add_all([
...     Lake(name='Garde', geom='POLYGON((1 0,3 0,3 2,1 2,1 0))'),
...     Lake(name='Orta', geom='POLYGON((3 0,6 0,6 3,3 3,3 0))')
... ])
>>> session.commit()
```

4.3.6 Query

Let's make a simple, non-spatial, query:

```
>>> query = session.query(Lake).order_by(Lake.name)
>>> for lake in query:
...     print(lake.name)
...
Garde
Majeur
Orta
```

Now a spatial query:

```
>>> from geolachemy2 import WKTElement
>>> query = session.query(Lake).filter(
...             func.ST_Contains(Lake.geom, WKTElement('POINT(4 1)')))
...
>>> for lake in query:
...     print(lake.name)
...
Orta
```

Here's another spatial query, using `ST_Intersects` this time:

```
>>> query = session.query(Lake).filter(
...             Lake.geom.ST_Intersects(WKTElement('LINESTRING(2 1,4 1)')))
...
>>> for lake in query:
...     print(lake.name)
...
Garde
Orta
```

We can also apply relationship functions to *geoalchemy2.elements.WKBElement*. For example:

```
>>> lake = session.query(Lake).filter_by(name='Garde').one()
>>> print(session.scalar(lake.geom.ST_Intersects(WKTElement('LINESTRING(2 1,4 1)'))))
1
```

`session.scalar` allows executing a clause and returning a scalar value (an integer value in this case).

The value `1` indicates that the lake "Garde" does intersects the `LINESTRING(2 1,4 1)` geometry. See the SpatiaLite SQL functions reference list for more information.

4.3.7 Further Reference

- GeoAlchemy 2 ORM Tutotial: *ORM Tutorial*
- GeoAlchemy 2 Spatial Functions Reference: *Spatial Functions*
- GeoAlchemy 2 Spatial Operators Reference: *Spatial Operators*
- GeoAlchemy 2 Elements Reference: *Elements*
- SpatiaLite 4.3.0 SQL functions reference list

CHAPTER 5

Reference Documentation

5.1 Types

This module defines the *geoalchemy2.types.Geometry*, *geoalchemy2.types.Geography*, and *geoalchemy2.types.Raster* classes, that are used when defining geometry, geography and raster columns/properties in models.

5.1.1 Reference

class geoalchemy2.types.**CompositeType**
Bases: sqlalchemy.sql.type_api.UserDefinedType

A wrapper for geoalchemy2.elements.CompositeElement, that can be used as the return type in PostgreSQL functions that return composite values.

This is used as the base class of *geoalchemy2.types.GeometryDump*.

class comparator_factory(*expr*)
Bases: sqlalchemy.sql.type_api.Comparator

typemap = {}
Dictionary used for defining the content types and their corresponding keys. Set in subclasses.

class geoalchemy2.types.**Geography**(*geometry_type='GEOMETRY'*, *srid=-1*, *dimension=2*, *spatial_index=True*, *management=False*, *use_typmod=None*)
Bases: *geoalchemy2.types._GISType*

The Geography type.

Creating a geography column is done like this:

```
Column(Geography(geometry_type='POINT', srid=4326))
```

See *geoalchemy2.types._GISType* for the list of arguments that can be passed to the constructor.

as_binary = 'ST_AsBinary'
The "as binary" function to use. Used by the parent class' `column_expression` method.

from_text = 'ST_GeogFromText'
The `FromText` geography constructor. Used by the parent class' `bind_expression` method.

name = 'geography'
Type name used for defining geography columns in `CREATE TABLE`.

class geoalchemy2.types.**Geometry**(*geometry_type='GEOMETRY'*, *srid=-1*, *dimension=2*, *spatial_index=True*, *management=False*, *use_typmod=None*)
Bases: *geoalchemy2.types._GISType*

The Geometry type.

Creating a geometry column is done like this:

```
Column(Geometry(geometry_type='POINT', srid=4326))
```

See *geoalchemy2.types._GISType* for the list of arguments that can be passed to the constructor.

If `srid` is set then the `WKBElement` objects resulting from queries will have that SRID, and, when constructing the ``WKBElement objects, the SRID won't be read from the data returned by the database. If `srid` is not set (meaning it's `-1`) then the SRID set in "WKBElement" objects will be read from the data returned by the database.

as_binary = 'ST_AsEWKB'
The "as binary" function to use. Used by the parent class' `column_expression` method.

from_text = 'ST_GeomFromEWKT'
The "from text" geometry constructor. Used by the parent class' `bind_expression` method.

name = 'geometry'
Type name used for defining geometry columns in `CREATE TABLE`.

class geoalchemy2.types.**GeometryDump**
Bases: *geoalchemy2.types.CompositeType*

The return type for functions like `ST_Dump`, consisting of a path and a geom field. You should normally never use this class directly.

typemap = {'geom': <class 'geoalchemy2.types.Geometry'>, 'path': ARRAY(Integer())}
Dictionary defining the contents of a `geometry_dump`.

class geoalchemy2.types.**Raster**(*spatial_index=True*)
Bases: `sqlalchemy.sql.type_api.UserDefinedType`

The Raster column type.

Creating a raster column is done like this:

```
Column(Raster)
```

This class defines the `result_processor` method, so that raster values received from the database are converted to *geoalchemy2.elements.RasterElement* objects.

Constructor arguments:

`spatial_index`

Indicate if a spatial index should be created. Default is `True`.

comparator_factory
alias of *geoalchemy2.comparator.BaseComparator*

result_processor(*dialect*, *coltype*)
Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect** – Dialect instance in use.
- **coltype** – DBAPI coltype argument received in cursor.description.

class geoalchemy2.types.**_GISType**(*geometry_type='GEOMETRY'*, *srid=-1*, *dimension=2*, *spatial_index=True*, *management=False*, *use_typmod=None*)
Bases: `sqlalchemy.sql.type_api.UserDefinedType`

The base class for *geoalchemy2.types.Geometry* and *geoalchemy2.types.Geography*.

This class defines `bind_expression` and `column_expression` methods that wrap column expressions in `ST_GeomFromEWKT`, `ST_GeogFromText`, or `ST_AsEWKB` calls.

This class also defines `result_processor` and `bind_processor` methods. The function returned by `result_processor` converts WKB values received from the database to *geoalchemy2.elements.WKBElement* objects. The function returned by `bind_processor` converts *geoalchemy2.elements.WKTElement* objects to EWKT strings.

Constructor arguments:

`geometry_type`

The geometry type.

Possible values are:

- `"GEOMETRY"`,
- `"POINT"`,
- `"LINESTRING"`,
- `"POLYGON"`,
- `"MULTIPOINT"`,
- `"MULTILINESTRING"`,
- `"MULTIPOLYGON"`,
- `"GEOMETRYCOLLECTION"`
- `"CURVE"`,
- `None`.

The latter is actually not supported with *geoalchemy2.types.Geography*.

When set to `None` then no "geometry type" constraints will be attached to the geometry type declaration. Using `None` here is not compatible with setting `management` to `True`.

Default is `"GEOMETRY"`.

`srid`

The SRID for this column. E.g. 4326. Default is `-1`.

`dimension`

The dimension of the geometry. Default is `2`.

When set to `3` then the "geometry_type" is constrained to terminate on either `"Z"` or `"M"`. When set to `4` then the "geometry_type" must terminate with `"ZM"`.

`spatial_index`

Indicate if a spatial index should be created. Default is `True`.

`management`

Indicate if the `AddGeometryColumn` and `DropGeometryColumn` managements functions should be called when adding and dropping the geometry column. Should be set to `True` for PostGIS 1.x. Default is `False`. Note that this option has no effect for *geoalchemy2.types.Geography*.

`use_typmod`

By default PostgreSQL type modifiers are used to create the geometry column. To use check constraints instead set `use_typmod` to `False`. By default this option is not included in the call to `AddGeometryColumn`. Note that this option is only taken into account if `management` is set to `True` and is only available for PostGIS 2.x.

as_binary = None
The name of the "as binary" function for this type. Set in subclasses.

bind_expression(*bindvalue*)
"Given a bind value (i.e. a `BindParameter` instance), return a SQL expression in its place.

This is typically a SQL function that wraps the existing bound parameter within the statement. It is used for special data types that require literals being wrapped in some special database function in order to coerce an application-level value into a database-specific format. It is the SQL analogue of the `TypeEngine.bind_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

Note that this method, when implemented, should always return the exact same structure, without any conditional logic, as it may be used in an executemany() call against an arbitrary number of bound parameter sets.

See also:

types_sql_value_processing

bind_processor(*dialect*)
Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

Parameters **dialect** – Dialect instance in use.

column_expression(*col*)
Given a SELECT column expression, return a wrapping SQL expression.

This is typically a SQL function that wraps a column expression as rendered in the columns clause of a SELECT statement. It is used for special data types that require columns to be wrapped in some special database function in order to coerce the value before being sent back to the application. It is the SQL analogue of the `TypeEngine.result_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

See also:

types_sql_value_processing

comparator_factory
alias of *geoalchemy2.comparator.Comparator*

from_text = None
The name of "from text" function for this type. Set in subclasses.

name = None
Name used for defining the main geo type (geometry or geography) in CREATE TABLE statements. Set in subclasses.

result_processor(*dialect*, *coltype*)
Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **dialect** – Dialect instance in use.
- **coltype** – DBAPI coltype argument received in cursor.description.

5.2 Elements

class geoalchemy2.elements.**WKTElement**(*data*, *srid=-1*, *extended=False*)
Bases: geoalchemy2.elements._SpatialElement

Instances of this class wrap a WKT or EWKT value.

Usage examples:

```
wkt_element_1 = WKTElement('POINT(5 45)')
wkt_element_2 = WKTElement('POINT(5 45)', srid=4326)
wkt_element_3 = WKTElement('SRID=4326;POINT(5 45)', extended=True)
```

desc
This element's description string.

geom_from = 'ST_GeomFromText'

geom_from_extended_version = 'ST_GeomFromEWKT'

class geoalchemy2.elements.**WKBElement**(*data*, *srid=-1*, *extended=False*)
Bases: geoalchemy2.elements._SpatialElement

Instances of this class wrap a WKB or EWKB value.

Geometry values read from the database are converted to instances of this type. In most cases you won't need to create `WKBElement` instances yourself.

If `extended` is `True` and `srid` is `-1` at construction time then the SRID will be read from the EWKB data.

Note: you can create `WKBElement` objects from Shapely geometries using the *geoalchemy2.shape.from_shape()* function.

desc
This element's description string.

geom_from = 'ST_GeomFromWKB'

geom_from_extended_version = 'ST_GeomFromEWKB'

class geoalchemy2.elements.**RasterElement**(*data*)

Bases: sqlalchemy.sql.functions.FunctionElement

Instances of this class wrap a `raster` value. Raster values read from the database are converted to instances of this type. In most cases you won't need to create `RasterElement` instances yourself.

desc

This element's description string.

5.3 Spatial Functions

This module defines the *GenericFunction* class, which is the base for the implementation of spatial functions in GeoAlchemy. This module is also where actual spatial functions are defined. Spatial functions supported by GeoAlchemy are defined in this module. See *GenericFunction* to know how to create new spatial functions.

Note: By convention the names of spatial functions are prefixed by `ST_`. This is to be consistent with PostGIS', which itself is based on the `SQL-MM` standard.

Functions created by subclassing *GenericFunction* can be called in several ways:

- By using the `func` object, which is the SQLAlchemy standard way of calling a function. For example, without the ORM:

```
select([func.ST_Area(lake_table.c.geom)])
```

and with the ORM:

```
Session.query(func.ST_Area(Lake.geom))
```

- By applying the function to a geometry column. For example, without the ORM:

```
select([lake_table.c.geom.ST_Area()])
```

and with the ORM:

```
Session.query(Lake.geom.ST_Area())
```

- By applying the function to a *geoalchemy2.elements.WKBElement* object (*geoalchemy2.elements.WKBElement* is the type into which GeoAlchemy converts geometry values read from the database), or to a *geoalchemy2.elements.WKTElement* object. For example, without the ORM:

```
conn.scalar(lake['geom'].ST_Area())
```

and with the ORM:

```
session.scalar(lake.geom.ST_Area())
```

5.3.1 Reference

class geoalchemy2.functions.**GenericFunction**(**args*, ***kwargs*)

The base class for GeoAlchemy functions.

This class inherits from `sqlalchemy.sql.functions.GenericFunction`, so functions defined by subclassing this class can be given a fixed return type. For example, functions like *ST_Buffer* and *ST_Envelope* have their `type` attributes set to *geoalchemy2.types.Geometry*.

This class allows constructs like `Lake.geom.ST_Buffer(2)`. In that case the `Function` instance is bound to an expression (`Lake.geom` here), and that expression is passed to the function when the function is actually called.

If you need to use a function that GeoAlchemy does not provide you will certainly want to subclass this class. For example, if you need the `ST_TransScale` spatial function, which isn't (currently) natively supported by GeoAlchemy, you will write this:

```
from geoalchemy2 import Geometry
from geoalchemy2.functions import GenericFunction

class ST_TransScale(GenericFunction):
    name = 'ST_TransScale'
    type = Geometry
```

class geoalchemy2.functions.**ST_Area**(**args*, ***kwargs*)
Returns the area of the surface if it is a polygon or multi-polygon. For `geometry` type area is in SRID units. For `geography` area is in square meters.

see http://postgis.net/docs/ST_Area.html

class geoalchemy2.functions.**ST_AsBinary**(**args*, ***kwargs*)
Return the Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.

see http://postgis.net/docs/ST_AsBinary.html

class geoalchemy2.functions.**ST_AsEWKB**(**args*, ***kwargs*)
Return the Well-Known Binary (WKB) representation of the geometry/geography with SRID meta data.

see http://postgis.net/docs/ST_AsEWKB.html

class geoalchemy2.functions.**ST_AsEWKT**(**args*, ***kwargs*)
Return the Well-Known Text (WKT) representation of the geometry/geography with SRID metadata.

see http://postgis.net/docs/ST_AsEWKT.html

class geoalchemy2.functions.**ST_AsGML**(**args*, ***kwargs*)
Return the geometry as a GML version 2 or 3 element.

see http://postgis.net/docs/ST_AsGML.html

class geoalchemy2.functions.**ST_AsGeoJSON**(**args*, ***kwargs*)
Return the geometry as a GeoJSON element.

see http://postgis.net/docs/ST_AsGeoJSON.html

class geoalchemy2.functions.**ST_AsKML**(**args*, ***kwargs*)
Return the geometry as a KML element. Several variants. Default version=2, default precision=15

see http://postgis.net/docs/ST_AsKML.html

class geoalchemy2.functions.**ST_AsRaster**(**args*, ***kwargs*)
Converts a PostGIS geometry to a PostGIS raster.

see http://postgis.net/docs/RT_ST_AsRaster.html

Return type: *geoalchemy2.types.Raster*.

type
alias of *geoalchemy2.types.Raster*

class geoalchemy2.functions.**ST_AsSVG**(**args*, ***kwargs*)
Returns a Geometry in SVG path data given a geometry or geography object.

see http://postgis.net/docs/ST_AsSVG.html

class geoalchemy2.functions.**ST_AsTWKB**(**args*, ***kwargs*)
Returns the geometry as TWKB, aka "Tiny Well-Known Binary"

see http://postgis.net/docs/ST_AsTWKB.html

class geoalchemy2.functions.**ST_AsText**(**args*, ***kwargs*)
Return the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.

see http://postgis.net/docs/ST_AsText.html

class geoalchemy2.functions.**ST_Azimuth**(**args*, ***kwargs*)
Returns the angle in radians from the horizontal of the vector defined by pointA and pointB. Angle is computed clockwise from down-to-up: on the clock: 12=0; 3=PI/2; 6=PI; 9=3PI/2.

see http://postgis.net/docs/ST_Azimuth.html

class geoalchemy2.functions.**ST_Buffer**(**args*, ***kwargs*)
For geometry: Returns a geometry that represents all points whose distance from this Geometry is less than or equal to distance. Calculations are in the Spatial Reference System of this Geometry.

For geography: Uses a planar transform wrapper. Introduced in 1.5 support for different end cap and mitre settings to control shape.

see http://postgis.net/docs/ST_Buffer.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Centroid**(**args*, ***kwargs*)
Returns the geometric center of a geometry.

see http://postgis.net/docs/ST_Centroid.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Contains**(**args*, ***kwargs*)
Returns `True` if and only if no points of B lie in the exterior of A, and at least one point of the interior of B lies in the interior of A.

see http://postgis.net/docs/ST_Contains.html

class geoalchemy2.functions.**ST_ContainsProperly**(**args*, ***kwargs*)
Returns `True` if B intersects the interior of A but not the boundary (or exterior). A does not contain properly itself, but does contain itself.

see http://postgis.net/docs/ST_ContainsProperly.html

class geoalchemy2.functions.**ST_CoveredBy**(**args*, ***kwargs*)
Returns `True` if no point in Geometry/Geography A is outside Geometry/Geography B

see http://postgis.net/docs/ST_CoveredBy.html

class geoalchemy2.functions.**ST_Covers**(**args*, ***kwargs*)
Returns `True` if no point in Geometry B is outside Geometry A

see http://postgis.net/docs/ST_Covers.html

class geoalchemy2.functions.**ST_Crosses**(**args*, ***kwargs*)
Returns `True` if the supplied geometries have some, but not all, interior points in common.

see http://postgis.net/docs/ST_Crosses.html

class geoalchemy2.functions.**ST_DFullyWithin**(**args*, ***kwargs*)
Returns `True` if all of the geometries are within the specified distance of one another

see http://postgis.net/docs/ST_DFullyWithin.html

class geoalchemy2.functions.**ST_DWithin**(**args*, ***kwargs*)
Returns `True` if the geometries are within the specified distance of one another. For geometry units are in those of spatial reference and for geography units are in meters and measurement is defaulted to `use_spheroid=True` (measure around spheroid), for faster check, `use_spheroid=False` to measure along sphere.

see http://postgis.net/docs/ST_DWithin.html

class geoalchemy2.functions.**ST_Difference**(**args*, ***kwargs*)
Returns a geometry that represents that part of geometry A that does not intersect with geometry B.

see http://postgis.net/docs/ST_Difference.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Disjoint**(**args*, ***kwargs*)
Returns `True` if the Geometries do not "spatially intersect" - if they do not share any space together.

see http://postgis.net/docs/ST_Disjoint.html

class geoalchemy2.functions.**ST_Distance**(**args*, ***kwargs*)
For geometry type Returns the 2-dimensional cartesian minimum distance (based on spatial ref) between two geometries in projected units. For geography type defaults to return spheroidal minimum distance between two geographies in meters.

see http://postgis.net/docs/ST_Distance.html

class geoalchemy2.functions.**ST_Distance_Sphere**(**args*, ***kwargs*)
Returns minimum distance in meters between two lon/lat geometries. Uses a spherical earth and radius of 6370986 meters. Faster than `ST_Distance_Spheroid`, but less accurate. PostGIS versions prior to 1.5 only implemented for points.

see http://postgis.net/docs/ST_Distance_Sphere.html

class geoalchemy2.functions.**ST_Dump**(**args*, ***kwargs*)
Returns a set of geometry_dump (geom,path) rows, that make up a geometry g1.

see http://postgis.net/docs/ST_Dump.html

Return type: *geoalchemy2.types.GeometryDump*.

type
alias of *geoalchemy2.types.GeometryDump*

class geoalchemy2.functions.**ST_DumpPoints**(**args*, ***kwargs*)
Returns a set of geometry_dump (geom,path) rows of all points that make up a geometry.

see http://postgis.net/docs/ST_DumpPoints.html

Return type: *geoalchemy2.types.GeometryDump*.

type
alias of *geoalchemy2.types.GeometryDump*

class geoalchemy2.functions.**ST_Envelope**(**args*, ***kwargs*)
Returns a geometry representing the double precision (float8) boundingbox of the supplied geometry.

see http://postgis.net/docs/ST_Envelope.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Equals**(**args*, ***kwargs*)
Returns `True` if the given geometries represent the same geometry. Directionality is ignored.

see http://postgis.net/docs/ST_Equals.html

class geoalchemy2.functions.**ST_GeogFromText**(**args*, ***kwargs*)
Returns a geography object from the well-known text or extended well-known representation.

see http://postgis.net/docs/ST_GeogFromText.html

Return type: *geoalchemy2.types.Geography*.

type
alias of *geoalchemy2.types.Geography*

class geoalchemy2.functions.**ST_GeomFromEWKB**(**args*, ***kwargs*)
Constructs a PostGIS ST_Geometry object from the OGC Extended Well-Known binary (EWKB) representation.

see http://postgis.net/docs/ST_GeomFromEWKB.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_GeomFromEWKT**(**args*, ***kwargs*)
Constructs a PostGIS ST_Geometry object from the OGC Extended Well-Known text (EWKT) representation.

see http://postgis.net/docs/ST_GeomFromEWKT.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_GeomFromText**(**args*, ***kwargs*)
Constructs a PostGIS ST_Geometry object from the OGC Well-Known text representation.

see http://postgis.net/docs/ST_GeomFromText.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_GeometryN**(**args*, ***kwargs*)
Return the 1-based Nth geometry if the geometry is a `GEOMETRYCOLLECTION`, `(MULTI)POINT`, `(MULTI)LINESTRING`, `MULTICURVE` or `(MULTI)POLYGON`, `POLYHEDRALSURFACE` Otherwise, return `None`.

see http://postgis.net/docs/ST_GeometryN.html

class geoalchemy2.functions.**ST_GeometryType**(**args*, ***kwargs*)
Return the geometry type of the `ST_Geometry` value.

see http://postgis.net/docs/ST_GeometryType.html

class geoalchemy2.functions.**ST_Height**(**args*, ***kwargs*)
Returns the height of the raster in pixels.

see http://postgis.net/docs/RT_ST_Height.html

class geoalchemy2.functions.**ST_Intersection**(**args*, ***kwargs*)
Returns a geometry that represents the shared portion of geomA and geomB. The geography implementation does a transform to geometry to do the intersection and then transform back to WGS84.

see http://postgis.net/docs/ST_Intersection.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Intersects**(**args*, ***kwargs*)
Returns `True` if the Geometries/Geography "spatially intersect in 2D" - (share any portion of space) and `False` if they don't (they are Disjoint). For geography – tolerance is 0.00001 meters (so any points that close are considered to intersect)

see http://postgis.net/docs/ST_Intersects.html

class geoalchemy2.functions.**ST_IsValid**(**args*, ***kwargs*)
Returns `True` if the `ST_Geometry` is well formed.

see http://postgis.net/docs/ST_IsValid.html

class geoalchemy2.functions.**ST_Length**(**args*, ***kwargs*)
Returns the 2d length of the geometry if it is a linestring or multilinestring. geometry are in units of spatial reference and geography are in meters (default spheroid)

see http://postgis.net/docs/ST_Length.html

class geoalchemy2.functions.**ST_LineLocatePoint**(**args*, ***kwargs*)
Returns a float between 0 and 1 representing the location of the closest point on LineString to the given Point, as a fraction of total 2d line length.

see http://postgis.net/docs/ST_LineLocatePoint.html

class geoalchemy2.functions.**ST_LineMerge**(**args*, ***kwargs*)
Returns a (set of) LineString(s) formed by sewing together the constituent line work of a MULTILINESTRING.

see http://postgis.net/docs/ST_LineMerge.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_LineSubstring**(**args*, ***kwargs*)
Return a linestring being a substring of the input one starting and ending at the given fractions of total 2d length. Second and third arguments are float8 values between 0 and 1. This only works with LINESTRINGs. To use with contiguous MULTILINESTRINGs use in conjunction with ST_LineMerge.If 'start' and 'end' have the same value this is equivalent to ST_LineInterpolatePoint.

see http://postgis.net/docs/ST_LineSubstring.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_NPoints**(**args*, ***kwargs*)
Return the number of points (vertices) in a geometry.

see http://postgis.net/docs/ST_NPoints.html

class geoalchemy2.functions.**ST_OrderingEquals**(**args*, ***kwargs*)
Returns `True` if the given geometries represent the same geometry and points are in the same directional order.

see http://postgis.net/docs/ST_OrderingEquals.html

class geoalchemy2.functions.**ST_Overlaps**(**args*, ***kwargs*)
Returns `True` if the Geometries share space, are of the same dimension, but are not completely contained by each other.

see http://postgis.net/docs/ST_Overlaps.html

class geoalchemy2.functions.**ST_Perimeter**(**args*, ***kwargs*)
Return the length measurement of the boundary of an ST_Surface or ST_MultiSurface geometry or geography. (Polygon, Multipolygon). geometry measurement is in units of spatial reference and geography is in meters.

see http://postgis.net/docs/ST_Perimeter.html

class geoalchemy2.functions.**ST_Project**(**args*, ***kwargs*)
Returns a `POINT` projected from a start point using a distance in meters and bearing (azimuth) in radians.

see http://postgis.net/docs/ST_Project.html

Return type: *geoalchemy2.types.Geography*.

type
alias of *geoalchemy2.types.Geography*

class geoalchemy2.functions.**ST_Relate**(**args*, ***kwargs*)
Returns `True` if this Geometry is spatially related to anotherGeometry, by testing for intersections between the Interior, Boundary and Exterior of the two geometries as specified by the values in the intersectionMatrixPattern. If no intersectionMatrixPattern is passed in, then returns the maximum intersectionMatrixPattern that relates the 2 geometries.

see http://postgis.net/docs/ST_Relate.html

class geoalchemy2.functions.**ST_SRID**(**args*, ***kwargs*)
Returns the spatial reference identifier for the `ST_Geometry` as defined in `spatial_ref_sys` table.

see http://postgis.net/docs/ST_SRID.html

class geoalchemy2.functions.**ST_Simplify**(**args*, ***kwargs*)
Returns a "simplified" version of the given geometry using the Douglas-Peucker algorithm.

see http://postgis.net/docs/ST_Simplify.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Touches**(**args*, ***kwargs*)
Returns `True` if the geometries have at least one point in common, but their interiors do not intersect.

see http://postgis.net/docs/ST_Touches.html

class geoalchemy2.functions.**ST_Transform**(**args*, ***kwargs*)
Return a new geometry with its coordinates transformed to the SRID referenced by the integer parameter.

see http://postgis.net/docs/ST_Transform.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Union**(**args*, ***kwargs*)
Returns a geometry that represents the point set union of the Geometries.

see http://postgis.net/docs/ST_Union.html

Return type: *geoalchemy2.types.Geometry*.

type
alias of *geoalchemy2.types.Geometry*

class geoalchemy2.functions.**ST_Value**(**args*, ***kwargs*)
Returns the value of a given band in a given columnx, rowy pixel or at a particular geometric point. Band numbers start at 1 and assumed to be 1 if not specified. If `exclude_nodata_value` is set to `false`, then all pixels include nodata pixels are considered to intersect and return value. If `exclude_nodata_value` is not passed in then reads it from metadata of raster.

see http://postgis.net/docs/RT_ST_Value.html

class geoalchemy2.functions.**ST_Width**(**args*, ***kwargs*)
Returns the width of the raster in pixels.

see http://postgis.net/docs/RT_ST_Width.html

class geoalchemy2.functions.**ST_Within**(**args*, ***kwargs*)
Returns `True` if the geometry A is completely inside geometry B

see http://postgis.net/docs/ST_Within.html

class geoalchemy2.functions.**ST_X**(**args*, ***kwargs*)
Return the X coordinate of the point, or `None` if not available. Input must be a point.

see http://postgis.net/docs/ST_X.html

class geoalchemy2.functions.**ST_Y**(**args*, ***kwargs*)
Return the Y coordinate of the point, or `None` if not available. Input must be a point.

see http://postgis.net/docs/ST_Y.html

class geoalchemy2.functions.**ST_Z**(**args*, ***kwargs*)
Return the Z coordinate of the point, or `None` if not available. Input must be a point.

see http://postgis.net/docs/ST_Z.html

5.4 Spatial Operators

This module defines a `Comparator` class for use with geometry and geography objects. This is where spatial operators, like `&&`, `&<`, are defined. Spatial operators very often apply to the bounding boxes of geometries. For example, `geom1 && geom2` indicates if geom1's bounding box intersects geom2's.

5.4.1 Examples

Select the objects whose bounding boxes are to the left of the bounding box of `POLYGON((-5 45,5 45,5 -45, -5 -45,-5 45))`:

```
select([table]).where(table.c.geom.to_left(
    'POLYGON((-5 45,5 45,5 -45,-5 -45,-5 45))'))
```

The << and >> operators are a bit specific, because they have corresponding Python operator (`__lshift__` and `__rshift__`). The above `SELECT` expression can thus be rewritten like this:

```
select([table]).where(
    table.c.geom << 'POLYGON((-5 45,5 45,5 -45,-5 -45,-5 45))')
```

Operators can also be used when using the ORM. For example:

```
Session.query(Cls).filter(
    Cls.geom << 'POLYGON((-5 45,5 45,5 -45,-5 -45,-5 45))')
```

Now some other examples with the <#> operator.

Select the ten objects that are the closest to `POINT(0 0)` (typical closed neighbors problem):

```
select([table]).order_by(table.c.geom.distance_box('POINT(0 0)')).limit(10)
```

Using the ORM:

```
Session.query(Cls).order_by(Cls.geom.distance_box('POINT(0 0)')).limit(10)
```

5.4.2 Reference

class geoalchemy2.comparator.**BaseComparator**(*expr*)

Bases: sqlalchemy.sql.type_api.Comparator

A custom comparator base class. It adds the ability to call spatial functions on columns that use this kind of comparator. It also defines functions that map to operators supported by `Geometry`, `Geography` and `Raster` columns.

This comparator is used by the *geoalchemy2.types.Raster*.

__weakref__
list of weak references to the object (if defined)

intersects(*other*)
The `&&` operator. A's BBOX intersects B's.

overlaps_or_to_left(*other*)
The `&<` operator. A's BBOX overlaps or is to the left of B's.

overlaps_or_to_right(*other*)
The `&>` operator. A's BBOX overlaps or is to the right of B's.

class geoalchemy2.comparator.**Comparator**(*expr*)

Bases: *geoalchemy2.comparator.BaseComparator*

A custom comparator class. Used in *geoalchemy2.types.Geometry* and *geoalchemy2.types.Geography*.

This is where spatial operators like << and <-> are defined.

__lshift__(*other*)
The << operator. A's BBOX is strictly to the left of B's. Same as `to_left`, so:

```
table.c.geom << 'POINT(1 2)'
```

is the same as:

```
table.c.geom.to_left('POINT(1 2)')
```

__rshift__(*other*)
The >> operator. A's BBOX is strictly to the left of B's. Same as *to_'right'*, so:

```
table.c.geom >> 'POINT(1 2)'
```

is the same as:

```
table.c.geom.to_right('POINT(1 2)')
```

above(*other*)
The |>> operator. A's BBOX is strictly above B's.

below(*other*)
The <<| operator. A's BBOX is strictly below B's.

contained(*other*)
The @ operator. A's BBOX is contained by B's.

contains(*other*, ***kw*)
The ~ operator. A's BBOX contains B's.

distance_box(*other*)
The <#> operator. The distance between bounding box of two geometries.

distance_centroid(*other*)
The <-> operator. The distance between two points.

overlaps_or_above(*other*)
The |&> operator. A's BBOX overlaps or is above B's.

overlaps_or_below(*other*)
The &<| operator. A's BBOX overlaps or is below B's.

same(*other*)
The ~= operator. A's BBOX is the same as B's.

to_left(*other*)
The << operator. A's BBOX is strictly to the left of B's.

to_right(*other*)
The >> operator. A's BBOX is strictly to the right of B's.

5.5 Shapely Integration

This module provides utility functions for integrating with Shapely.

Note: As GeoAlchemy 2 itself has no dependency on *Shapely*, applications using functions of this module have to ensure that *Shapely* is available.

geoalchemy2.shape.**from_shape**(*shape*, *srid=-1*)

Function to convert a Shapely geometry to a `geoalchemy2.types.WKBElement`.

Additional arguments:

`srid`

An integer representing the spatial reference system. E.g. 4326. Default value is -1, which means no/unknown reference system.

Example:

```
from shapely.geometry import Point
wkb_element = from_shape(Point(5, 45), srid=4326)
```

geoalchemy2.shape.**to_shape**(*element*)

Function to convert a `geoalchemy2.types.SpatialElement` to a Shapely geometry.

Example:

```
lake = Session.query(Lake).get(1)
polygon = to_shape(lake.geom)
```

CHAPTER 6

Development

The code is available on GitHub: https://github.com/geoalchemy/geoalchemy2.

Contributors:

- Adrien Berchet (https://github.com/adrien-berchet)
- Éric Lemoine (https://github.com/elemoine)
- Dolf Andringa (https://github.com/dolfandringa)
- Frédéric Junod, Camptocamp SA (https://github.com/fredj)
- ijl (https://github.com/ijl)
- Loïc Gasser (https://github.com/loicgasser)
- Marcel Radischat (https://github.com/quiqua)
- rapto (https://github.com/rapto)
- Serge Bouchut (https://github.com/SergeBouchut)
- Tobias Bieniek (https://github.com/Turbo87)
- Tom Payne (https://github.com/twpayne)

Many thanks to Mike Bayer for his guidance and support! He also fostered the birth of GeoAlchemy 2.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

B

C

D

F

G

T

W