# GeoAlchemy2 Documentation

***Release 0.8.1***

**Eric Lemoine**

**May 15, 2020**

# 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;
↪'0103000000010000000500000000000000000000000000000000000000000000000000000000f03f00000000000000000000000
↪'>
>>> 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 documentation 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 Reflecting tables

The reflection system of SQLAlchemy can be used on tables containing `geoalchemy2.types.Geometry` or `geoalchemy2.types.Geography` columns. In this case, the type must be imported to be registered into SQLAlchemy, even if it is not used explicitely.

```
>>> from geoalchemy2 import Geometry  # <= not used but must be imported
>>> from sqlalchemy import create_engine, MetaData
>>> engine = create_engine("postgresql://myuser:mypass@mydb.host.tld/mydbname")
>>> meta = MetaData()
>>> meta.reflect(bind=engine)
```

### 4.2.5 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.

### 4.2.6 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:
```

(continues on next page)

(continued from previous page)

```
...     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.

### 4.2.7 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
```

## 4.2.8 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;
↪'0103000000010000000500000000000000000000000000000000000000000000000000000000f03f000000000000000000000000
↪'>
>>> 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)')))
```

(continues on next page)

(continued from previous page)

```
...
>>> 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

# Gallery

## 5.1 Gallery

### 5.1.1 Automatically use a function at insert or select

Sometimes the application wants to apply a function in an insert or in a select. For example, the application might need the geometry with lat/lon coordinates while they are projected in the DB. To avoid having to always tweak the query with a `ST_Transform()`, it is possible to define a TypeDecorator

```
from pkg_resources import parse_version
import pytest

import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import MetaData
from sqlalchemy import Column
from sqlalchemy import Integer
from sqlalchemy import func
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy.types import TypeDecorator

from geoalchemy2.compat import PY3
from geoalchemy2 import Geometry
from geoalchemy2 import shape


engine = create_engine('postgresql://gis:gis@localhost/gis', echo=True)
metadata = MetaData(engine)

Base = declarative_base(metadata=metadata)


```

(continues on next page)

(continued from previous page)

```
class TransformedGeometry(TypeDecorator):
    """This class is used to insert a ST_Transform() in each insert or select."""
    impl = Geometry

    def __init__(self, db_srid, app_srid, **kwargs):
        kwargs["srid"] = db_srid
        self.impl = self.__class__.impl(**kwargs)
        self.app_srid = app_srid
        self.db_srid = db_srid

    def column_expression(self, col):
        """The column_expression() method is overrided to ensure that the
        SRID of the resulting WKBElement is correct"""
        return getattr(func, self.impl.as_binary)(
            func.ST_Transform(col, self.app_srid),
            type_=self.__class__.impl(srid=self.app_srid)
            # srid could also be -1 so that the SRID is deduced from the
            # WKB data
        )

    def bind_expression(self, bindvalue):
        return func.ST_Transform(
            self.impl.bind_expression(bindvalue), self.db_srid)


class ThreeDGeometry(TypeDecorator):
    """This class is used to insert a ST_Force3D() in each insert."""
    impl = Geometry

    def bind_expression(self, bindvalue):
        return func.ST_Force3D(self.impl.bind_expression(bindvalue))


class Point(Base):
    __tablename__ = "point"
    id = Column(Integer, primary_key=True)
    raw_geom = Column(Geometry(srid=4326, geometry_type="POINT"))
    geom = Column(
        TransformedGeometry(
            db_srid=2154, app_srid=4326, geometry_type="POINT"))
    three_d_geom = Column(
        ThreeDGeometry(srid=4326, geometry_type="POINTZ", dimension=3))


session = sessionmaker(bind=engine)()


def check_wkb(wkb, x, y):
    pt = shape.to_shape(wkb)
    assert round(pt.x, 5) == x
    assert round(pt.y, 5) == y


class TestTypeDecorator():

    def setup(self):
        metadata.drop_all(checkfirst=True)
```

(continues on next page)

(continued from previous page)

```
        metadata.create_all()

    def teardown(self):
        session.rollback()
        metadata.drop_all()

    def _create_one_point(self):
        # Create new point instance
        p = Point()
        p.raw_geom = "SRID=4326;POINT(5 45)"
        p.geom = "SRID=4326;POINT(5 45)"
        p.three_d_geom = "SRID=4326;POINT(5 45)"  # Insert 2D geometry into 3D column

        # Insert point
        session.add(p)
        session.flush()
        session.expire(p)

        return p.id

    def test_transform(self):
        self._create_one_point()

        # Query the point and check the result
        pt = session.query(Point).one()
        assert pt.id == 1
        assert pt.raw_geom.srid == 4326
        check_wkb(pt.raw_geom, 5, 45)

        assert pt.geom.srid == 4326
        check_wkb(pt.geom, 5, 45)

        # Check that the data is correct in DB using raw query
        q = "SELECT id, ST_AsEWKT(geom) AS geom FROM point;"
        res_q = session.execute(q).fetchone()
        assert res_q.id == 1
        assert res_q.geom == "SRID=2154;POINT(857581.899319668 6435414.7478354)"

        # Compare geom, raw_geom with auto transform and explicit transform
        pt_trans = session.query(
            Point,
            Point.raw_geom,
            func.ST_Transform(Point.raw_geom, 2154).label("trans")
        ).one()

        assert pt_trans[0].id == 1

        assert pt_trans[0].geom.srid == 4326
        check_wkb(pt_trans[0].geom, 5, 45)

        assert pt_trans[0].raw_geom.srid == 4326
        check_wkb(pt_trans[0].raw_geom, 5, 45)

        assert pt_trans[1].srid == 4326
        check_wkb(pt_trans[1], 5, 45)

        assert pt_trans[2].srid == 2154
```

(continues on next page)

(continued from previous page)

```
        check_wkb(pt_trans[2], 857581.89932, 6435414.74784)

    @pytest.mark.skipif(
        not PY3 and parse_version(str(sqlalchemy.__version__)) < parse_version("1.3"),
        reason="Need sqlalchemy >= 1.3")
    def test_force_3d(self):
        self._create_one_point()

        # Query the point and check the result
        pt = session.query(Point).one()

        assert pt.id == 1
        assert pt.three_d_geom.srid == 4326
        assert pt.three_d_geom.desc.lower() == (
            '01010000a0e6100000000000000000144000000000000080464000000000000000000')
```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

### 5.1.2 Compute length on insert

It is possible to insert a geometry and ask PostgreSQL to compute its length at the same time. This example uses SQLAlchemy core queries.

```
from sqlalchemy import bindparam
from sqlalchemy import Column
from sqlalchemy import create_engine
from sqlalchemy import Float
from sqlalchemy import func
from sqlalchemy import Integer
from sqlalchemy import MetaData
from sqlalchemy import select
from sqlalchemy import Table

from geoalchemy2 import Geometry
from geoalchemy2.shape import to_shape


engine = create_engine('postgresql://gis:gis@localhost/gis', echo=True)
metadata = MetaData(engine)

table = Table(
    "inserts",
    metadata,
    Column("id", Integer, primary_key=True),
    Column("geom", Geometry("LINESTRING", 4326)),
    Column("distance", Float),
)


class TestLengthAtInsert():

    def setup(self):
        self.conn = engine.connect()
        metadata.drop_all(checkfirst=True)
```

(continues on next page)

(continued from previous page)

```python
        metadata.create_all()

    def teardown(self):
        self.conn.close()
        metadata.drop_all()

    def test_query(self):
        conn = self.conn

        # Define geometries to insert
        values = [
            {"ewkt": "SRID=4326;LINESTRING(0 0, 1 0)"},
            {"ewkt": "SRID=4326;LINESTRING(0 0, 0 1)"}
        ]

        # Define the query to compute distance (without spheroid)
        distance = func.ST_Length(func.ST_GeomFromText(bindparam("ewkt")), False)

        i = table.insert()
        i = i.values(geom=bindparam("ewkt"), distance=distance)

        # Execute the query with values as parameters
        conn.execute(i, values)

        # Check the result
        q = select([table])
        res = conn.execute(q).fetchall()

        # Check results
        assert len(res) == 2

        r1 = res[0]
        assert r1[0] == 1
        assert r1[1].srid == 4326
        assert to_shape(r1[1]).wkt == "LINESTRING (0 0, 1 0)"
        assert round(r1[2]) == 111195

        r2 = res[1]
        assert r2[0] == 2
        assert r2[1].srid == 4326
        assert to_shape(r2[1]).wkt == "LINESTRING (0 0, 0 1)"
        assert round(r2[2]) == 111195
```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

### 5.1.3 Disable wrapping in select

If the application wants to build queries with GeoAlchemy 2 and gets them as strings, the wrapping of geometry columns with a *ST_AsEWKB()* function might be annoying. In this case it is possible to disable this wrapping. This example uses SQLAlchemy ORM queries.

```python
from sqlalchemy import Column
from sqlalchemy import Integer
from sqlalchemy import func
from sqlalchemy import select
```

(continues on next page)

(continued from previous page)

```python
from sqlalchemy.ext.declarative import declarative_base

from geoalchemy2 import Geometry


Base = declarative_base()


class RawGeometry(Geometry):
    """This class is used to remove the 'ST_AsEWKB()'' function from select queries"""

    def column_expression(self, col):
        return col


class Point(Base):
    __tablename__ = "point"
    id = Column(Integer, primary_key=True)
    geom = Column(Geometry(srid=4326, geometry_type="POINT"))
    raw_geom = Column(
        RawGeometry(srid=4326, geometry_type="POINT"))


def test_no_wrapping():
    # Select all columns
    select_query = select([Point])

    # Check that the 'geom' column is wrapped by 'ST_AsEWKB()' and that the column
    # 'raw_geom' is not.
    assert str(select_query) == (
        "SELECT point.id, ST_AsEWKB(point.geom) AS geom, point.raw_geom \n"
        "FROM point"
    )


def test_func_no_wrapping():
    # Select query with function
    select_query = select([
        func.ST_Buffer(Point.geom),  # with wrapping (default behavior)
        func.ST_Buffer(Point.geom, type_=Geometry),  # with wrapping
        func.ST_Buffer(Point.geom, type_=RawGeometry)  # without wrapping
    ])

    # Check the query
    assert str(select_query) == (
        "SELECT "
        "ST_AsEWKB(ST_Buffer(point.geom)) AS \"ST_Buffer_1\", "
        "ST_AsEWKB(ST_Buffer(point.geom)) AS \"ST_Buffer_2\", "
        "ST_Buffer(point.geom) AS \"ST_Buffer_3\" \n"
        "FROM point"
    )
```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

The *Gallery* page shows examples of the GeoAlchemy 2's functionalities.

CHAPTER 6

# Reference Documentation

## 6.1 Types

## 6.2 Elements

## 6.3 Spatial Functions

## 6.4 Spatial Operators

## 6.5 Shapely Integration

# CHAPTER 7

# 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 8

# Indices and tables

- genindex
- modindex
- search