SQLAlchemy 0.5.6 Documentation

Version: **0.5.6** Last Updated: 12/07/2009 14:40:39

In this tutorial we will cover a basic SQLAlchemy object-relational mapping scenario, where we store and retrieve Python objects from a database representation. The tutorial is in doctest format, meaning each `>>>` line represents something you can type at a Python command prompt, and the following text represents the expected return value.

Version Check

A quick check to verify that we are on at least **version 0.5** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
0.5.0
```

Connecting

For this tutorial we will use an in-memory-only SQLite database. To connect we use `create_engine()`:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:', echo=True)
```

The `echo` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard `logging` module. With it enabled, we'll see all the generated SQL produced. If you are working through this tutorial and want less output generated, set it to `False`. This tutorial will format the SQL behind a popup window so it doesn't get in our way; just click the "SQL" links to see what's being generated.

Define and Create a Table

Next we want to tell SQLAlchemy about our tables. We will start with just a single table called `users`, which will store records for the end-users using our application (lets assume it's a website). We define our tables within a catalog called `MetaData`, using the `Table` construct, which is used in a manner similar to SQL's CREATE TABLE syntax:

```
>>> from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey
>>> metadata = MetaData()
>>> users_table = Table('users', metadata,
...     Column('id', Integer, primary_key=True),
...     Column('name', String),
...     Column('fullname', String),
...     Column('password', String)
... )
```

All about how to define `Table` objects, as well as how to load their definition from an existing database (known as **reflection**), is described in *Database Meta Data*.

Next, we can issue CREATE TABLE statements derived from our table metadata, by calling `create_all()` and passing it the `engine` instance which points to our database. This will check for the presence of a table first before creating, so it's safe to call multiple times:

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

Users familiar with the syntax of CREATE TABLE may notice that the VARCHAR columns were generated without a length; on SQLite, this is a valid datatype, but on most databases it's not allowed. So if running this tutorial on a database such as PostgreSQL or MySQL, and you wish to use SQLAlchemy to generate the tables, a "length" may be provided to the `String` type as below:

```
Column('name', String(50))
```

The length field on `String`, as well as similar precision/scale fields available on `Integer`, `Numeric`, etc. are not referenced by SQLAlchemy other than when creating tables.

Define a Python Class to be Mapped

While the `Table` object defines information about our database, it does not say anything about the definition or behavior of the business objects used by our application; SQLAlchemy views this as a separate concern. To correspond to our `users` table, let's create a rudimentary `User` class. It only need subclass Python's built-in `object` class (i.e. it's a new style class):

```
>>> class User(object):
...     def __init__(self, name, fullname, password):
...         self.name = name
...         self.fullname = fullname
...         self.password = password
...
...     def __repr__(self):
...        return "<User('%s','%s', '%s')>" % (self.name, self.fullname, self.password)
```

The class has an `__init__()` and a `__repr__()` method for convenience. These methods are both entirely optional, and can be of any form. SQLAlchemy never calls `__init__()` directly.

Setting up the Mapping

With our `users_table` and `User` class, we now want to map the two together. That's where the SQLAlchemy ORM package comes in. We'll use the `mapper` function to create a **mapping** between `users_table` and `User`:

```
>>> from sqlalchemy.orm import mapper
>>> mapper(User, users_table)
<Mapper at 0x...; User>
```

The `mapper()` function creates a new `Mapper` object and stores it away for future reference, associated with our class. Let's now create and inspect a `User` object:

```
>>> ed_user = User('ed', 'Ed Jones', 'edspassword')
>>> ed_user.name
'ed'
>>> ed_user.password
'edspassword'
>>> str(ed_user.id)
'None'
```

The `id` attribute, which while not defined by our `__init__()` method, exists due to the `id` column present within the `users_table` object. By default, the `mapper` creates class attributes for all columns present within the `Table`. These class attributes exist as Python descriptors, and define **instrumentation** for the mapped class. The functionality of this instrumentation is very rich and includes the ability to track modifications and automatically load new data from the database when needed.

Since we have not yet told SQLAlchemy to persist `Ed Jones` within the database, its id is `None`. When we persist the object later, this attribute will be populated with a newly generated value.

Creating Table, Class and Mapper All at Once Declaratively

The preceding approach to configuration involving a `Table`, user-defined class, and `mapper()` call illustrate classical SQLAlchemy usage, which values the highest separation of concerns possible. A large number of applications don't require this degree of separation, and for those SQLAlchemy offers an alternate "shorthand" configurational style called **declarative**. For many applications, this is the only style of configuration needed. Our above example using this style is as follows:

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

>>> Base = declarative_base()
>>> class User(Base):
...     __tablename__ = 'users'
...
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
...     def __init__(self, name, fullname, password):
...         self.name = name
...         self.fullname = fullname
...         self.password = password
...
...     def __repr__(self):
...        return "<User('%s','%s', '%s')>" % (self.name, self.fullname, self.password)
```

Above, the `declarative_base()` function defines a new class which we name `Base`, from which all of our ORM-enabled classes will derive. Note that we define `Column` objects with no “name” field, since it’s inferred from the given attribute name.

The underlying `Table` object created by our `declarative_base()` version of `User` is accessible via the `__table__` attribute:

```
>>> users_table = User.__table__
```

and the owning `MetaData` object is available as well:

```
>>> metadata = Base.metadata
```

Yet another “declarative” method is available for SQLAlchemy as a third party library called Elixir. This is a full-featured configurational product which also includes many higher level mapping configurations built in. Like declarative, once classes and mappings are defined, ORM usage is the same as with a classical SQLAlchemy configuration.

Creating a Session

We’re now ready to start talking to the database. The ORM’s “handle” to the database is the `Session`. When we first set up the application, at the same level as our `create_engine()` statement, we define a `Session` class which will serve as a factory for new `Session` objects:

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

In the case where your application does not yet have an `Engine` when you define your module-level objects, just set it up like this:

```
>>> Session = sessionmaker()
```

Later, when you create your engine with `create_engine()`, connect it to the `Session` using `configure()`:

```
>>> Session.configure(bind=engine)  # once engine is available
```

This custom-made `Session` class will create new `Session` objects which are bound to our database. Other transactional characteristics may be defined when calling `sessionmaker()` as well; these are described in a later chapter. Then, whenever you need to have a conversation with the database, you instantiate a `Session`:

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

The above `Session` is associated with our SQLite `engine`, but it hasn't opened any connections yet. When it's first used, it retrieves a connection from a pool of connections maintained by the `engine`, and holds onto it until we commit all changes and/or close the session object.

Adding new Objects

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

```
>>> ed_user = User('ed', 'Ed Jones', 'edspassword')
>>> session.add(ed_user)
```

At this point, the instance is **pending**; no SQL has yet been issued. The `Session` will issue the SQL to persist `Ed Jones` as soon as is needed, using a process known as a **flush**. If we query the database for `Ed Jones`, all pending information will first be flushed, and the query is issued afterwards.

For example, below we create a new `Query` object which loads instances of `User`. We "filter by" the `name` attribute of `ed`, and indicate that we'd like only the first result in the full list of rows. A `User` instance is returned which is equivalent to that which we've added:

```
>>> our_user = session.query(User).filter_by(name='ed').first()                SQL
>>> our_user
<User('ed','Ed Jones', 'edspassword')>
```

In fact, the `Session` has identified that the row returned is the **same** row as one already represented within its internal map of objects, so we actually got back the identical instance as that which we just added:

```
>>> ed_user is our_user
True
```

The ORM concept at work here is known as an **identity map** and ensures that all operations upon a particular row within a `Session` operate upon the same set of data. Once an object with a particular primary key is present in the `Session`, all SQL queries on that `Session` will always return

the same Python object for that particular primary key; it also will raise an error if an attempt is made to place a second, already-persisted object with the same primary key within the session.

We can add more `User` objects at once using `add_all()`:

```
>>> session.add_all([
...     User('wendy', 'Wendy Williams', 'foobar'),
...     User('mary', 'Mary Contrary', 'xxg527'),
...     User('fred', 'Fred Flinstone', 'blah')])
```

Also, Ed has already decided his password isn't too secure, so lets change it:

```
>>> ed_user.password = 'f8s7ccs'
```

The `Session` is paying attention. It knows, for example, that `Ed Jones` has been modified:

```
>>> session.dirty
IdentitySet([<User('ed','Ed Jones', 'f8s7ccs')>])
```

and that three new `User` objects are pending:

```
>>> session.new
IdentitySet([<User('wendy','Wendy Williams', 'foobar')>,
<User('mary','Mary Contrary', 'xxg527')>,
<User('fred','Fred Flinstone', 'blah')>])
```

We tell the `Session` that we'd like to issue all remaining changes to the database and commit the transaction, which has been in progress throughout. We do this via `commit()`:

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

`commit()` flushes whatever remaining changes remain to the database, and commits the transaction. The connection resources referenced by the session are now returned to the connection pool. Subsequent operations with this session will occur in a **new** transaction, which will again re-acquire connection resources when first needed.

If we look at Ed's `id` attribute, which earlier was `None`, it now has a value:

```
>>> ed_user.id                                                          SQL
1
```

After the `Session` inserts new rows in the database, all newly generated identifiers and database-generated defaults become available on the instance, either immediately or via load-on-first-access. In this case, the entire row was re-loaded on access because a new transaction was begun after we issued `commit()`. SQLAlchemy by default refreshes data from a previous transaction the first time it's accessed within a new transaction, so that the most recent state is available. The level of reloading is configurable as is described in the chapter on Sessions.

Rolling Back

Since the `Session` works within a transaction, we can roll back changes made too. Let's make two changes that we'll revert; `ed_user`'s user name gets set to `Edwardo`:

```
>>> ed_user.name = 'Edwardo'
```

and we'll add another erroneous user, `fake_user`:

```
>>> fake_user = User('fakeuser', 'Invalid', '12345')
>>> session.add(fake_user)
```

Querying the session, we can see that they're flushed into the current transaction:

```
>>> session.query(User).filter(User.name.in_(['Edwardo', 'fakeuser'])).all()                SQL
[<User('Edwardo','Ed Jones', 'f8s7ccs')>, <User('fakeuser','Invalid', '12345')>]
```

Rolling back, we can see that `ed_user`'s name is back to `ed`, and `fake_user` has been kicked out of the session:

```
>>> session.rollback()                                                                      SQL
>>> ed_user.name                                                                  SQL
u'ed'
>>> fake_user in session
False
```

issuing a SELECT illustrates the changes made to the database:

```
>>> session.query(User).filter(User.name.in_(['ed', 'fakeuser'])).all()                     SQL
[<User('ed','Ed Jones', 'f8s7ccs')>]
```

Querying

A `Query` is created using the `query()` function on `Session`. This function takes a variable number of arguments, which can be any combination of classes and class-instrumented descriptors. Below, we indicate a `Query` which loads `User` instances. When evaluated in an iterative context, the list of `User` objects present is returned:

```
>>> for instance in session.query(User).order_by(User.id):                                  SQL
...     print instance.name, instance.fullname
ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

The `Query` also accepts ORM-instrumented descriptors as arguments. Any time multiple class entities or column-based entities are expressed as arguments to the `query()` function, the return

result is expressed as tuples:

```
>>> for name, fullname in session.query(User.name, User.fullname):
...     print name, fullname
ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

The tuples returned by `Query` are *named* tuples, and can be treated much like an ordinary Python object. The names are the same as the attribute's name for an attribute, and the class name for a class:

```
>>> for row in session.query(User, User.name).all():
...     print row.User, row.name
<User('ed','Ed Jones', 'f8s7ccs')> ed
<User('wendy','Wendy Williams', 'foobar')> wendy
<User('mary','Mary Contrary', 'xxg527')> mary
<User('fred','Fred Flinstone', 'blah')> fred
```

You can control the names using the `label()` construct for scalar attributes and `aliased()` for class constructs:

```
>>> from sqlalchemy.orm import aliased
>>> user_alias = aliased(User, name='user_alias')
>>> for row in session.query(user_alias, user_alias.name.label('name_label')).all():
...     print row.user_alias, row.name_label
```

Basic operations with `Query` include issuing LIMIT and OFFSET, most conveniently using Python array slices and typically in conjunction with ORDER BY:

```
>>> for u in session.query(User).order_by(User.id)[1:3]:
...     print u
<User('wendy','Wendy Williams', 'foobar')>
<User('mary','Mary Contrary', 'xxg527')>
```

and filtering results, which is accomplished either with `filter_by()`, which uses keyword arguments:

```
>>> for name, in session.query(User.name).filter_by(fullname='Ed Jones'):
...     print name
ed
```

...or `filter()`, which uses more flexible SQL expression language constructs. These allow you to use regular Python operators with the class-level attributes on your mapped class:

```
>>> for name, in session.query(User.name).filter(User.fullname=='Ed Jones'):
...     print name
ed
```

The `Query` object is fully *generative*, meaning that most method calls return a new `Query` object

upon which further criteria may be added. For example, to query for users named “ed” with a full name of “Ed Jones”, you can call `filter()` twice, which joins criteria using `AND`:

```
>>> for user in session.query(User).filter(User.name=='ed').filter(User.fullname=='Ed Jones'):
...     print user
<User('ed','Ed Jones', 'f8s7ccs')>
```

Common Filter Operators

Here’s a rundown of some of the most common operators used in `filter()`:

- equals:

```
query.filter(User.name == 'ed')
```

- not equals:

```
query.filter(User.name != 'ed')
```

- LIKE:

```
query.filter(User.name.like('%ed%'))
```

- IN:

```
query.filter(User.name.in_(['ed', 'wendy', 'jack']))

# works with query objects too:

query.filter(User.name.in_(session.query(User.name).filter(User.name.like('%ed%'))))
```

- NOT IN:

```
query.filter(~User.name.in_(['ed', 'wendy', 'jack']))
```

- IS NULL:

```
filter(User.name == None)
```

- IS NOT NULL:

```
filter(User.name != None)
```

- AND:

```
from sqlalchemy import and_
filter(and_(User.name == 'ed', User.fullname == 'Ed Jones'))

# or call filter()/filter_by() multiple times
filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')
```

- OR:

```
from sqlalchemy import or_
filter(or_(User.name == 'ed', User.name == 'wendy'))
```

- match:

```
query.filter(User.name.match('wendy'))
```

The contents of the match parameter are database backend specific.

Returning Lists and Scalars

The `all()`, `one()`, and `first()` methods of `Query` immediately issue SQL and return a non-iterator value. `all()` returns a list:

```
>>> query = session.query(User).filter(User.name.like('%ed')).order_by(User.id)
>>> query.all()
[<User('ed','Ed Jones', 'f8s7ccs')>, <User('fred','Fred Flinstone', 'blah')>]
```

SQL

`first()` applies a limit of one and returns the first result as a scalar:

```
>>> query.first()
<User('ed','Ed Jones', 'f8s7ccs')>
```

SQL

`one()`, applies a limit of *two*, and if not exactly one row returned, raises an error:

```
>>> from sqlalchemy.orm.exc import MultipleResultsFound
>>> try:
...     user = query.one()
... except MultipleResultsFound, e:
...     print e
Multiple rows were found for one()
```

SQL

```
>>> from sqlalchemy.orm.exc import NoResultFound
>>> try:
...     user = query.filter(User.id == 99).one()
... except NoResultFound, e:
...     print e
No row was found for one()
```

SQL

Using Literal SQL

Literal strings can be used flexibly with `Query`. Most methods accept strings in addition to SQLAlchemy clause constructs. For example, `filter()` and `order_by()`:

```
>>> for user in session.query(User).filter("id<224").order_by("id").all():
...     print user.name
ed
wendy
mary
fred
```

Bind parameters can be specified with string-based SQL, using a colon. To specify the values, use the `params()` method:

```
>>> session.query(User).filter("id<:value and name=:name").\
...     params(value=224, name='fred').order_by(User.id).one()
```

To use an entirely string-based statement, using `from_statement()`; just ensure that the columns clause of the statement contains the column names normally used by the mapper (below illustrated using an asterisk):

```
>>> session.query(User).from_statement("SELECT * FROM users where name=:name").params(name=
[<User('ed','Ed Jones', 'f8s7ccs')>]
```

Counting

`Query` includes a convenience method for counting called `count()`:

```
>>> session.query(User).filter(User.name.like('%ed')).count()
2
```

The `count()` method is used to determine how many rows the SQL statement would return, and is mainly intended to return a simple count of a single type of entity, in this case `User`. For more complicated sets of columns or entities where the "thing to be counted" needs to be indicated more specifically, `count()` is probably not what you want. Below, a query for individual columns does return the expected result:

```
>>> session.query(User.id, User.name).filter(User.name.like('%ed')).count()
2
```

...but if you look at the generated SQL, SQLAlchemy saw that we were placing individual column expressions and decided to wrap whatever it was we were doing in a subquery, so as to be assured that it returns the "number of rows". This defensive behavior is not really needed here and in other cases is not what we want at all, such as if we wanted a grouping of counts per name:

```
>>> session.query(User.name).group_by(User.name).count()                    SQL
4
```

We don't want the number `4`, we wanted some rows back. So for detailed queries where you need to count something specific, use the `func.count()` function as a column expression:

```
>>> from sqlalchemy import func
>>> session.query(func.count(User.name), User.name).group_by(User.name).all()    SQL
[]
[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]
```

Building a Relation

Now let's consider a second table to be dealt with. Users in our system also can store any number of email addresses associated with their username. This implies a basic one to many association from the `users_table` to a new table which stores email addresses, which we will call `addresses`. Using declarative, we define this table along with its mapped class, `Address`:

```
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy.orm import relation, backref
>>> class Address(Base):
...     __tablename__ = 'addresses'
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey('users.id'))
...
...     user = relation(User, backref=backref('addresses', order_by=id))
...
...     def __init__(self, email_address):
...         self.email_address = email_address
...
...     def __repr__(self):
...         return "<Address('%s')>" % self.email_address
```

The above class introduces a **foreign key** constraint which references the `users` table. This defines for SQLAlchemy the relationship between the two tables at the database level. The relationship between the `User` and `Address` classes is defined separately using the `relation()` function, which defines an attribute `user` to be placed on the `Address` class, as well as an `addresses` collection to be placed on the `User` class. Such a relation is known as a **bidirectional** relationship. Because of the placement of the foreign key, from `Address` to `User` it is **many to one**, and from `User` to `Address` it is **one to many**. SQLAlchemy is automatically aware of many-to-one/one-to-many based on foreign keys.

The `relation()` function is extremely flexible, and could just have easily been defined on the `User` class:

```
class User(Base):
    # ....
    addresses = relation(Address, order_by=Address.id, backref="user")
```

We are also free to not define a backref, and to define the `relation()` only on one class and not the other. It is also possible to define two separate `relation()` constructs for either direction, which is generally safe for many-to-one and one-to-many relations, but not for many-to-many relations.

When using the `declarative` extension, `relation()` gives us the option to use strings for most arguments that concern the target class, in the case that the target class has not yet been defined. This **only** works in conjunction with `declarative`:

```
class User(Base):
    ....
    addresses = relation("Address", order_by="Address.id", backref="user")
```

When `declarative` is not in use, you typically define your `mapper()` well after the target classes and `Table` objects have been defined, so string expressions are not needed.

We'll need to create the `addresses` table in the database, so we will issue another CREATE from our metadata, which will skip over tables which have already been created:

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

Working with Related Objects

Now when we create a `User`, a blank `addresses` collection will be present. Various collection types, such as sets and dictionaries, are possible here (see *Alternate Collection Implementations* for details), but by default, the collection is a Python list.

```
>>> jack = User('jack', 'Jack Bean', 'gjffdd')
>>> jack.addresses
[]
```

We are free to add `Address` objects on our `User` object. In this case we just assign a full list directly:

```
>>> jack.addresses = [Address(email_address='jack@google.com'), Address(email_address='j25@yah
```

When using a bidirectional relationship, elements added in one direction automatically become visible in the other direction. This is the basic behavior of the **backref** keyword, which maintains the relationship purely in memory, without using any SQL:

```
>>> jack.addresses[1]
<Address('j25@yahoo.com')>

>>> jack.addresses[1].user
<User('jack','Jack Bean', 'gjffdd')>
```

Let's add and commit `Jack Bean` to the database. `jack` as well as the two `Address` members in his `addresses` collection are both added to the session at once, using a process known as **cascading**:

```
>>> session.add(jack)
>>> session.commit()
```
SQL

Querying for Jack, we get just Jack back. No SQL is yet issued for Jack's addresses:

```
>>> jack = session.query(User).filter_by(name='jack').one()
>>> jack
<User('jack','Jack Bean', 'gjffdd')>
```
SQL

Let's look at the `addresses` collection. Watch the SQL:

```
>>> jack.addresses
[<Address('jack@google.com')>, <Address('j25@yahoo.com')>]
```
SQL

When we accessed the `addresses` collection, SQL was suddenly issued. This is an example of a **lazy loading relation**. The `addresses` collection is now loaded and behaves just like an ordinary list.

If you want to reduce the number of queries (dramatically, in many cases), we can apply an **eager load** to the query operation. With the same query, we may apply an **option** to the query, indicating that we'd like `addresses` to load "eagerly". SQLAlchemy then constructs an outer join between the `users` and `addresses` tables, and loads them at once, populating the `addresses` collection on each `User` object if it's not already populated:

```
>>> from sqlalchemy.orm import eagerload

>>> jack = session.query(User).options(eagerload('addresses')).filter_by(name='jack').one()
>>> jack
<User('jack','Jack Bean', 'gjffdd')>

>>> jack.addresses
[<Address('jack@google.com')>, <Address('j25@yahoo.com')>]
```
SQL

SQLAlchemy has the ability to control exactly which attributes and how many levels deep should be joined together in a single SQL query. More information on this feature is available in *Relation Configuration*.

Querying with Joins

While the eager load created a JOIN specifically to populate a collection, we can also work explicitly with joins in many ways. For example, to construct a simple inner join between `User` and `Address`, we can just `filter()` their related columns together. Below we load the `User` and `Address` entities at once using this method:

```
>>> for u, a in session.query(User, Address).filter(User.id==Address.user_id).\
...         filter(Address.email_address=='jack@google.com').all():
...     print u, a
<User('jack','Jack Bean', 'gjffdd')> <Address('jack@google.com')>
```
SQL

Or we can make a real JOIN construct; one way to do so is to use the ORM `join()` function, and tell `Query` to "select from" this join:

```
>>> from sqlalchemy.orm import join
>>> session.query(User).select_from(join(User, Address)).\
...         filter(Address.email_address=='jack@google.com').all()
[<User('jack','Jack Bean', 'gjffdd')>]
```

SQL

`join()` knows how to join between `User` and `Address` because there's only one foreign key between them. If there were no foreign keys, or several, `join()` would require a third argument indicating the ON clause of the join, in one of the following forms:

```
join(User, Address, User.id==Address.user_id)  # explicit condition
join(User, Address, User.addresses)            # specify relation from left to right
join(User, Address, 'addresses')               # same, using a string
```

The functionality of `join()` is also available generatively from `Query` itself using `Query.join`. This is most easily used with just the "ON" clause portion of the join, such as:

```
>>> session.query(User).join(User.addresses).\
...     filter(Address.email_address=='jack@google.com').all()
[<User('jack','Jack Bean', 'gjffdd')>]
```

SQL

To explicitly specify the target of the join, use tuples to form an argument list similar to the standalone join. This becomes more important when using aliases and similar constructs:

```
session.query(User).join((Address, User.addresses))
```

Multiple joins can be created by passing a list of arguments:

```
session.query(Foo).join(Foo.bars, Bar.bats, (Bat, 'widgets'))
```

The above would produce SQL something like `foo JOIN bars ON <onclause> JOIN bats ON <onclause> JOIN widgets ON <onclause>`.

Using Aliases

When querying across multiple tables, if the same table needs to be referenced more than once, SQL typically requires that the table be *aliased* with another name, so that it can be distinguished against other occurrences of that table. The `Query` supports this most explicitly using the `aliased` construct. Below we join to the `Address` entity twice, to locate a user who has two distinct email addresses at the same time:

```
>>> from sqlalchemy.orm import aliased
>>> adalias1 = aliased(Address)
>>> adalias2 = aliased(Address)
>>> for username, email1, email2 in \                                    SQL
...     session.query(User.name, adalias1.email_address, adalias2.email_address).\
...     join((adalias1, User.addresses), (adalias2, User.addresses)).\
...     filter(adalias1.email_address=='jack@google.com').\
...     filter(adalias2.email_address=='j25@yahoo.com'):
...     print username, email1, email2
jack jack@google.com j25@yahoo.com
```

Using Subqueries

The `Query` is suitable for generating statements which can be used as subqueries. Suppose we wanted to load `User` objects along with a count of how many `Address` records each user has. The best way to generate SQL like this is to get the count of addresses grouped by user ids, and JOIN to the parent. In this case we use a LEFT OUTER JOIN so that we get rows back for those users who don't have any addresses, e.g.:

```
SELECT users.*, adr_count.address_count FROM users LEFT OUTER JOIN
    (SELECT user_id, count(*) AS address_count FROM addresses GROUP BY user_id) AS adr_coun
    ON users.id=adr_count.user_id
```

Using the `Query`, we build a statement like this from the inside out. The `statement` accessor returns a SQL expression representing the statement generated by a particular `Query` - this is an instance of a `select()` construct, which are described in *SQL Expression Language Tutorial*:

```
>>> from sqlalchemy.sql import func
>>> stmt = session.query(Address.user_id, func.count('*').label('address_count')).group_by(
```

The `func` keyword generates SQL functions, and the `subquery()` method on `Query` produces a SQL expression construct representing a SELECT statement embedded within an alias (it's actually shorthand for `query.statement.alias()`).

Once we have our statement, it behaves like a `Table` construct, such as the one we created for `users` at the start of this tutorial. The columns on the statement are accessible through an attribute called `c`:

```
>>> for u, count in session.query(User, stmt.c.address_count).\                SQL
...     outerjoin((stmt, User.id==stmt.c.user_id)).order_by(User.id):
...     print u, count
<User('ed','Ed Jones', 'f8s7ccs')> None
<User('wendy','Wendy Williams', 'foobar')> None
<User('mary','Mary Contrary', 'xxg527')> None
<User('fred','Fred Flinstone', 'blah')> None
<User('jack','Jack Bean', 'gjffdd')> 2
```

Selecting Entities from Subqueries

Above, we just selected a result that included a column from a subquery. What if we wanted our subquery to map to an entity ? For this we use `aliased()` to associate an "alias" of a mapped class to a subquery:

```
>>> stmt = session.query(Address).filter(Address.email_address != 'j25@yahoo.com').subquery()
>>> adalias = aliased(Address, stmt)
>>> for user, address in session.query(User, adalias).join((adalias, User.addresses)):
...     print user, address
<User('jack','Jack Bean', 'gjffdd')> <Address('jack@google.com')>
```

SQL

Using EXISTS

The EXISTS keyword in SQL is a boolean operator which returns True if the given expression contains any rows. It may be used in many scenarios in place of joins, and is also useful for locating rows which do not have a corresponding row in a related table.

There is an explicit EXISTS construct, which looks like this:

```
>>> from sqlalchemy.sql import exists
>>> stmt = exists().where(Address.user_id==User.id)
>>> for name, in session.query(User.name).filter(stmt):
...     print name
jack
```

SQL

The `Query` features several operators which make usage of EXISTS automatically. Above, the statement can be expressed along the `User.addresses` relation using `any()`:

```
>>> for name, in session.query(User.name).filter(User.addresses.any()):
...     print name
jack
```

SQL

`any()` takes criterion as well, to limit the rows matched:

```
>>> for name, in session.query(User.name).\
...     filter(User.addresses.any(Address.email_address.like('%google%'))):
...     print name
jack
```

SQL

`has()` is the same operator as `any()` for many-to-one relations (note the `~` operator here too, which means "NOT"):

```
>>> session.query(Address).filter(~Address.user.has(User.name=='jack')).all()
[]
```

SQL

Common Relation Operators

Here's all the operators which build on relations:

- equals (used for many-to-one):

```
query.filter(Address.user == someuser)
```

- not equals (used for many-to-one):

```
query.filter(Address.user != someuser)
```

- IS NULL (used for many-to-one):

```
query.filter(Address.user == None)
```

- contains (used for one-to-many and many-to-many collections):

```
query.filter(User.addresses.contains(someaddress))
```

- any (used for one-to-many and many-to-many collections):

```
query.filter(User.addresses.any(Address.email_address == 'bar'))

# also takes keyword arguments:
query.filter(User.addresses.any(email_address='bar'))
```

- has (used for many-to-one):

```
query.filter(Address.user.has(name='ed'))
```

- with_parent (used for any relation):

```
session.query(Address).with_parent(someuser, 'addresses')
```

Deleting

Let's try to delete `jack` and see how that goes. We'll mark as deleted in the session, then we'll issue a `count` query to see that no rows remain:

```
>>> session.delete(jack)
>>> session.query(User).filter_by(name='jack').count()
0
```

SQL

So far, so good. How about Jack's `Address` objects ?

```
>>> session.query(Address).filter(
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
...  ).count()
2
```

SQL

Uh oh, they're still there ! Analyzing the flush SQL, we can see that the `user_id` column of each address was set to NULL, but the rows weren't deleted. SQLAlchemy doesn't assume that deletes cascade, you have to tell it to do so.

Configuring delete/delete-orphan Cascade

We will configure **cascade** options on the `User.addresses` relation to change the behavior. While SQLAlchemy allows you to add new attributes and relations to mappings at any point in time, in this case the existing relation needs to be removed, so we need to tear down the mappings completely and start again. This is not a typical operation and is here just for illustrative purposes.

Removing all ORM state is as follows:

```
>>> session.close()  # roll back and close the transaction
>>> from sqlalchemy.orm import clear_mappers
>>> clear_mappers() # clear mappers
```

Below, we use `mapper()` to reconfigure an ORM mapping for `User` and `Address`, on our existing but currently un-mapped classes. The `User.addresses` relation now has `delete, delete-orphan` cascade on it, which indicates that DELETE operations will cascade to attached `Address` objects as well as `Address` objects which are removed from their parent:

```
>>> mapper(User, users_table, properties={
...     'addresses':relation(Address, backref='user', cascade="all, delete, delete-orphan")
... })
<Mapper at 0x...; User>

>>> addresses_table = Address.__table__
>>> mapper(Address, addresses_table)
<Mapper at 0x...; Address>
```

Now when we load Jack (below using `get()`, which loads by primary key), removing an address from his `addresses` collection will result in that `Address` being deleted:

```
# load Jack by primary key
>>> jack = session.query(User).get(5)                                    SQL
# remove one Address (lazy load fires off)
>>> del jack.addresses[1]                                               SQL
# only one address remains
>>> session.query(Address).filter(                                      SQL
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
1
```

Deleting Jack will delete both Jack and his remaining `Address`:

```
>>> session.delete(jack)

>>> session.query(User).filter_by(name='jack').count()                    SQL
0

>>> session.query(Address).filter(                                        SQL
...    Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
0
```

Building a Many To Many Relation

We're moving into the bonus round here, but lets show off a many-to-many relationship. We'll sneak in some other features too, just to take a tour. We'll make our application a blog application, where users can write `BlogPost` items, which have `Keyword` items associated with them.

The declarative setup is as follows:

```
>>> from sqlalchemy import Text

>>> # association table
>>> post_keywords = Table('post_keywords', metadata,
...     Column('post_id', Integer, ForeignKey('posts.id')),
...     Column('keyword_id', Integer, ForeignKey('keywords.id'))
... )

>>> class BlogPost(Base):
...     __tablename__ = 'posts'
...
...     id = Column(Integer, primary_key=True)
...     user_id = Column(Integer, ForeignKey('users.id'))
...     headline = Column(String(255), nullable=False)
...     body = Column(Text)
...
...     # many to many BlogPost<->Keyword
...     keywords = relation('Keyword', secondary=post_keywords, backref='posts')
...
...     def __init__(self, headline, body, author):
...         self.author = author
...         self.headline = headline
...         self.body = body
...
...     def __repr__(self):
...         return "BlogPost(%r, %r, %r)" % (self.headline, self.body, self.author)

>>> class Keyword(Base):
...     __tablename__ = 'keywords'
...
...     id = Column(Integer, primary_key=True)
...     keyword = Column(String(50), nullable=False, unique=True)
...
...     def __init__(self, keyword):
...         self.keyword = keyword
```

Above, the many-to-many relation is `BlogPost.keywords`. The defining feature of a many-to-many relation is the `secondary` keyword argument which references a `Table` object representing the

association table. This table only contains columns which reference the two sides of the relation; if it has *any* other columns, such as its own primary key, or foreign keys to other tables, SQLAlchemy requires a different usage pattern called the "association object", described at *Association Object*.

The many-to-many relation is also bi-directional using the `backref` keyword. This is the one case where usage of `backref` is generally required, since if a separate `posts` relation were added to the `Keyword` entity, both relations would independently add and remove rows from the `post_keywords` table and produce conflicts.

We would also like our `BlogPost` class to have an `author` field. We will add this as another bidirectional relationship, except one issue we'll have is that a single user might have lots of blog posts. When we access `User.posts`, we'd like to be able to filter results further so as not to load the entire collection. For this we use a setting accepted by `relation()` called `lazy='dynamic'`, which configures an alternate **loader strategy** on the attribute. To use it on the "reverse" side of a `relation()`, we use the `backref()` function:

```
>>> from sqlalchemy.orm import backref
>>> # "dynamic" loading relation to User
>>> BlogPost.author = relation(User, backref=backref('posts', lazy='dynamic'))
```

Create new tables:

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

Usage is not too different from what we've been doing. Let's give Wendy some blog posts:

```
>>> wendy = session.query(User).filter_by(name='wendy').one()                      SQL
>>> post = BlogPost("Wendy's Blog Post", "This is a test", wendy)
>>> session.add(post)
```

We're storing keywords uniquely in the database, but we know that we don't have any yet, so we can just create them:

```
>>> post.keywords.append(Keyword('wendy'))
>>> post.keywords.append(Keyword('firstpost'))
```

We can now look up all blog posts with the keyword 'firstpost'. We'll use the `any` operator to locate "blog posts where any of its keywords has the keyword string 'firstpost'":

```
>>> session.query(BlogPost).filter(BlogPost.keywords.any(keyword='firstpost')).all()   SQL
[BlogPost("Wendy's Blog Post", 'This is a test', <User('wendy','Wendy Williams', 'foobar')>)]
```

If we want to look up just Wendy's posts, we can tell the query to narrow down to her as a parent:

```
>>> session.query(BlogPost).filter(BlogPost.author==wendy).\                SQL
... filter(BlogPost.keywords.any(keyword='firstpost')).all()
[BlogPost("Wendy's Blog Post", 'This is a test', <User('wendy','Wendy Williams', 'foobar')>)]
```

Or we can use Wendy's own `posts` relation, which is a "dynamic" relation, to query straight from there:

```
>>> wendy.posts.filter(BlogPost.keywords.any(keyword='firstpost')).all()    SQL
[BlogPost("Wendy's Blog Post", 'This is a test', <User('wendy','Wendy Williams', 'foobar')>)]
```

Further Reference

Query Reference: *Querying*

Further information on mapping setups are in *Mapper Configuration*.

Further information on working with Sessions: *Using the Session*.