SQLAlchemy and You

written on Tuesday, July 19, 2011

Without doubt are most new Python web programmers these days chosing the Django framework as their gateway drug to Python web development. As such many people's first experience with a Python ORM (or maybe an ORM altogether) is the Django one. When they are later switching to something else they often find SQLAlchemy unnecessarily complex and hard to use. Why is that the case?

I made a quick poll on Twitter about why people prefer the Django ORM over SQLAlchemy and I got back a few interesting results. First of all that question was obviously asked with the intent to attract answers from people that do prefer Django over SQLAlchemy or at least have some issues with SQLAlchemy that they don't seem to have with Django. Without a doubt there is a large fanbase behind SQLAlchemy, myself included.

SQLAlchemy in general just has a much larger featureset and it's the only ORM for Python which allows you to take full advantage of your database and does not stand in your way. It exposes all features of your underlying database if you want and can be heavily fine tuned.

This article assumes that you have some basic Django knowledge and want to give SQLAlchemy a try. Step by step it walks through the differences and common idioms.

Design Differences

There are two very important differences between SQLAlchemy and Django. The first one is the less obvious one: SQLAlchemy is a deeply layered system, whereas Django's ORM is basically just one layer which is the ORM you see. In SQLAlchemy you have at the very bottom the engine which with abstracts away connection pools and basic API differences between different databases, on top of that you have the SQL abstraction language, sitting on top of that and the table definitions you have the basic ORM and on top of that you have the declarative ORM which looks very closely to the Django ORM. The other more striking difference however is that SQLAlchemy follows the “Unit of Work” pattern whereas Django's ORM follows something that is very close to the “Active Record” pattern.

What's the difference? Django's ORM is basically quite simple. Each time you do any query it generates a SQL expression for you and sends a query to the database. Then it constructs and object for you. That object can be modified and if you call `save()` on it, it will update the record in the database with the new values of the attributes. This is not at all how SQLAlchemy's ORM component works. In SQLAlchemy you have an object called the “session”. It basically encapsulates a transaction. However it does more. Each object is tracked by primary key in this session. As such each object only exists once by primary key. As such you can safe a lot of

queries and you never have things out of sync. When you commit the session it will send all changes at once to the database in correct order, if you rollback the session nothing happens instead.

SQLAlchemy's Complexity

SQLAlchemy has to fight with some basic acceptance problems which are caused by the fact that it's framework independent and is not even something you would only use in web applications. This is why projects like Flask-SQLAlchemy exist to make the integration for you. Many frameworks either provide something that preconfigures SQLAlchemy for you with some sane defaults or have a section in the cookbook to copy/paste code from.

This also is the reason why many people find SQLAlchemy's documentation overwhelming. Not only does the documentation guide you through the different levels of SQLAlchemy but also shows all the different ways you can configure SQLAlchemy and the ORM.

Djang on the other hand does not have many different ways the configure the ORM. It comes preconfigured with the exception of some database related setting such as server name, port and a few other things.

The Session — The Heart of the ORM

If you chose to use SQLAlchemy's ORM component and not just the engine or SQL abstraction layer you will sooner or later be confronted with the session object. What is the session object? It is the one object that records all the changes on models you do. How does a model know about the session? Let's compare this to Django for a moment.

In Django if you have a model it also has an *objects* attribute attached. That attribute points to a manager object which in turn can generate queryset objects and the queryset objects then fire the query and hold the results. How does this queryset object find the current transaction? In Django the answer is that transactions are bound to a thread always. Each thread can have one transaction and the queryset uses. So Django needs to find the transaction in the queryset when the actual query fires and it does that by finding the current thread which owns a transaction.

So how does that work in SQLAlchemy? As we have established, objects are always “owned” by a session and keyed by primary key. Each primary key can only exist once. Because that session is quite fundamental and needs to work in many setups this is configurable. But first we need to figure out what the difference between a Django queryset and a SQLAlchemy query object is. They look similar on the surface, but are very different in practice. A Django queryset is created by the manager of a model and also holds the results once the query is fired. In SQLAlchemy the query object can be created in many different ways and unifies the idea of the manager and Django's querysets in itself. When the query object is evaluated (the query is sent to the database) it instead returns a list or the only result object, depending on what methods are used. This means the manager object is entirely unnecessary in SQLAlchemy, if you need custom manager methods you would just subclass the query and attach new methods on there. Since the query object can be joined it makes for quite a nice API.

But back to sessions. In Django default of having one transaction per thread makes a lot of sense, but limits the usefulness when you have other means of concurrency somewhat. Also

it makes it hard if you want to use the same model against two different databases. This is where the explicit API comes into play which is the one that the documentation uses.

So instead of this Django idiom:

```
MyModel.objects.all()
```

In default SQLAlchemy you would do this instead:

```
session.query(MyModel).all()
```

Here it's explicit what session object should be used and you can have multiple of those side by side obviously. Since many people do not need this and are fine with having one session per thread you can take advantage of the scoped session support in SQLAlchemy. For instance the Flask-SQLAlchemy extension will by default attach a *query* class level attribute to your models which looks at the current thread and it's session object. So each thread will only have one session. Furthermore at the end of an HTTP request in Flask the extension will automatically destroy the session and discarding uncommitted changes.

With that, it looks a lot closer to Django:

```
MyModel.query.all()
```

You however will still need the session to commit and insert and delete objects from the database. The scoped session automatically provides a proxy that always point to the current active session.

The Declarative Extension

For a long time SQLAlchemy made you declare table objects first and then separately create the classes and map those together. This has the advantage over just subclassing some magical baseclass that you can map already existing classes to things in the database. The downside always was however that you had to declare multiple objects and the common case was unnecessarily complex.

SQLAlchemy since introduced the declarative base. It's a extension module shipped with SQLAlchemy that provides a function which creates a brand new baseclass (which you can also customize) which does metaclass magic very similar to Django. As such you can directly declare relationships and attributes in the class itself.

There are still some differences though:

- Relationships are not magically created for you, you have to be expicit. The same is true with foreign keys.
- Primary keys are not automatically generated for you for the simple reason that SQLAlchemy supports more than one primary key type. If you want one chosen by default, you can provide a baseclass that implements that.
- The table name has to be set explicitly. Again you can customize the baseclass to derive the table name from the class name if you like.

To get this baseclass you basically just need this:

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

Basic Models

A basic Django model looks something like this:

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

The equivalent SQLAlchemy model with declarative base looks like this:

```
from sqlalchemy import Column, Integer, String

class Person(Base):
    __tablename__ = 'persons'
    id = Column(Integer, primary_key=True)
    first_name = Column(String(30))
    last_name = Column(String(30))
```

It's a little more to type, but if you want to make this implicit you just need a proper baseclass. Flask-SQLAlchemy for instance sets the lowercase version of the class as default tablename unless overridden.

Many-to-One Relationships

In Django this is straightforward:

```
class Manufacturer(models.Model):
    name = models.CharField(max_length=30)

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer,
                                     related_name='cars')
    name = models.CharField(max_length=30)
```

In SQLAlchemy we have to be a little bit more expressive:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref

class Manufacturer(Base):
    __tablename__ = 'manufacturers'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))

class Car(models.Model):
    __tablename__ = 'cars'
    id = Column(Integer, primary_key=True)
    manufacturer_id = Column(Integer, ForeignKey('manufacturers.id'))
    name = Column(String(30))

    manufacturer = relationship('Manufacturer', backref=
        backref('cars', lazy='dynamic'))
```

Here we have to model the relationship ourself. First we need to declare the foreign key. It has to have the same type as the primary key of the table we want to point to and additionally the column needs to be given a *ForeignKey* instance with the first argument being the dotted name to the column referenced. Note that this is the table name, not the class name.

The relationship is then declared on *Car* with *relationship*. The first argument is a class or the

name of a class we want to have the relationship with. By default it will try to find a valid join condition automatically. If it does not, you can explicitly provide one as a string or real expression:

```
manufacturer = relationship('Manufacturer',
    primaryjoin='Car.manufacturer_id == Manufacturer.id',
    backref=backref('cars', lazy='dynamic'))
```

The *backref* argument automatically declares the reverse. It will attach a *cars* property on the manufacturer. The `lazy='dynamic'` tells SQLAlchemy to make the backref lazy and a dynamic loading one. In that case accessing *manufacturer.cars* will be a query object you can further refine instead of directly firing the query and returning a list.

Other lazy settings:

- `'select'`: if accessed load everything as list with another select statement. This is the default.
- `'joined'`: uses a join to automatically load that backref with the query of the parent itself.
- `'dynamic'`: returns a query object instead of firing the query. This can be sliced and further extended.

The lazy settings can also be set on *relationship* and not just backref.

Backref in a nutshell:

`'lazy'` and `'select'`. The first one fires a query when *honda.cars* is accessed, the other one will fetch it when honda is queried:

```
>>> honda.cars
[<Car 1>, <Car 2>]
```

And here with `'dynamic'`:

```
>>> honda.cars
<AppenderQuery ...>
>>> honda.cars.all()
[<Car 1>, <Car 2>]
```

Many-To-Many

Many to many relationships in Django are easy cake because everything is done for you:

```
class Topping(models.Model):
    name = models.CharField(max_length=30)

class Pizza(models.Model):
    toppings = models.ManyToManyField(Topping)
    name = models.CharField(max_length=30)
```

In SQLAlchemy we have to construct a helper table to join over:

```
from sqlalchemy import Column, Integer, String, ForeignKey, Table
from sqlalchemy.orm import relationship, backref

pizza_toppings = Table('pizza_toppings', Base.metadata,
    Column('topping_id', Integer, ForeignKey('toppings.id')),
    Column('pizza_id', Integer, ForeignKey('pizzas.id'))
)
```

```
class Topping(Base):
    __tablename__ = 'toppings'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))

class Pizza(models.Model):
    __tablename__ = 'pizzas'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))

    toppings = relationship('Topping', secondary=pizza_toppings,
                            backref=backref('pizzas', lazy='dynamic'))
```

Translating Queries From Django To SQLAlchemy

So this here assumes that you are using scoped sessions like Flask-SQLAlchemy does and unmodified Django. The first example is always how the equivalent Django code looks like and how you would do that with SQLAlchemy:

Inserting Entries

Inserting entries in Django can be done with either creating an instance of a model or by using the `create()` method of the object manager:

```
foo = MyModel(field1='value', field2='value')
foo.save()

# or alternatively
foo = MyModel.objects.create(field1='value', field2='value')
```

In SQLAlchemy you need to do this instead:

```
foo = MyModel(field1='value', field2='value')
session.add(foo)
```

But with that you have only added the object to the session, at that point it has not yet committed the transaction. This has to be done explicitly by yourself when you are happy with all the changes:

```
session.commit()
```

Deleting Entries

Deleting works very much like saving in Django. You get your object and then call the `delete()` method on it:

```
obj = MyModel.objects.filter(pk=the_id).get()
obj.delete()
```

In SQLAlchemy that operation is performed via the session:

```
obj = MyModel.query.get(the_id)
session.delete(obj)
```

Again, remember to commit your session.

Updating Entries

How do you update an entry? Just get the object, modify it and commit the session:

```
obj = MyModel.query.get(the_id)
obj.name = 'New Value'
session.commit()
```

Primary Key Queries

Queries is where Django and SQLAlchemy are the most different. Django uses keyword arguments to the query functions to filter the query, SQLAlchemy generally uses expressions composed out operator objects.

Query by primary key in Django:

```
obj = MyModel.query.filter(pk=the_id).get()
```

And in SQLAlchemy:

```
obj = MyModel.query.get(the_id)
```

Note that `get()` returns *None* if the primary key does not exist in SQLAlchemy and will raise a *DoesNotExist* exception in Django.

Generally the `get()` method is a shortcut in SQLAlchemy that will also not issue a query for that object if it was already queried for that session before. Also unlike Django your primary key can be of any type or be a compound of more than one column.

General Query Syntax

If you want to filter a query in Django you generally use keyword arguments in the format `column__operation=value`. For instance `column__contains='e'` to check if a string column named *column* contains the letter “e”. In SQLAlchemy instead you are using expressions. These expressions can be printed to see what query they would generate.

Here some examples:

```
>>> print MyModel.id == 23
model.model_id = :model_id_1
>>> print MyModel.id.in_([1, 2, 3])
model.model_id IN (:model_id_1, :model_id_2, :model_id_3)
>>> print MyModel.name.contains('e')
model.name LIKE '%%' || :name_1 || '%%'
```

Note that SQLAlchemy shows you the placeholders there because it will let the database insert those values later.

The whole expression language expresses pretty much everything that SQL has to offer:

```
>>> print MyModel.thread_count + MyModel.post_count + 1
(model.thread_count + model.post_count) + :param_1
>>> print MyModel.id.between(1, 10) & MyModel.name.startswith('a')
model.model_id BETWEEN :model_id_1 AND :model_id_2 AND
    model.name LIKE :name_1 || '%%'
```

Now this is a biggie, because this is how you can filter for anything if you pass such an expression to `filter()`:

```
active_users_with_a_or_b = User.query.filter(
    (User.name.startswith('a') | User.name.startswith('b')) &
    User.is_active == True
).all()
```

To evaluate a query you have a few choices:

1. `first()` returns the first result from the query and will also tell the database to perform an implicit `LIMIT 1`. If more than one result is found you won't know and if none is found you get *None* back.
2. `one()` is similar to `first()` but it will not limit the result in any way but perform a sanity check on getting the results. It will raise an *NoResultFound* exception back if it did not found a single row or a *MultipleResultsFound* exception if it got more than one result which indicates a bug on your part.
3. `all()` just evaluates the whole query and returns each row as a list. Why as a list and not as an iterator? First of all because each object returned is also immediately registered on the session. There are of course ways to bypass that, but unless you have an enormous result count you won't notice, secondly because most Python database adapters don't support streaming results anyways.

Now this is nice and everything, but all that model repetition can be annoying. For as long as you are just comparing a column to a given value you can use the `filter_by()` function and pass keyword arguments:

```
user = User.query.filter_by(username=username).first()
```

Multiple arguments are automatically joined with `AND`.

Date Based Queries

In Django you can use `field__year=2011` to select all entries where the year of a field has a specific value. Underneath what usually happens is that an *EXTRACT* expression is issued. Unfortunately that's hugely database dependent and does not map nicely to a function. Thankfully SQLAlchemy provides a helper for that which automatically does the right thing for each database:

```
from sqlalchemy.sql import extract

entries_a_month = Entry.query.filter(
    (extract(Entry.pub_date, 'year') == 2011) &
    (extract(Entry.pub_date, 'month') == 1)
).all()
```

Quite a few extractions are possible. The most common ones are `month`, `day`, `year`, `hour`, `minute`, `second`, `doy` (day of year) and `dow` (day of week).

Sorting

In Django if you sort something you do that by calling `order_by()` and passing it some strings with the columns to order by:

```
forwards = MyModel.objects.order_by('pub_date')
```

```
backwards = MyModel.objects.order_by('-pub_date')
```

While it appears that the same is possible in SQLAlchemy you have to be careful because it only works as SQLAlchemy inserts that text directly into the query. What instead you want to be doing is using the expressions again:

```
forwards = MyModel.query.order_by(MyModel.pub_date)
backwards = MyModel.query.order_by(MyModel.pub_date.desc())
```

And again, any expression works in that situation, so you can just easily order by ridiculous expressions if you want.

Aggregates

Aggregates in Django are a quite new feature and generally not all that awesome, so we're skipping the Django part here. Thankfully they are much better supported in SQLAlchemy as SQLAlchemy just handles them by querying over arbitrary expressions. Functions on the database can be expressed by `sqlalchemy.func.functionname` in SQLAlchemy. This in combination with arbitrary expressions makes it quite potent. But first the simple case:

```
from sqlalchemy.sql import func

q = session.query(func.count(User.id))
```

Now that query obviously does not resolve to a model but a scalar value. In this case if we would call `q.first()` we would get a single tuple back with a single item: the count. For this case SQLAlchemy provides a nice shortcut: `scalar()`:

```
>>> session.query(func.count(User.id)).scalar()
1337
```

What if we want to group by something? Use `group_by()` and just iterate over it:

```
for age, count in session.query(User.age,
        func.count(User.id)).group_by(User.age).all():
    print 'Users aged %d: %d' % (age, count)
```

Distinct counts are simple as well, just call `.distinct()` on the query. In fact: if you have a rough idea of what the SQL would look like you can get to the expected result with pure guesswork and SQLAlchemy will most likely “just work” ™.

Joins

Now this is the part where people get constantly confused with SQLAlchemy but fear not, I have you covered. Django hides the business of joins from you. For instance if you want to get all posts written by a specific author that is known by name you would do something like this:

```
posts = Post.objects.filter(author__name__exact=the_author_name)
```

So how do you do that in SQLAlchemy? The answer is that this means a join is taking place. There are two ways to model that select. First the simple one:

```
posts = Post.query.join(Author).filter(Author.name == the_author_name)
```

That wasn't too tricky. How does SQLAlchemy know how to do the right thing? It looks at what joins are possible and if only one is, it selects the right one. Alternatively you can explicitly

provide what to join on as an expression as second argument to `join()`. Again, you can get arbitrarily complex there. Everything after the join automatically operates on the last `.join()`-ed model. If you want to further filter the former model (here *Post*) you can either move them before the `.join()` call or use `.reset_joinpoint()`.

Alternatively you could also express this as a subselect:

```
author_query = Author.query.filter(Author.name == the_author_name)
posts = Post.query.filter(Post.author_id.in_(author_query))
```

Why does SQLAlchemy not do what Django does? Well, first of all explicit is better than implicit: you know exactly what happens. A regular join is not always what you want or SQL would not provide an outerjoin which of course you can use with SQLAlchemy as well. Secondly, it's really easy to replicated. If you are curious of how that works you can have a look at this subclass of the builtin query that implements Django's filtering with keyword arguments: sqlalchemy-django-query.

Why Consider SQLAlchemy?

This article did not really give you any reasons to use SQLAlchemy, did it? But the simple cases is not where SQLAlchemy shines. It's the more complex situations which you can't do at all in Django that work nicely in SQLAlchemy. Oh, and SQLAlchemy does not override all your columns when you just changed one on update ;-)

This entry was tagged python and sqlalchemy

Contact me via mail, twitter, github or bitbucket. (feed)