SQLAlchemy: an Architectural Retrospective
Front Matter

• This talk is loosely based on the SQLAlchemy chapter I'm writing for *The Architecture of Open Source Applications*

Introduction

• SQLAlchemy, the Database Toolkit for Python
• Introduced in 2005
• End-to-end system for working with the Python DBAPI
• Got early attention fast: fluent SQL, ORM with Unit of Work pattern
Part I - Philosophy
"Abstraction"

- When we talk about relational database tools, the term "database abstraction layer" is often used.

- What is implied by "Abstraction"?
  - Conceal details of how data is stored and queried?
  - Should an abstraction layer conceal even that the database is relational?
  - Should it talk to S3, MongoDB, DBM files just like a SQL database?
  - In this definition, "abstraction" means "hiding".
Problems with "abstraction=hiding"

• SQL language involves "relations" (i.e. tables, views, SELECT statements) that can be sliced into subsets, intersected on attributes (i.e. joins)

• Ability to organize and query for data in a relational way is the primary feature of relational databases.

• Hiding it means you no longer have that capability.

• Why use a relational database then? Many alternatives now.

• We don't want "hiding". We want "automation"!
Automation

• Provide succinct patterns that automate the usage of lower level systems
• Establish single points of behavioral variance
• We still need to do work, know how everything works, design all strategies!
• But we work efficiently, instructing our tools to do the grunt work we give them.
• Use our own schema/design conventions, not someone else's
SQLAlchemy's Approach

• The developer must consider the relational form of the target data.

• Query and schema design decisions are all made by the developer. Tools don't make decisions.

• Provide a rich, detailed vocabulary to express these decisions

• Developer creates patterns and conventions based on this vocabulary.

• Opposite to the approach of providing defaults + ways to override some of them
An out of the box mapping

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    username = Column(String(50), nullable=False)
    addresses = relationship("Address", backref="user",
                            cascade="all, delete-orphan")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'),
                     nullable=False)
    street = Column(String(50))
    city = Column(String(50))
    state = Column(CHAR(2))
    zip = Column(String(15))
Dealing with Verbosity

- But what about how verbose that was?
- Is that verbosity a problem when...
- Your whole app has just the two classes? No.
- You have a large app, using those same patterns over and over again - yes.
- Then we're writing a large app. Large apps should have foundations!
Building a Foundation

Use a base that defines the conventions for all tables and classes

```python
from sqlalchemy import Column, Integer
from sqlalchemy.ext.declarative import declarative_base

class Base(object):
    """Define the base conventions for all tables/classes."""

    @declared_attr
    def __tablename__(cls):
        """Table is named after the class name"""
        return cls.__name__.lower()

    id = Column(Integer, primary_key=True)
    """Surrogate primary key column named 'id'"""

Base = declarative_base(cls=Base)
```
Building a Foundation

Use functions to represent common idioms, like foreign key columns, datatypes that are common

```
def fk(tablename, nullable=False):
    """Define a convention for all foreign key columns.
    Just give it the table name.""

    return Column("%s_id" % tablename, Integer,
                  ForeignKey("%s.id" % tablename),
                  nullable=nullable)
```
Building a Foundation

Use prototypes and similar techniques for particular table structures that are common

class AddressPrototype(object):
    """Lots of objects will have an 'address'. Let's build a prototype for it.'""

    street = Column(String(50))
    city = Column(String(50))
    state = Column(CHAR(2))
    zip = Column(String(15))
Building a Foundation

Use mixins to define table/class attributes common to subsets of the domain model

class HasAddresses(object):
    """Define classes that have a collection of addresses via the AddressPrototype foundation."""

    @declared_attr
def addresses(cls):
        cls.Address = type("%sAddress" % cls.__name__,
                         (AddressPrototype, Base),
                         {'%s_id' % cls.__tablename__:fk(cls.__tablename__)})
        return relationship(cls.Address,
                            backref=cls.__name__.lower(),
                            cascade="all, delete-orphan")
Use the Foundation
With our conventions in place, the actual mapping for both user/address looks like this

```python
from myapp.base import (
    HasAddresses, Base, Column, String
)

class User(HasAddresses, Base):
    username = Column(String(50), nullable=False)

Address = User.Address
```
More Foundation

• More examples of convention-oriented "helpers", including pre-fab one_to_many()/many_to_one()/many_to_many() helpers, at http://techspot.zzeek.org/2011/05/17/magic-a-new-orm/
Exposing Relational Constructs

- SQLAlchemy's querying system doesn't try to hide that a relational database is in use.
- Like "power steering" for SQL. Doesn't teach you how to drive!
- Developer should be very aware of the SQL being emitted. SQLAlchemy makes this easy via logging or "echo" flag.
- Just like your car has windows to see where you're going!
Exposing Relational Constructs - An example

Users on a certain street with no address in NYC - a hypothetical "relational-agnostic" way

```python
my_user = User.\\
    filter(addresses__street = '123 Green Street').\\
    has_none(addresses__city = "New York")[0]

# obvious SQL from the above

SELECT * FROM user JOIN address ON user.id=address.user_id
WHERE address.street == '123 Green Street'
AND NOT EXISTS (  
    SELECT * FROM address WHERE city='New York'
    AND user_id=user.id
  )
```
Exposing Relational Constructs - An example

• Now I want:
  • Give me all households in New York with exactly two occupants where neither occupant has any residences outside of the city.

• Our model isn't terrifically optimized for this query, since the "address" rows are not normalized

• This query needs to be built relationally, referencing relational structures explicitly and building from the inside out

• This is why we like relational databases!
Build a query from the inside out

-- New York addresses that have two
-- occupants

SELECT street, city, zip FROM address
  WHERE city='New York'
GROUP BY street, city, zip
HAVING count(user_id) = 2
Build a query from the inside out

-- users who are different from each other

SELECT * FROM user AS u_1 JOIN
user AS u_2 ON u_1.id > u_2.id
Build a query from the inside out

-- join them to their addresses, join addresses
-- to the two occupant NY addresses

```
SELECT * FROM user AS u_1 JOIN
    user AS u_2 ON u_1.id > u_2.id JOIN
    address AS a_1 ON u_1.id = a_1.user_id JOIN
    address AS a_2 ON u_2.id = a_2.user_id JOIN
    (SELECT street, city, zip FROM address
     WHERE city='New York'
     GROUP BY street, city, zip
     HAVING count(user_id) = 2
    ) AS two_occupant_ny ON (a_1.street=two_occupant_ny.street AND
                             a_1.city=two_occupant_ny.city AND
                             a_1.zip=two_occupant_ny.zip AND
                             a_2.street=two_occupant_ny.street AND
                             a_2.city=two_occupant_ny.city AND
                             a_2.zip=two_occupant_ny.zip
    )
```
Build a query from the inside out

-- ... who don't have a house outside of New York

SELECT * FROM user AS u_1 JOIN
    user AS u_2 ON u_1.id > u_2.id JOIN
    address AS a_1 ON u_1.id = a_1.user_id JOIN
    address AS a_2 ON u_2.id = a_2.user_id JOIN
    (SELECT street, city, zip FROM address
        WHERE city='New York'
        GROUP BY street, city, zip
        HAVING count(user_id) = 2
    ) AS two_occupant_ny ON (
        a_1.street == two_occupant_ny.street AND
        a_1.city == two_occupant_ny.city AND
        a_1.zip == two_occupant_ny.zip AND
        a_2.street == two_occupant_ny.street AND
        a_2.city == two_occupant_ny.city AND
        a_2.zip == two_occupant_ny.zip
    )
    AND NOT EXISTS (SELECT * FROM address WHERE
        city!='New York' AND
        user_id=u_1.id OR
        user_id=u_2.id)
Build a query from the inside out

• SQLAlchemy gives you this same "inside out" paradigm - you think in terms of SQL relations and joins in the same way as when constructing plain SQL.

• SQLAlchemy can then apply automated enhancements like eager loading, row limiting, further relational transformations
Build a Query() from the inside out

# New York addresses that have two
# occupants

two_occupant_ny = \
    Session.query(Address.street, Address.city, Address.zip).\
    filter(Address.city == 'New York').\
    group_by(Address.street, Address.city, Address.zip).\
    having(func.count(Address.user_id) == 2).\
    subquery()
Build a Query() from the inside out

# users who are different from each other

```
u_1, u_2 = aliased(User), aliased(User)

user_q = Session.query(u_1, u_2).
    select_from(u_1).
    join(u_2, u_1.id > u_2.id)
```
Build a Query() from the inside out

# join them to their addresses, join addresses
# to the two occupant NY addresses

a_1, a_2 = aliased(Address), aliased(Address)
user_q = user_q.\n    join(a_1, u_1.addresses).\n    join(a_2, u_2.addresses).\n    join(two_occupant_ny,
        and_
            (a_1.street==two_occupant_ny.c.street,
             a_1.city==two_occupant_ny.c.city,
             a_1.zip==two_occupant_ny.c.zip,
             a_2.street==two_occupant_ny.c.street,
             a_2.city==two_occupant_ny.c.city,
             a_2.zip==two_occupant_ny.c.zip,
        )
    )
# who don't have a house outside of New York

user_q = user_q.filter(~exists([Address.id]).where(Address.city != 'New York').where(or_(where(Address.user_id==u_1.id, Address.user_id==u_2.id)))
Build a Query() from the inside out

```python
# pre-load all the Address objects for each
# User too !

user_q = user_q.options(
    joinedload(u_1.addresses),
    joinedload(u_2.addresses))

users = user_q.all()
```
What's it look like?

```sql
SELECT user_1.id AS user_1_id, user_1.username AS user_1_username, user_2.id AS user_2_id, user_2.username AS user_2_username, address_1.id AS address_1_id, address_1.street AS address_1_street, address_1.city AS address_1_city, address_1.zip AS address_1_zip, address_1.user_id AS address_1_user_id, address_2.id AS address_2_id, address_2.street AS address_2_street, address_2.city AS address_2_city, address_2.zip AS address_2_zip, address_2.user_id AS address_2_user_id
FROM user AS user_1
    JOIN user AS user_2 ON user_1.id > user_2.id
    JOIN address AS address_3 ON user_1.id = address_3.user_id
    JOIN address AS address_4 ON user_2.id = address_4.user_id
    JOIN (SELECT address.street AS street, address.city AS city, address.zip AS zip
          FROM address
          WHERE address.city = ? GROUP BY address.street, address.city, address.zip
          HAVING count(address.user_id) = ?) AS anon_1
    ON address_3.street = anon_1.street AND address_3.city = anon_1.city AND address_3.zip = anon_1.zip AND address_4.street = anon_1.street AND address_4.city = anon_1.city AND address_4.zip = anon_1.zip
    LEFT OUTER JOIN address AS address_1 ON user_1.id = address_1.user_id
    LEFT OUTER JOIN address AS address_2 ON user_2.id = address_2.user_id
WHERE NOT (EXISTS (SELECT address.id
                     FROM address
                     WHERE address.city != ? AND (address.user_id = user_1.id OR address.user_id = user_2.id)))

--params: ('New York', 2, 'New York')

# result!
User(name=u5, addresses=s1/New York/12345, s2/New York/12345, s3/New York/12345) /
User(name=u2, addresses=s2/New York/12345, s4/New York/12345, s5/New York/12345)
```
"Leaky Abstraction"

- Is this "leaky abstraction"?
- You bet!
- Joel On Software - "All non-trivial abstractions, to some degree, are leaky."
- SQLAlchemy works with this reality up front to create the best balance possible.
- The goal is controlled automation, reduced boilerplate, succinct patterns of usage. **Not** "don't make me understand things".
Part II
Architectural Overview
The Core / ORM Dichotomy

- SQLAlchemy has two distinct areas
- Core
- Object Relational Mapper (ORM)
The Core / ORM Dichotomy

SQLAlchemy ORM

Object Relational Mapper (ORM)

SQLAlchemy Core

Schema / Types

SQL Expression Language

Engine

Connection Pooling

Dialect

DBAPI
Core/ORM Dichotomy - Core

- All DBAPI interaction
- Schema description system (metadata)
- SQL expression system
- Fully usable by itself as the basis for any database-enabled application, other ORMs, etc.
- A Core-oriented application is schema-centric
- Biggest example of a custom Core-only persistence layer is Reddit
Core/ORM Dichotomy - ORM

• Built on top of Core. Knows nothing about DBAPI.

• Maps user-defined classes in terms of table metadata defined with core constructs

• Maintains a local set of in-Python objects linked to an ongoing transaction, passes data back and forth within the transactional scope, using a unit-of-work pattern.

• Data passed uses queries that ultimately are generated and emitted using the Core

• An ORM-oriented application is domain model centric.
Core/ORM Dichotomy - Pros

- ORM is built agnostic of SQL rendering / DBAPI details. All SQL/DBAPI behavior can be maintained and extended without any ORM details being involved.
- Core concepts are exposed within the ORM, allowing one to "drop down" to more SQL/DBAPI-centric usages within the context of ORM usage.
- Early ORM was able to be highly functional, as missing features were still possible via Core usage.
Core/ORM Dichotomy - Cons

- New users need to be aware of separation
- Performance. ORM and Core both have their own object constructs and method calls, leading to a greater number of objects generated, deeper call stacks. CPython is heavily impacted by function calls.
- Pypy hopes to improve this situation - SQLAlchemy is Pypy compatible
- Alex Gaynor hangs on #sqlalchemy-devel and runs our tests against Pypy constantly
Selected Architectural Highlights
Taming the DBAPI

- DBAPI is the pep-249 specification for database interaction.
- Most Python database client libraries conform to the DBAPI specification.
- Lots of "suggestions", "guidelines", areas left open to interpretation
- Unicode, numerics, dates, bound parameter behavior, behavior of execute(), result set behavior, all have wide ranges of inconsistent behaviors.
A rudimentary SQLAlchemy Engine interaction

engine = create_engine(
    "postgresql://user:pw@host/dbname")

connection = engine.connect()

result = connection.execute(
    "select * from user_table where name=?", 
    "jack")

print result.fetchall()
connection.close()
SQLAlchemy's Dialect System

- **Pool**
- **Engine** <<creates>> **Connection** <<creates>> **ResultProxy**
- **Dialect** <<uses>> **DBAPI** <<produces>> **DBAPI cursor**
- **ExecutionContext** <<uses>> **Connection** <<creates>> **ResultProxy**
- **sqlalchemy.engine**
- **psycopg2**
- **psycopg2 DBAPI** <<produces>> **DBAPI connection**
- **sqlalchemy.pool** <<maintains>> **Pool**
- **sqlalchemy.pool**
Dealing with database and DBAPI Variability

Two levels of variance

sqlalchemy.engine

Dialect

DefaultDialect

PGDialect

PGDialect_psycopg2

ExecutionContext

DefaultExecutionContext

PGExecutionContext

PGExecutionContext_psycopg2

sqlalchemy.dialects.postgresql
DBAPIs with Multiple Backends

sqlalchemy.engine

Dialect

DefaultDialect

sqlalchemy.dialects.mssql

MSDialect

MSDialect_pyodbc

sqlalchemy.connectors

PyODBCConnector

sqlalchemy.dialects.mysql

MySQLDialect

MySQLDialect_pyodbc
SQL Expression Constructs

• Wasn't clear in the early days how SQL expressions should be constructed
• Strings ? Hibernate HQL ?
• statement.addWhereClause(isGreaterThanOrEqualTo(x, 5)) ?
• ...
• Ian Bicking's SQLObject has a great idea !! Let's do that !!
We can use Python expressions and overload operators!

```python
from sqlobject.sqlbuilder import EXISTS, Select

select = Test1.select(EXISTS(Select(Test2.q.col2,
    where=(Test1.q.col1 == Test2.q.col2))))
```
Operator Overloading

This expression does not produce True or False

cOLUMN('a') == 2
Operator Overloading

Instead, `__eq__()` is overloaded so it's equivalent to...

column('a') == 2

from sqlalchemy.sql.expression import _BinaryExpression
from sqlalchemy.sql import column, bindparam
from sqlalchemy.operators import eq

_BinaryExpression(
    left=column('a'),
    right=bindparam('a', value=2, unique=True),
    operator=eq
)
Example SQL Expression

Statement:

SELECT id FROM user WHERE name = ?

SQL Expression:

from sqlalchemy import select

stmt = select([user.c.id]).where(user.c.name=='ed')
Example SQL Expression

```
Select
  _froms
  _raw_columns
  _whereclause

_ColumnClause
  name='id'

_TableClause
  columns
  name='user'

_ColumnClause
  name='name'

(BinaryExpression
  operator=eq
  left
  right

_BindParam
  value='ed'
```
Intro to Mapping

In original SQLAlchemy, mappings looked like this: first define "table metadata":

```python
from sqlalchemy import (MetaData, String, Integer, CHAR, Column, Table, ForeignKey)

metadata = MetaData()
user = Table('user', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50), nullable=False)
)

data = Table('data', metadata,
    Column('id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey('user.id'),
           nullable=False),
    Column('street', String(50)),
    Column('city', String(50)),
    Column('state', CHAR(2)),
    Column('zip', String(14))
)```

... then, define classes and "map" them to the tables using the mapper() function:

```python
from sqlalchemy.orm import mapper, relationship

class User(object):
    def __init__(self, name):
        self.name = name

class Address(object):
    def __init__(self, street, city, state, zip):
        self.street = street
        self.city = city
        self.state = state
        self.zip = zip

mapper(User, user, properties={
    'addresses':relationship(Address)
})
mapper(Address, address)
```
This strict separation of database metadata and class definition, linked by the also separate mapper() step, we now call *classical* mapping.
In modern SQLAlchemy, we usually use the Declarative system to "declare" Table metadata and class mapping at the same time...

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

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(50), nullable=False)
    addresses = relationship("Address")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'),
                     nullable=False)

# ...
```
Declarative Mapping

... or at least, the class definition and mapping. Table metadata can still be separate...

class User(Base):
    __table__ = user
    addresses = relationship("Address", backref="user",
                            cascade="all, delete-orphan")

class Address(Base):
    __table__ = address
class User(Base):
    __table__ = Table('user', Base.metadata,
                     Column('id', Integer, primary_key=True),
                     Column('name', String(50), nullable=False))

    addresses = relationship("Address", backref="user",
                           cascade="all, delete-orphan")

class Address(Base):
    __table__ = Table('address', Base.metadata,
                      Column('id', Integer, primary_key=True),
                      Column('user_id', Integer, ForeignKey('user.id'),
                             nullable=False),
                      # ...
                      )
What do classical mapping, declarative mapping, declarative with table etc. all have in common?
They all create a `mapper()` and instrument the class in the identical way!

The `mapper()` Object for the `User` class

```python
>>> from sqlalchemy.orm import class_mapper
>>> class_mapper(User)
<Map at 0x1267970; User>
```
They all create a `mapper()` and instrument the class in the identical way!

Attributes are "instrumented" - when using Declarative, this replaces the `Column` object originally placed in the class for the "username" attribute.

```python
>>> User.username
<sqlalchemy.orm.attributes.InstrumentedAttribute object at 0x1267c50>
```
Anatomy of a Mapping
Fun facts about Declarative

• A "SQLObject"-like declarative layer was always planned, since the beginning. It was delayed so that focus could be placed on classical mappings first.

• An early extension, ActiveMapper, was introduced and later superseded by Elixir - a declarative layer that redefined basic mapping semantics.

• Declarative was introduced as a "one click away from classical mapping" system, which retains standard SQLA constructs - only rearranging how they are combined.

• zzzzeek had to be convinced by Chris Withers to support Declarative "mixins" - thanks Chris!
Unit of Work

• Unit of work's job is to find all pending data changes in a particular Session, and emit them to the database.

• (the Session is an in-memory "holding zone" for the mapped objects we're working with)

• Organizes pending changes into commands which emit batches of INSERT, UPDATE, DELETE statements

• Organizes the statement batches such that dependent statements execute after their dependencies

• In between blocks of statements, data is synchronized from the result of a completed statement into the parameter list of another yet to be executed.
from sqlalchemy.orm import Session

session = Session(bind=some_engine)

session.add_all([
    User(name='ed'),
    User(name='jack', addresses=[address1, address2])
])

# force a flush
session.flush()
Unit of work example

-- INSERT statements
BEGIN (implicit)
INSERT INTO user (name) VALUES (?,
('ed',)
INSERT INTO user (name) VALUES (?,
('jack',)

-- statements are batched if primary key already present
INSERT INTO address (id, user_id, street, city, state, zip) VALUES (?, ?, ?, ?, ?, ?)
((1, 2, '350 5th Ave.', 'New York', 'NY', '10118'), (2, 2, '900 Market Street', 'San Francisco', 'CA', '94102'))
Dependency Sorting

• The core concept used by the UOW is the **topological sort**.

• In this sort, an ordering is produced which is compatible with a "partial ordering" - pairs of values where one must come before the other.
Topological Sort

Partial Ordering

"A" comes before "C"

"B" comes before "C"

"A" comes before "D"

Topologically Sorted Sets

A, B, C, D

B, A, C, D

A, B, D, C

B, A, D, C

A, D, B, C

... etc
The Dependency Graph

- The Unit of Work sees the mapping configuration as a "dependency graph" - Mapper objects represent nodes, relationship() objects represent edges.
- For each "edge", the Mapper on the right is dependent on the Mapper on the left.
- A dependency usually corresponds to mapper B's table having a foreign key to that of mapper A.
- These pairs of Mapper objects form the "partial ordering" passed to the topological sort.
Unit of work
sorting per-mapper

Partial Ordering

(User, Address)

User.addresses
Unit of work sorting per-mapper

Partial Ordering

(User, Address)

User.addresses

Dependency: (user, address)

ProcessAll (User->Address)

SaveUpdateAll (User)

SaveUpdateAll (Address)

DONE

Topological Sort

INSERT INTO user

INSERT INTO user

copy user.id to address.user_id

copy user.id to address.user_id

INSERT INTO address

INSERT INTO address
UOW - Cycle Resolution

• A cycle forms when a mapper is dependent on itself, or on another mapper that's ultimately dependent on it.

• Those portions of the dependency graph with cycles are broken into inter-object sorts.

• I used a function on Guido's blog to detect the cycles.

• Rationale - don't waste time sorting individual items if we don't need it!
Unit of work sorting per-row

Partial Ordering

( User, Address )
User.addresses

( User, User ) → Cycle
User.contact
Unit of work sorting per-row

Partial Ordering

(User, Address)
User.addresses

(User, User)
User.contact

Partial Ordering

(User 1 (obj), Address (mapper))

(User 2 (obj), Address (mapper))

(User 1 (obj), User 2 (obj))
Unit of work sorting per-row

Partial Ordering

( User 1 (obj) , Address (mapper) )

( User 2 (obj) , Address (mapper) )

( User 1 (obj) , User 2 (obj) )
Unit of work
sorting per-row

Partial Ordering

(User 1, Address)
(User 2, Address)
(User 1, User 2)

SaveUpdateState
INSERT INTO user

ProcessState
(User->User)
copy user.id to user.contact_id

SaveUpdateState
INSERT INTO user

ProcessAll
(User->Address)
copy user.id to address.user_id
copy user.id to address.user_id

SaveUpdateAll
(Address)
INSERT INTO address
INSERT INTO address

DONE
We're done!
Hope this was enlightening.

http://www.sqlalchemy.org