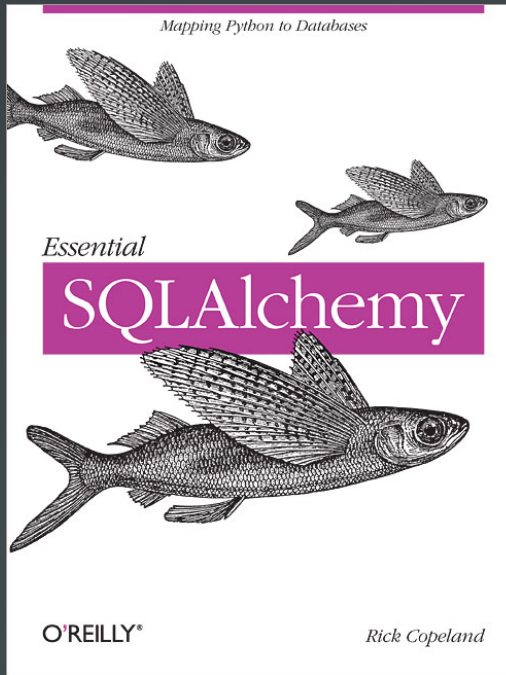


# Essential SQLAlchemy



## An Overview of SQLAlchemy

Rick Copeland  
Author, *Essential SQLAlchemy*  
Predictix, LLC



# SQLAlchemy Philosophy

- SQL databases behave less like object collections the more size and performance start to matter
- Object collections behave less like tables and rows the more abstraction starts to matter
- SQLAlchemy aims to accommodate both of these principles

From <http://www.sqlalchemy.org/>



# SQLAlchemy Philosophy (abridged)

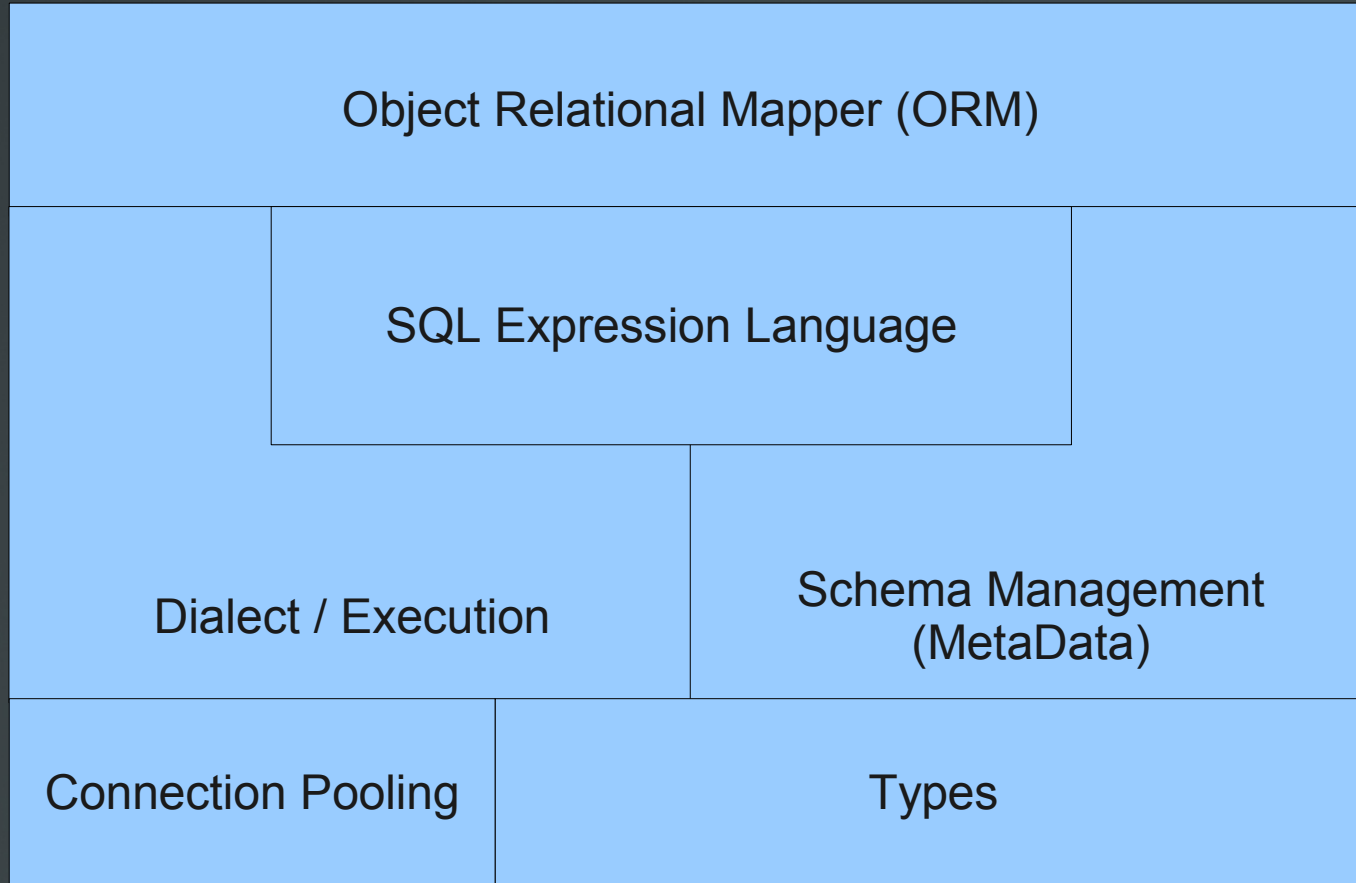
Let tables be tables

Let objects be objects

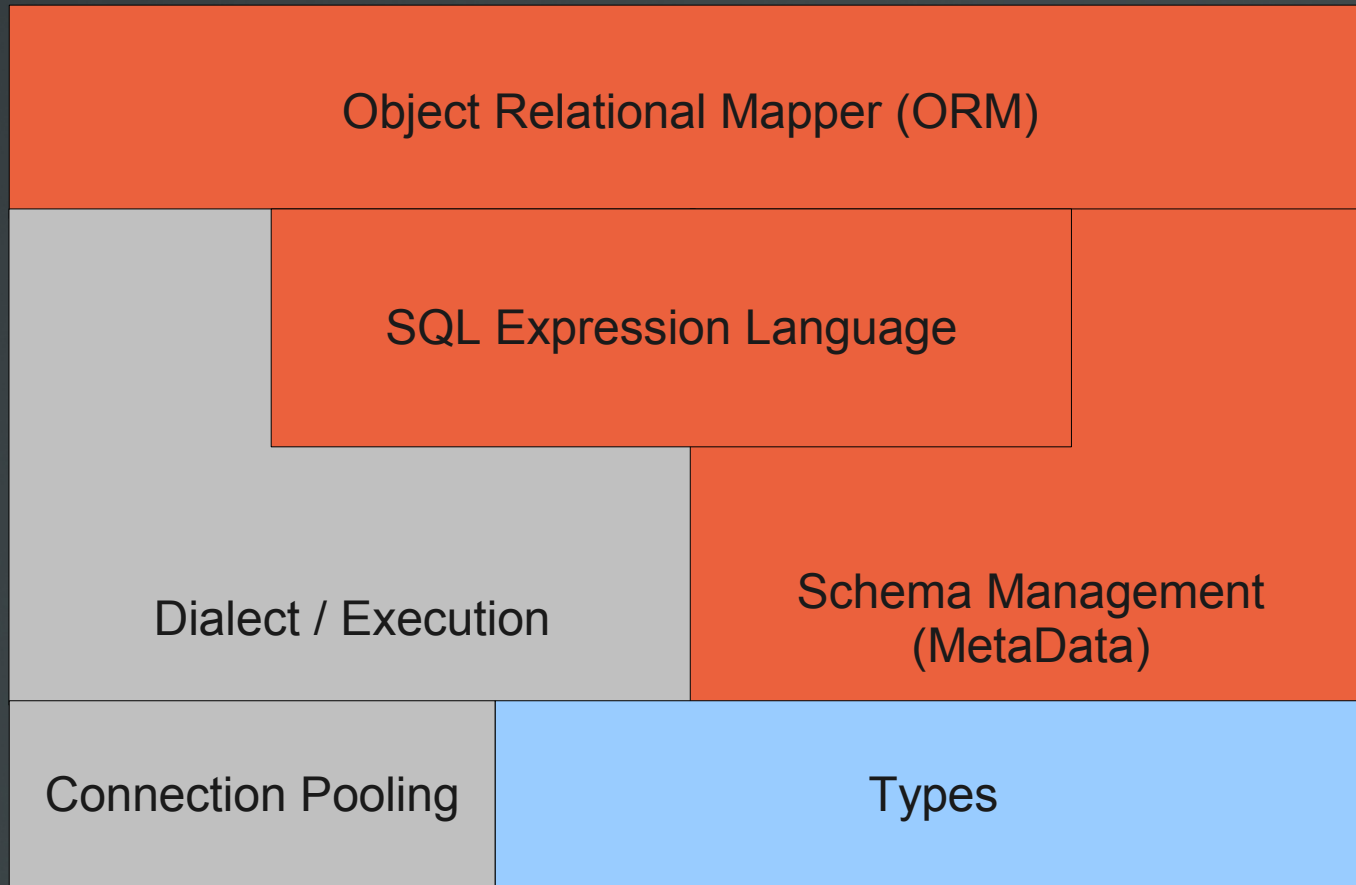
(my book is short)



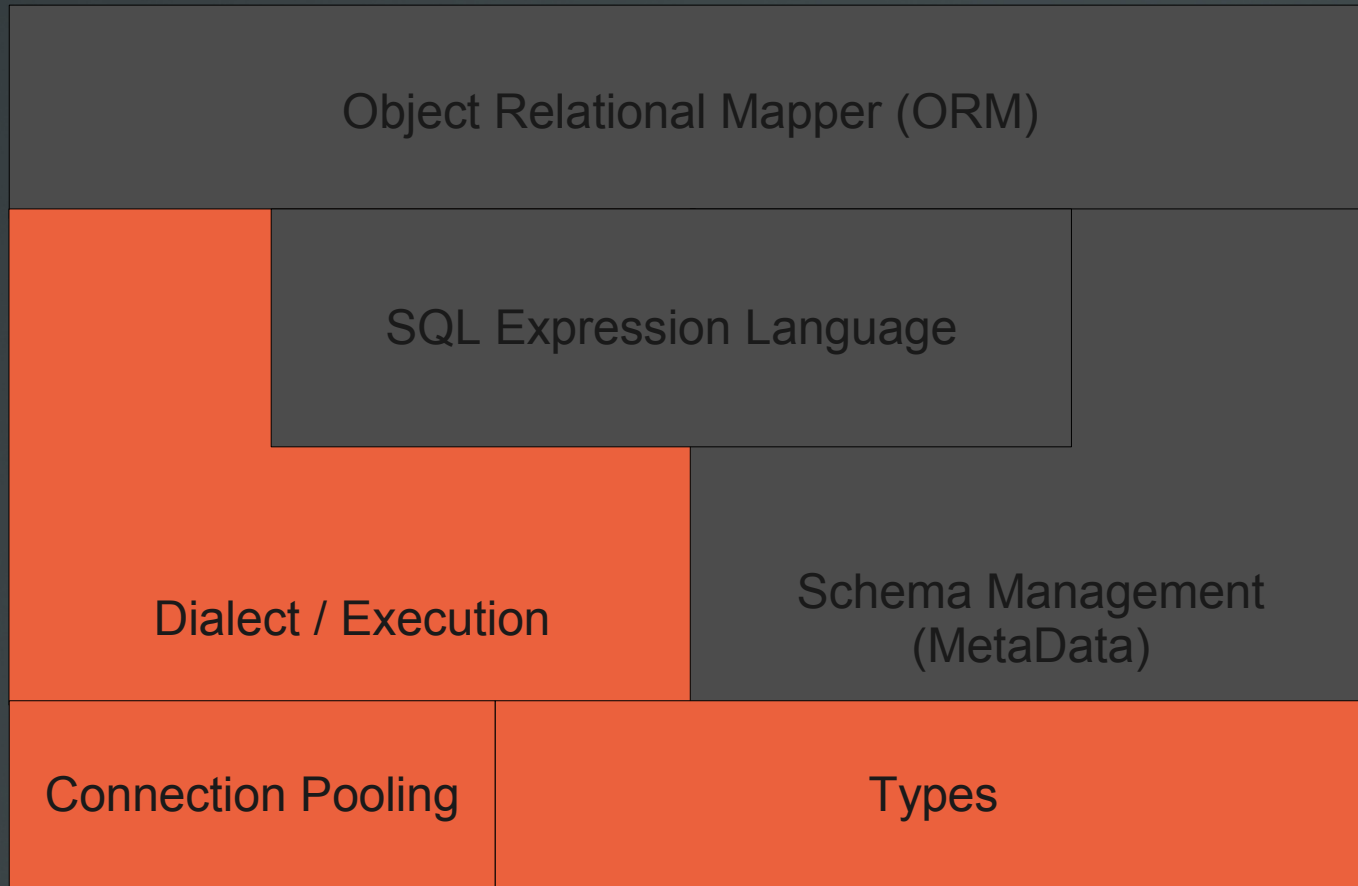
# SQLAlchemy Architecture



# SQLAlchemy Architecture (Interesting parts)



# SQLAlchemy “Plumbing”



# SQLAlchemy “Plumbing”

- Connection Pooling
  - Manage a pool of long-lived connections to the database
  - Different strategies available (one connection per thread, one per statement, one per database)
  - Usually “just works” without intervention
- Dialect / Execution
  - Provides a database independence layer
  - Postgres, SQLite, MySQL, Oracle, MS-SQL, Firebird, Informix, .... (more?)



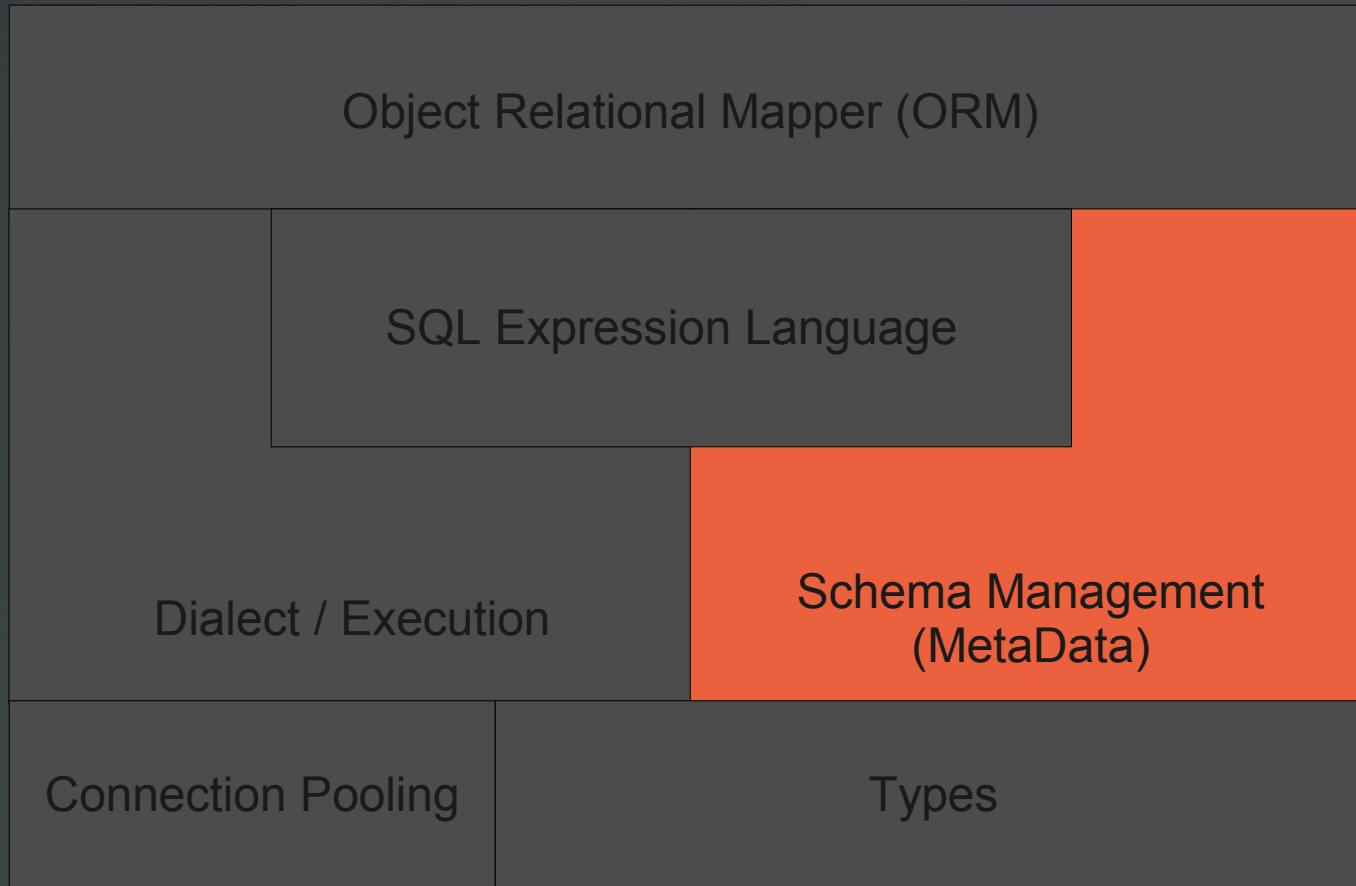
# SQLAlchemy “Plumbing”

- Types
  - Support for a variety of common SQL types
  - Support for driver-specific types (at the cost of portability)
  - TypeEngines convert Python values to SQL values and vice-versa
  - Custom TypeEngines easy to implement

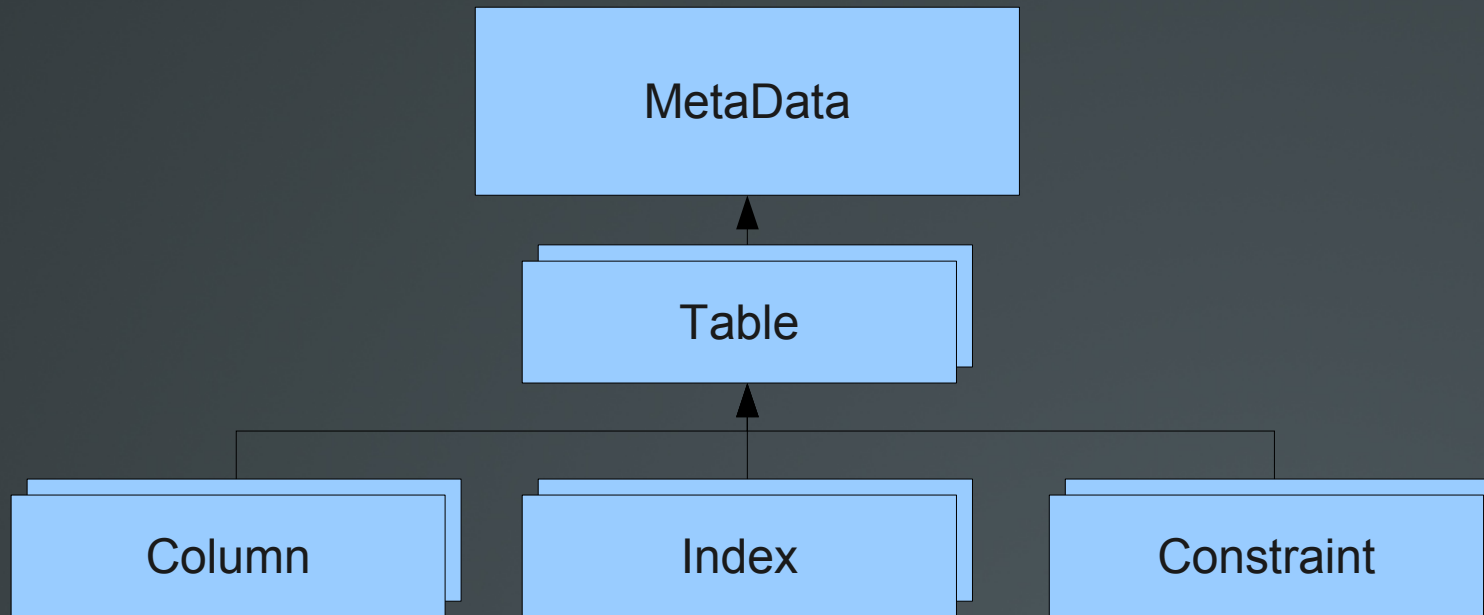




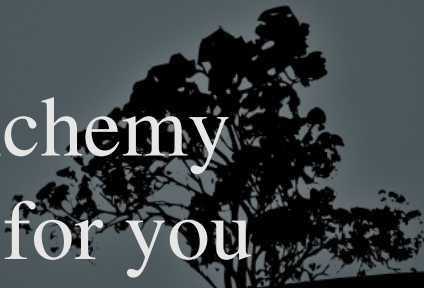
# Schema Management



# Schema Management

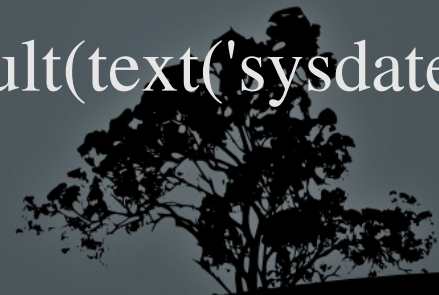


# Schema Management

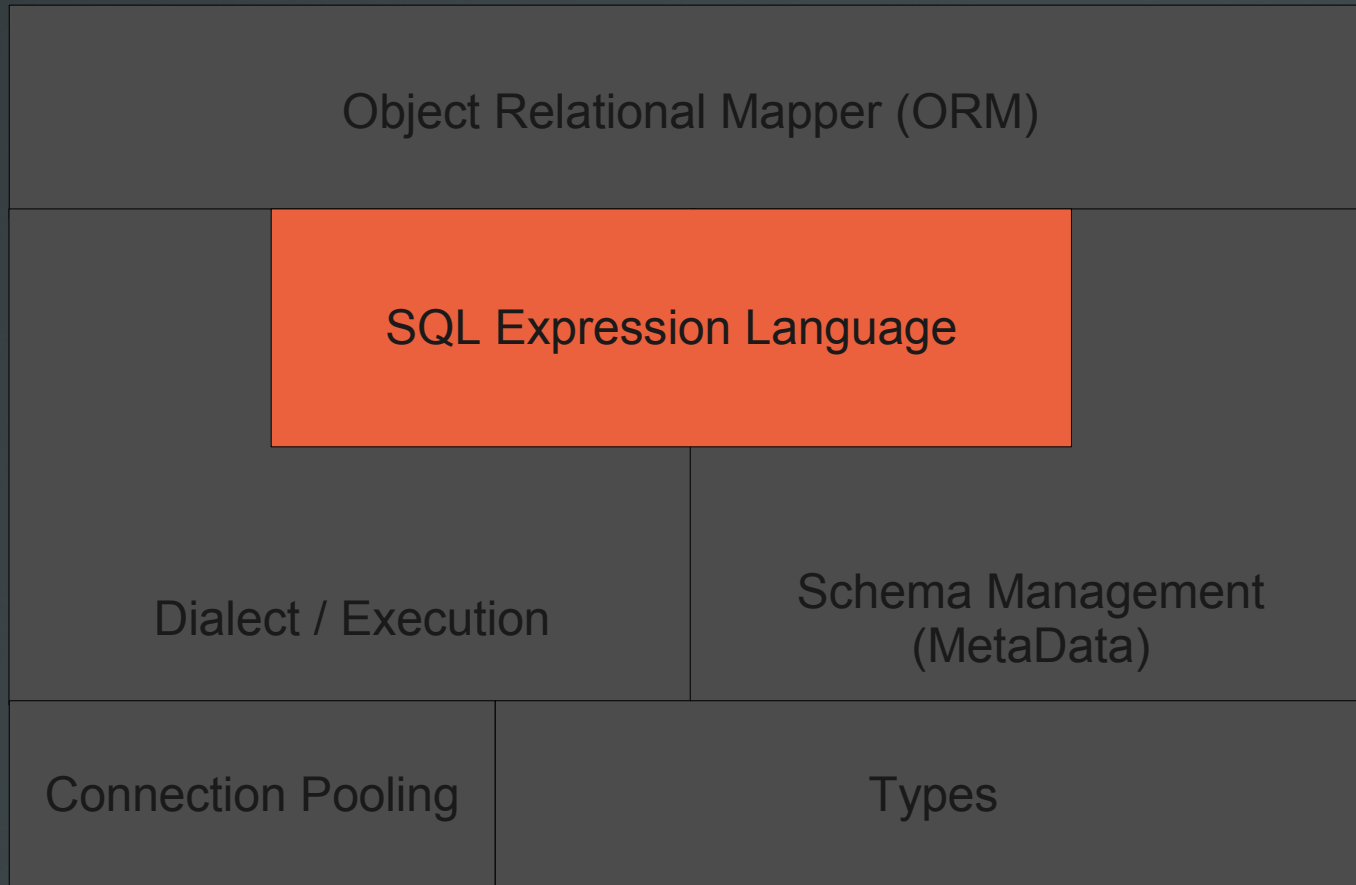
- For “blue sky” development, you can define your schema in Python and stay in Python
  - For “legacy” development, you can tell tables (or even the entire MetaData object!) to autoload from the database
  - The MetaData, Tables, and Columns provide convenient proxy objects for SQL constructs used in the SQL Expression Language
  - Foreign key relationships let SQLAlchemy automatically create join conditions for you
- 

# Schema Management


- Simple syntax for simple constraints
  - `Column('col', Integer, index=True, unique=True)`
  - `Column('col', None, ForeignKey('t2.col'))`
- Default Values
  - `Column('col', Integer, default=None)`
  - `Column('col', DateTime, default=datetime.now)`
  - `Column('col', Integer, default=select(...))`
  - `Column('col', DateTime, PassiveDefault(text('sysdate')))`



# Schema Management



# SQL Expression Language

- DDL (Data Definition Language) Statements
    - `users_table.create()` # table defined with MetaData
    - `users_table.drop()`
    - `metadata.create_all()`
    - `metadata.drop_all()`
  - DML (Data Manipulation Language)
    - `s_ins = users.insert(values=dict(name='rick', pass='foo'))`
    - `s_del = users.delete(whereclause=users.c.name=='rick')`
    - `s_upd = users.update(values=dict(age=users.c.age +  
timedelta(days=1)))`
- 

# SQL Expression Language

- Executing DML Statements
  - `s_ins.execute()`
  - `s_ins.execute(a=5, b=6)`
  - `conn.execute(s_ins, [ dict(a=1,b=1), dict(a=1,b=2)...])`



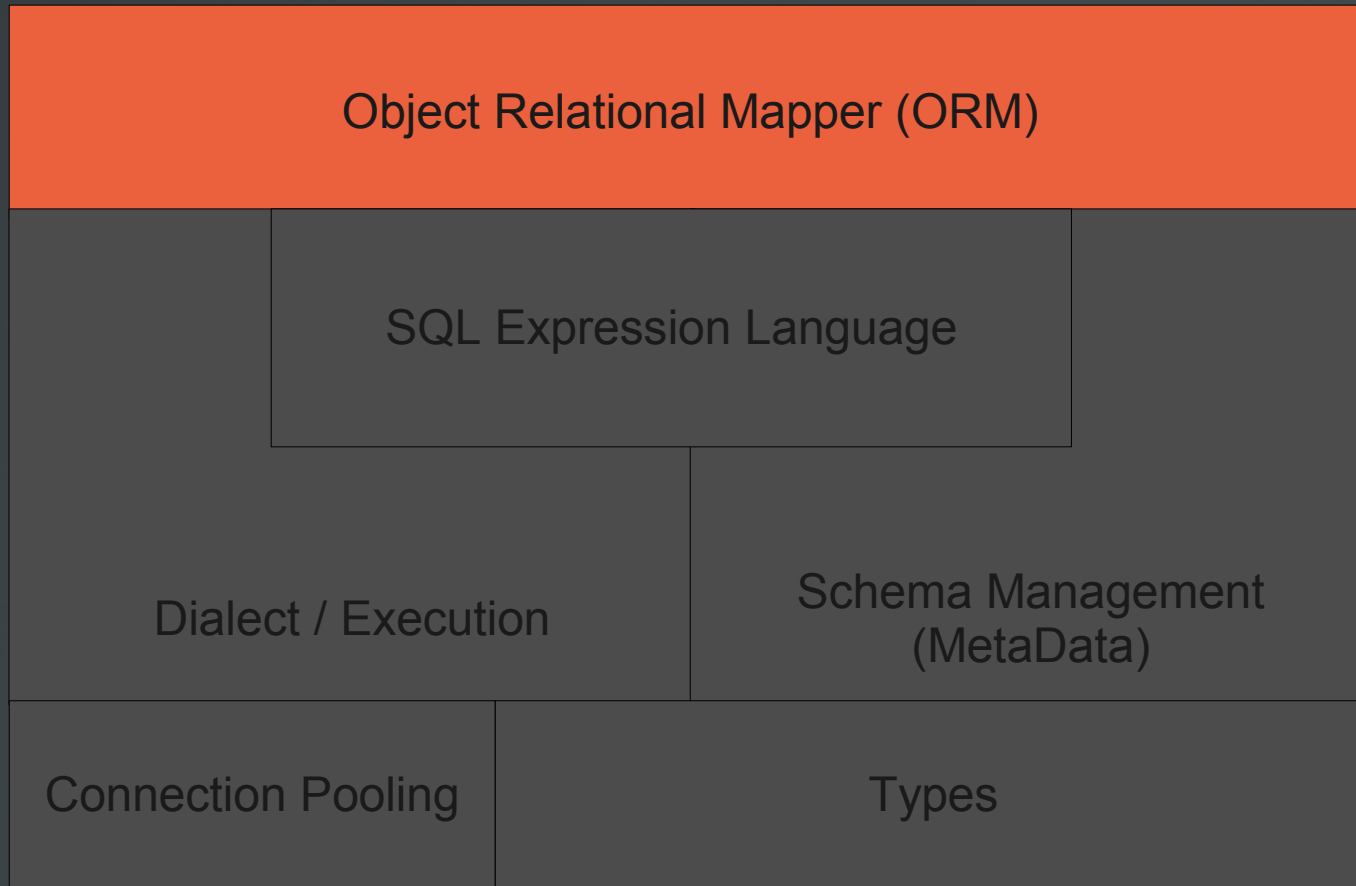
# SQL Expression Language

- DQL (Data Query Language) statements
  - `users.select()`
  - `select([users.c.user_name])`
  - `users.select(users.c.user_name=='rick')`
  - `select([users, addresses], users.c.id==addresses.c.userid)`
  - `s = text("SELECT users.fullname FROM users WHERE users.name LIKE :x")`
    - `s.execute(x='rick')`
  - `users.join(addresses).select()`
  - `users.outerjoin(addresses).select()`





# Schema Management

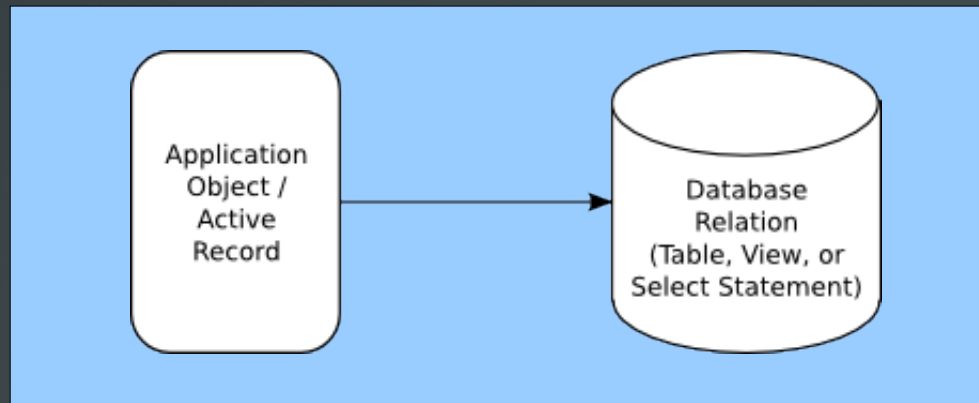


# ORM Design

- Basic idea: use the database as a persistence layer for Python objects
- Tables are classes, rows are instances
- Relationships modeled as properties



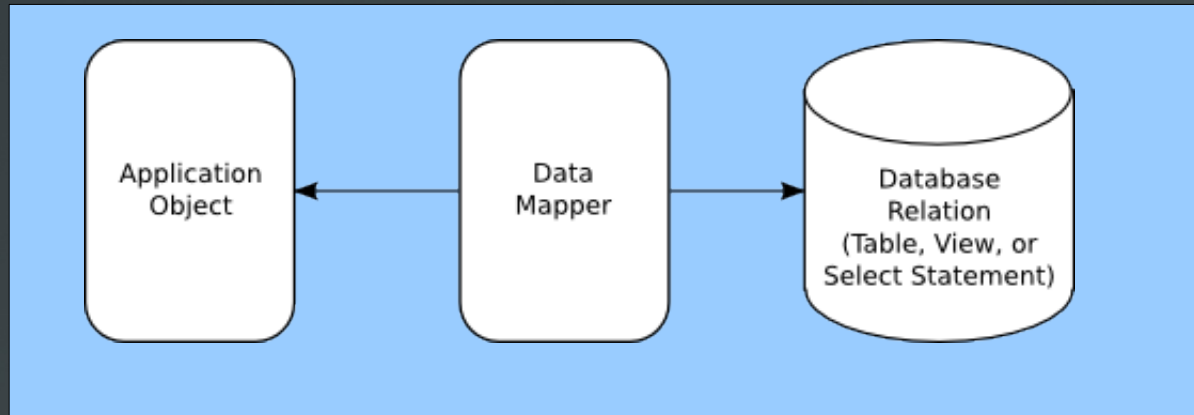
# ORM Design



Active Record – wrap every table in a class  
The class is aware of the mapping  
Examples: RoR ActiveRecord, SQLAlchemy



# ORM Design



Data Mapper – use a *mapper* to connect tables to classes  
The class is ignorant of the mapping  
Examples: SQLAlchemy, Hibernate

# The Session

- Unlike other ORMs (at least SQLAlchemy), SQLAlchemy uses the *Unit of Work* (UoW) pattern to collect changes to your objects as you make them
- At some point, these changes are *flushed* to the database
- This is a Good Thing
  - Less chattiness with the DB server
  - Sometimes the DB server can amortize compilation overhead for many updates



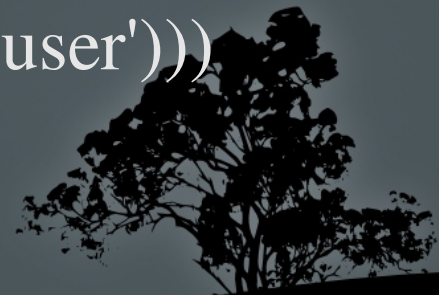
# Simple Mapping

- Example:
  - `users = Table('users', metadata, Column(...))`
  - `class User(object): pass`
  - `mapper(User, users)`
- All columns are mapped as properties



# Mapping Relations

- `users = Table('users', metadata, Column('id', ...))`
- `addresses = Table('addresses', metadata,`
  - `Column('id', ...),`
  - `Column('user_id', None, ForeignKey('users.id'))...`
- `class User(object): pass`
- `class Address(object): pass`
- `mapper(User, users, properties=dict(addresses=relation(Address, backref='user')))`
- `mapper(Address, addresses)`



# Cool advanced features I won't go over in detail

- Eager / lazy loaded relations
- Deferred column loading
- Custom collection types
- Database partitioning
  - Vertical (some tables in DB1, some in DB2)
  - Horizontal (*sharding* - one table partitioned)
- Mapping classes against arbitrary SELECT statements
- Inheritance mapping





# Questions?

