Querying with SQLAlchemy

Xavier Morera

HELPING DEVELOPERS UNDERSTAND SEARCH & BIG DATA

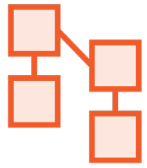
@xmorera www.xaviermorera.com

Querying with SQLAlchemy

Picking a database

Connectors and connection strings

Querying using SQL vs. ORM



Joins, and hierarchical tables

Picking a Database: Which One?

SQLite

PostgreSQL

MySQL

Oracle

Microsoft
SQL Server

Or another one...

Powerful open source database

Runs on virtually all major platforms

- Top 3 of most widely used

Client-server model

MariaDB too

Database engine

- Store and work with relational data

No need for a database server

- Simple to use and portable

Does not scale well

Object-relational db management system

- Advanced and open source

Highly programmable and extensible

Extremely efficient concurrency support

Not as popular as MySQL

Multi-model database management system

Commercial

- Widely used
- Enterprise

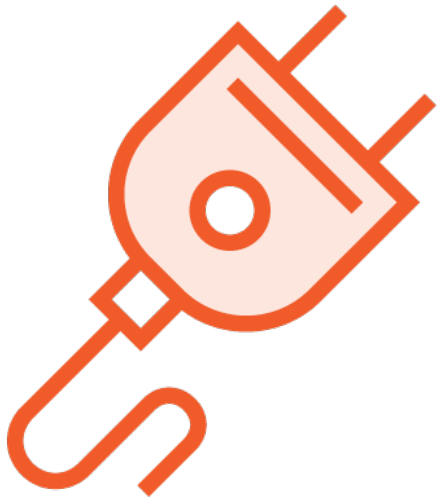
Microsoft SQL Server

Widespread in the enterprise world

- On-premises and in the cloud

Efficient and easy to use

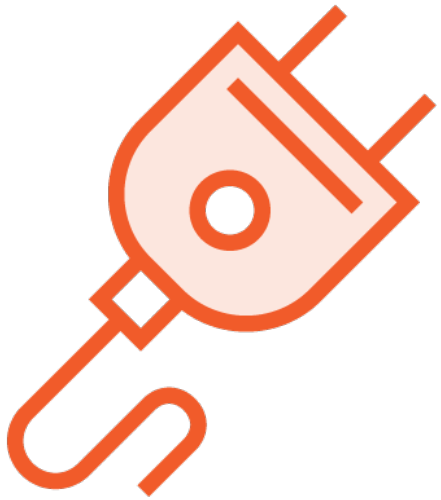
Connecting to Databases with SQLAlchemy



Connector

Connection String

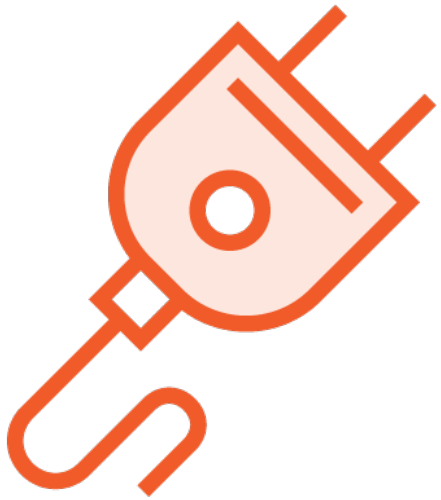
Connecting to Databases with SQLAlchemy



Connector

Connection String

Connecting to Databases with SQLAlchemy



Connectors

Connection String

Connectors

SQLite

PostgreSQL

MySQL

Oracle

MS SQL Server

Or another one...

Connectors

SQLite pysqlite (sqlite3)	**PostgreSQL** psycopg2	**MySQL** MySQL Connector Python
Oracle cx_Oracle	**MS SQL Server** PyODBC	More?

SQLAlchemy 1.3 Documentation

Release: **1.3.0b2** **BETA RELEASE** | Release Date: January 25, 2019

SQLAlchemy 1.3 Documentation

BETA RELEASE

Contents | Index

Search terms: search...

ads via Carbon

Dialects

PostgreSQL

MySQL

SQLite

- Support for the SQLite database.
- Date and Time Types
  - Ensuring Text affinity
- SQLite Auto Incrementing Behavior
  - Using the AUTOINCREMENT Keyword
  - Allowing autoincrement behavior SQLAlchemy types other than Integer/INTEGER
- Database Locking Behavior /

SQLite

Support for the SQLite database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- pysqlite
- pysqlcipher

Date and Time Types

SQLite does not have built-in DATE, TIME, or DATETIME types, and pysqlite does not provide out of the box functionality for translating values between Python *datetime* objects and a SQLite-supported format. SQLAlchemy's own `DateTime` and related types provide date formatting and parsing functionality when SQLite is used. The implementation classes are `DATETIME`, `DATE` and `TIME`. These types represent dates and times as ISO formatted strings, which also nicely support ordering. There's no reliance on typical "libc" internals for these functions so historical dates are fully supported.

Ensuring Text affinity

The DDL rendered for these types is the standard `DATE`, `TIME` and `DATETIME` indicators. However, custom storage formats can also be applied to these

SQLAlchemy 1.3 Documentation

Release: **1.3.0b2** **BETA RELEASE** | Release Date: January 25, 2019

PostgreSQL

Support for the PostgreSQL database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- psycopg2
- pg8000
- psycopg2cffi
- py-postgresql
- pygresql
- zxJDBC for Jython

Sequences/SERIAL/IDENTITY

PostgreSQL supports sequences, and SQLAlchemy uses these as the default means of creating new primary key values for integer-based primary key columns. When creating tables, SQLAlchemy will issue the `SERIAL` datatype for integer-based primary key columns, which generates a sequence and server side default corresponding to the column.

SQLAlchemy 1.3 Documentation

Release: **1.3.0b2** **BETA RELEASE** | Release Date: January 25, 2019

MySQL

Support for the MySQL database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- mysqlclient (maintained fork of MySQL-Python)
- PyMySQL
- MySQL Connector/Python
- CyMySQL
- OurSQL
- Google Cloud SQL
- PyODBC
- zxjdbc for Jython

Supported Versions and Features

SQLAlchemy supports MySQL starting with version 4.1 through modern releases. However, no heroic measures are taken to work around major missing SQL features - if your server version does not support sub-selects, for example, they won't work in SQLAlchemy either.

SQLAlchemy 1.3 Documentation

Release: **1.3.0b2** **BETA RELEASE** | Release Date: January 25, 2019

SQLAlchemy 1.3 Documentation

BETA RELEASE

Contents | Index

Search terms: search...

ads via Carbon

Dialects

PostgreSQL
MySQL
SQLite
Oracle

- Support for the Oracle database.
- Connect Arguments
- Auto Increment Behavior
- Identifier Casing
- LIMIT/OFFSET Support
- RETURNING Support
- ON UPDATE CASCADE
- Oracle 8 Compatibility
- Synonym/DBLINK Reflection

Oracle

Support for the Oracle database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- cx-Oracle
- zxJDBC for Jython

Connect Arguments

The dialect supports several `create_engine()` arguments which affect the behavior of the dialect regardless of driver in use.

- `use_ansi` - Use ANSI JOIN constructs (see the section on Oracle 8). Defaults to `True`. If `False`, Oracle-8 compatible constructs are used for joins.
- `optimize_limits` - defaults to `False`. see the section on LIMIT/OFFSET.
- `use_binds_for_limits` - defaults to `True`. see the section on LIMIT/OFFSET.

SQLAlchemy 1.3 Documentation

Release: **1.3.0b2** **BETA RELEASE** | Release Date: January 25, 2019

SQLAlchemy 1.3 Documentation

BETA RELEASE

Contents | Index

Search terms: search...

Microsoft SQL Server

Support for the Microsoft SQL Server database.

DBAPI Support

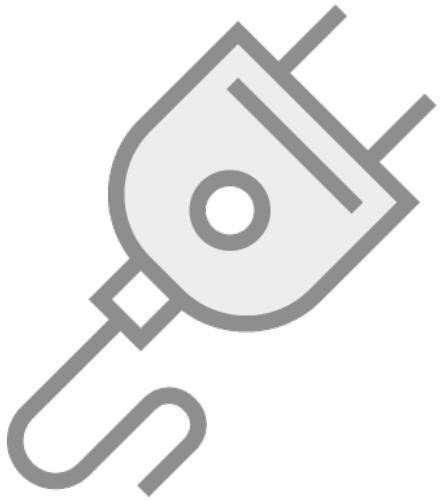
The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- PyODBC
- mxODBC
- pymssql
- zxJDBC for Jython
- adodbapi

Auto Increment Behavior / IDENTITY Columns

SQL Server provides so-called "auto incrementing" behavior using the `IDENTITY` construct, which can be placed on any single integer column in a table. SQLAlchemy considers `IDENTITY` within its default "autoincrement" behavior for an integer primary key column, described at `Column.autoincrement`. This means that by default, the first integer primary key column in a `Table` will be considered to be the identity column and will generate DDL as such:

Connecting to Databases with SQLAlchemy



Connector

Connection String

Anatomy of a Connection String

```
engine = create_engine('sqlite:///sqlalchemy_sqlite.db')
```

Anatomy of a Connection String

```
engine = create_engine('sqlite:///sqlalchemy_sqlite.db')
```

Anatomy of a Connection String

```
sqlite:///sqlalchemy_sqlite.db
```

Anatomy of a Connection String

```
dialect:///
```

Anatomy of a Connection String

```
dialect:///dbname
```

Anatomy of a Connection String

```
dialect[+driver]://user:password@hostname/dbname
```

Anatomy of a Connection String

```
dialect[+driver]://user:password@hostname/dbname[?key=value]
```

```
engine_sqlite = create_engine('sqlite:///importing_sqlite.db')

engine_postgres = create_engine('postgresql://xavier:postgres@localhost:5432/importing_postgres')

engine_mysql = create_engine('mysql+mysqlconnector://root:mysql@localhost:3306/importing_mysql')
```

A SQL Query

```
SELECT OwnerUserId,
    SUM(AnswerCount) AS 'TotalAnswers',
    SUM(ViewCount) AS 'TotalViews'
FROM posts
WHERE owneruserid is not NULL
GROUP BY OwnerUserId
ORDER BY 'TotalAnswers' DESC
LIMIT 10;
```

A SQL Query

```
SELECT OwnerUserId,
   SUM(AnswerCount) AS 'TotalAnswers',
   SUM(ViewCount) AS 'TotalViews'
FROM posts
WHERE owneruserid is not NULL
GROUP BY OwnerUserId
ORDER BY 'TotalAnswers' DESC
LIMIT 10;
```

```
SELECT OwnerUserId,
    SUM(AnswerCount) AS 'TotalAnswers',
    SUM(ViewCount) AS 'TotalViews'
FROM posts
WHERE owneruserid is not NULL
GROUP BY OwnerUserId
ORDER BY 'TotalAnswers' DESC
LIMIT 10;
```

```
mysql> SELECT OwnerUserId,
    -> SUM(AnswerCount) AS 'TotalAnswers',
    -> SUM(ViewCount) AS 'TotalViews'
    -> FROM posts
    -> WHERE owneruserid is not NULL
    -> GROUP BY OwnerUserId
    -> ORDER BY 'TotalAnswers' DESC
    -> LIMIT 10;
```

```
mysql> SELECT OwnerUserId,
    -> SUM(AnswerCount) AS 'TotalAnswers',
    -> SUM(ViewCount) AS 'TotalViews'
    -> FROM posts
    -> WHERE owneruserid is not NULL
    -> GROUP BY OwnerUserId
    -> ORDER BY 'TotalAnswers' DESC
    -> LIMIT 10;
```

```
mysql> SELECT OwnerUserId,
    -> SUM(AnswerCount) AS 'TotalAnswers',
    -> SUM(ViewCount) AS 'TotalViews'
    -> FROM posts
    -> WHERE owneruserid is not NULL
    -> GROUP BY OwnerUserId
    -> ORDER BY 'TotalAnswers' DESC
    -> LIMIT 10;
+-------------+--------------+------------+
| OwnerUserId | TotalAnswers | TotalViews |
+-------------+--------------+------------+
|           5 |            1 |        448 |
|          36 |            5 |        528 |
|          51 |         NULL |       NULL |
|          22 |         NULL |       NULL |
|          66 |            4 |       1335 |
|          64 |            0 |        543 |
|          63 |            2 |        322 |
|          -1 |         NULL |       NULL |
|          84 |           42 |      37229 |
|          96 |            5 |        311 |
```

```
result = engine.execute("""SELECT OwnerUserId,
    SUM(AnswerCount) AS 'TotalAnswers',
    SUM(ViewCount) AS 'TotalViews'
FROM posts
WHERE owneruserid is not NULL
GROUP BY OwnerUserId
ORDER BY 'TotalAnswers' DESC
LIMIT 10;""").fetchall()

pd.DataFrame(result, columns=['OwnerUserId', 'TotalAnswers',
'TotalViews'])
```

Executing a SQL Query

Use execute with a SQL statement

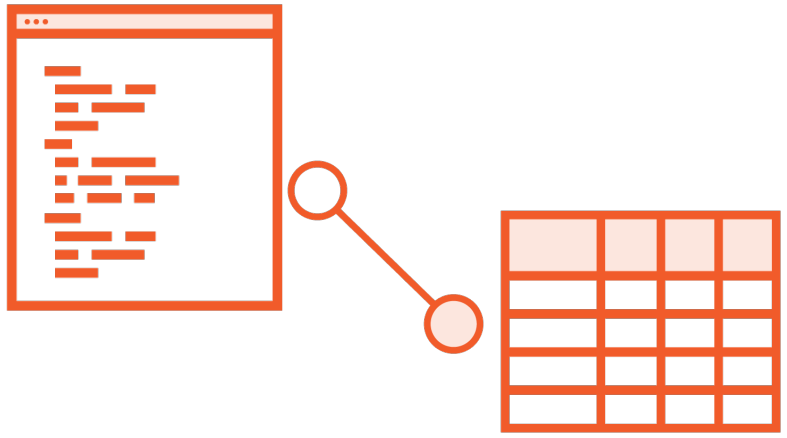
- Error prone

Use SQLAlchemy ORM

Object-relational Mapping

Programming technique for converting data between incompatible type systems in object-oriented programming languages

Object-relational Mapper



Table

- Represents a table in the database

Mapper

- Maps a Python class to a table

Class

- Object that defines how a record maps to an object

Classical Mapping or Declarative API

```
from sqlalchemy import Table, MetaData, Column, Integer,
String
from sqlalchemy.orm import mapper


metadata = MetaData()
```

Classical Mapping

Mapped class using the mapper function

- Original class mapping

Create MetaData

- What the database looks like for SQLAlchemy

```
tags = Table('Tags', metadata,
  Column('Id', Integer, primary_key=True),
  Column('Count', Integer),
  Column('ExcerptPostId', Integer),
  Column('TagName', String(255)),
  Column('WikiPostId', Integer))
```

Classical Mapping

Define our Table

- Specify each Column, with the type
- Other attributes

```
class Tags(object):
    def __init__(self, Count, ExcerptPostId, TagName, WikiPostId):
        self.Count = Count
        self.ExcerptPostId = ExcerptPostId
        self.TagName = TagName
        self.WikiPostId = WikiPostId

tags_mapper = mapper(Tags, tags)
```

Classical Mapping

Define Tags class

- Initialize

Associate via mapper function

```
larger_tags = tags.select(Tags.Count > 1000)

larger_tags

engine.execute(larger_tags).fetchall()
```

Classical Mapping

Create a query

- Use select and a condition

Execute

Declarative

ORM

Typically used system

- Provided by SQLAlchemy ORM

Define classes

Mapped to relational database tables

Series of extensions

- On top of the mapper construct

```
from sqlalchemy.orm import sessionmaker

session = sessionmaker()

session.configure(bind=engine)

my_session = session()
```

Session

Core concepts of SQLAlchemy

Establishes and maintains conversations

- Between our program and the database
- Entry point for queries

```
my_session.query(Tags).all()

len(my_session.query(Tags).all())

my_session.query(Tags).first()

my_session.query(Tags).first().TagName

my_session.query(Tags.Id, Tags.TagName).first()
```

Session

Can use session to query

- List of Tags objects
- Nicer representation?

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

Declarative Base

Base class for declarative class definitions

- Define models
- Connect them to the database

```
class Users(Base):
    __tablename__ = 'users'

    Id = db.Column(db.Integer, primary_key = True)
    Reputation = db.Column(db.Integer)
    CreationDate = db.Column(db.DateTime)
    DisplayName = db.Column(db.String(255))
    LastAccessDate = db.Column(db.DateTime)
    …
```

Define a Model

Class

- Inherit from declarative_base
- Specify __tablename__
- Columns, with types

Define a Model

```
class Users(Base):
    __tablename__ = 'users'

    Id = db.Column(db.Integer, primary_key = True)
    Reputation = db.Column(db.Integer)
    CreationDate = db.Column(db.DateTime)
    DisplayName = db.Column(db.String)
    LastAccessDate = db.Column(db.DateTime)
    WebsiteUrl = db.Column(db.String)
    Location = db.Column(db.String)
    AboutMe = db.Column(db.String)
    Views = db.Column(db.Integer)
    UpVotes = db.Column(db.Integer)
    DownVotes = db.Column(db.Integer)
    AccountId = db.Column(db.Integer)

    def __repr__(self):
        return "<{0} Id: {1} - Name: {2}>".format(self.__class__.__name__,
self.Id, self.DisplayName)
```

```
my_session.query(Users).first()
type(my_session.query(Users).first())
my_session.query(Users).first().DisplayName
for each_user in my_session.query(Users):
    print(each_user)
```

Querying

Can query using the session

Returns the __repr__ of the object

Can access the fields

Iterate over the results

```
the_query = my_session.query(Users)
type(the_query)
print(the_query)
engine_echo =
db.create_engine('mysql+mysqlconnector://root:mysql@localho
st:3306/sqlalchemy_mysql', echo=True)
connection_echo = engine_echo.connect()
session_echo = sessionmaker(bind=engine_echo)()
session_echo.query(Users).first()
```

The Query and Echo Parameter

What query is executed?

- Print the query

Use the echo parameter when creating engine

- Execute and get the SQL statements

```
my_session.query(Users).filter_by(DisplayName='Community').all()

my_session.query(Users).filter(Users.DisplayName=='Community').all()
```

Filtering Results

Possible to use filter_by

- Can be prone to confusion

Use filter

- Explicit

```
my_session.query(Users.DisplayName).filter(Users.DisplayNam
e.like('%Comm%')).all()

my_session.query(Users.DisplayName).filter(Users.DisplayNam
e.contains('Comm')).all()
```

ClauseElements

What we just passed to filter

- Base class for elements of a programmatically constructed SQL expression

For example like, but there are many others

```
from sqlalchemy import func

dir(func)

tags_count =
my_session.query(func.sum(Tags.Count)).scalar()
```

Functions

Many functions available through func

- Function generator

Use scalar to return a single element

```
my_session.query(Users.DisplayName, db.cast((Users.UpVotes -
Users.DownVotes), db.Numeric(12, 2)).label('vote_difference') ,
Users.UpVotes, Users.DownVotes).all()
```

Operators and Labels

Perform operations

- Calculate a difference between two columns
- Use cast

Create a label to refer to the column

```
my_session.query(Users.DisplayName, db.cast((Users.UpVotes -
Users.DownVotes), db.Numeric(12, 2)).label('vote_difference'),
Users.UpVotes, Users.DownVotes).limit(5).all()
```

Limiting Results

Control how many results are retrieved

- With limit

```
my_session.query(Users.DisplayName, db.cast((Users.UpVotes -
Users.DownVotes), db.Numeric(12, 2)).label('vote_difference'),
Users.UpVotes,
Users.DownVotes).order_by('vote_difference').limit(5).all()

my_session.query(Users.DisplayName, db.cast((Users.UpVotes -
Users.DownVotes), db.Numeric(12, 2)).label('vote_difference'),
Users.UpVotes,
Users.DownVotes).order_by(db.desc('vote_difference')).limit(5).all()
```

Ordering Results

Use order_by to sort results

- Ascending (default)
- Descending (desc)

```
my_session.query(Users.DisplayName).filter(Users.DisplayName ==
'Community', Users.DownVotes.between(300,600)).all()

my_session.query(Users.DisplayName,
Users.DownVotes).filter(db.or_(Users.DisplayName == 'Community',
Users.DownVotes.between(300,600))).all()
```

Conjunctions

Filter by multiple statements

- Using and_, or_, and not_
- And is the default, but better to add it

```
class Posts(Base):
    __tablename__ = 'posts'
    Id = db.Column(db.Integer(), primary_key=True)
    Title = db.Column(db.String(255), nullable=False)
    ViewCount = db.Column(db.Integer(), default=1000)

    PostTypeId = db.Column(db.Integer(), default=True)
    OwnerUserId = db.Column(db.Integer())
```

Define Posts

The Users can have many Posts

Use it to join data

```
my_session.query(Users, Posts).filter(Users.Id ==
Posts.OwnerUserId).limit(1).all()
```

Implicit Join

Do not use the join

Returns both Users and Posts

- Where Id matches OwnerUserId

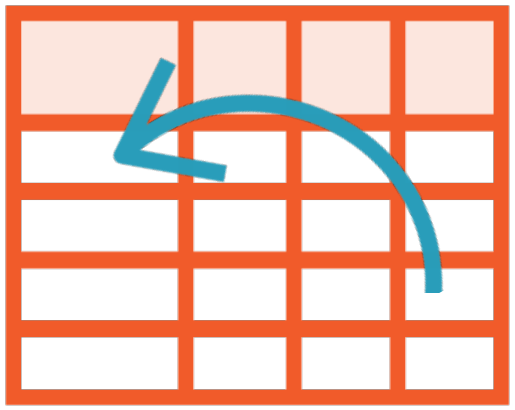
```
my_session.query(Users, Posts).join(Posts, Users.Id ==
Posts.OwnerUserId).limit(1).all()
```

Explicit Join

Use join

From Users to Posts

Hierarchical Tables



Tables that refer to themselves

Posts

- An answer refers to a question
- Both are posts
- ParentId to Id

Self join

- Using alias

```
class Posts(Base):
    __tablename__ = 'posts'
    Id = db.Column(db.Integer(), primary_key=True)
    Title = db.Column(db.String(255), nullable=False)
    ViewCount = db.Column(db.Integer(), default=1000)

    PostTypeId = db.Column(db.Integer(), default=True)
    OwnerUserId = db.Column(db.Integer())
    __table_args__ = {'extend_existing': True}
    AnswerCount = db.Column(db.Integer)
    ParentId = db.Column(db.Integer)
```

Extending a Model

Extend a model using __table_args__

- With extend_existing set to True

```
my_session.query(Posts.Id, Posts.Title, Posts.AnswerCount).filter(Posts.Id ==
14).all()

my_session.query(Posts.Id).filter(Posts.ParentId == 14).all()

from sqlalchemy.orm import aliased

Questions = aliased(Posts)

my_session.query(Posts.Id, Questions.Id, Posts.ViewCount, Posts.Score,
Questions.Score).filter(Posts.Id ==
Questions.ParentId).order_by(db.desc(Posts.ViewCount)).limit(10).all()
```

Hierarchical Tables

Create an alias with aliased

Use the aliased table

- To join data

Takeaway

Querying

- Common operation

Picking a database

- Connector
- Connection string

Takeaway

Query with SQL statements

- With engine.execute

Object-relational mapper

- Classical mappings
- Declarative API

Takeaway

Declarative API

- Session & declarative_base
- Define a model
  - Base, __tablename__, and columns

Query

- Filter, functions, sort, limit, join, self join