SQLAlchemy Documentation

Release 0.5.4

Mike Bayer

May 17, 2009

CONTENTS

CHAPTER

ONE

OVERVIEW / INSTALLATION

1.1 Overview

The SQLAlchemy SQL Toolkit and Object Relational Mapper is a comprehensive set of tools for working with databases and Python. It has several distinct areas of functionality which can be used individually or combined together. Its major API components, all public-facing, are illustrated below:

```
+-----------------------------------------------------------+
|             Object Relational Mapper (ORM)                |
+-----------------------------------------------------------+
+---------+ +------------------------------------+ +--------+
|         | |       SQL Expression Language      | |        |
|         | +------------------------------------+ |        |
|         +-----------------------+ +--------------+        |
|        Dialect/Execution        | |    Schema Management  |
+---------------------------------+ +-----------------------+
+----------------------+ +----------------------------------+
|  Connection Pooling  | |              Types               |
+----------------------+ +----------------------------------+
```

Above, the two most significant front-facing portions of SQLAlchemy are the **Object Relational Mapper** and the **SQL Expression Language**. These are two separate toolkits, one building off the other. SQL Expressions can be used independently of the ORM. When using the ORM, the SQL Expression language is used to establish object-relational configurations as well as in querying.

1.2 Tutorials

- *Object Relational Tutorial* - This describes the richest feature of SQLAlchemy, its object relational mapper. If you want to work with higher-level SQL which is constructed automatically for you, as well as management of Python objects, proceed to this tutorial.
- *SQL Expression Language Tutorial* - The core of SQLAlchemy is its SQL expression language. The SQL Expression Language is a toolkit all its own, independent of the ORM package, which can be used to construct manipulable SQL expressions which can be programmatically constructed, modified, and executed, returning cursor-like result sets. It's a lot more lightweight than the ORM and is appropriate for higher scaling SQL operations. It's also heavily present within the ORM's public facing API, so advanced ORM users will want to master this language as well.

1.3 Main Documentation

- *Mapper Configuration* - A comprehensive walkthrough of major ORM patterns and techniques.
- *Using the Session* - A detailed description of SQLAlchemy's Session object
- *Database Engines* - Describes SQLAlchemy's database-connection facilities, including connection documentation and working with connections and transactions.
- *Database Meta Data* - All about schema management using `MetaData` and `Table` objects; reading database schemas into your application, creating and dropping tables, constraints, defaults, sequences, indexes.
- *Connection Pooling* - Further detail about SQLAlchemy's connection pool library.
- *Column and Data Types* - Datatypes included with SQLAlchemy, their functions, as well as how to create your own types.
- *sqlalchemy.ext* - Included addons for SQLAlchemy

1.4 API Reference

An organized section of all SQLAlchemy APIs is at *API Reference*.

1.5 Installing SQLAlchemy

Installing SQLAlchemy from scratch is most easily achieved with setuptools. Assuming it's installed, just run this from the command-line:

```
# easy_install SQLAlchemy
```

This command will download the latest version of SQLAlchemy from the Python Cheese Shop and install it to your system.

- setuptools
- install setuptools
- pypi

Otherwise, you can install from the distribution using the `setup.py` script:

```
# python setup.py install
```

1.6 Installing a Database API

SQLAlchemy is designed to operate with a DB-API implementation built for a particular database, and includes support for the most popular databases. The current list is at *Supported Databases*.

1.7 Checking the Installed SQLAlchemy Version

This documentation covers SQLAlchemy version 0.5. If you're working on a system that already has SQLAlchemy installed, check the version from your Python prompt like this:

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

1.8 0.4 to 0.5 Migration

Notes on what's changed from 0.4 to 0.5 is available on the SQLAlchemy wiki at 05Migration.

CHAPTER

TWO

OBJECT RELATIONAL TUTORIAL

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.

2.1 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
```

2.2 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.

2.3 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)
PRAGMA table_info("users")
()
CREATE TABLE users (
    id INTEGER NOT NULL,
    name VARCHAR,
    fullname VARCHAR,
    password VARCHAR,
    PRIMARY KEY (id)
)
()
COMMIT
```

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 Postgres 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.

2.4 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.

2.5 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.

2.6 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.

2.7 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.

2.8 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()
BEGIN
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
['ed', 'Ed Jones', 'edspassword']
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, u
FROM users
WHERE users.name = ?
 LIMIT 1 OFFSET 0
['ed']>>> 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()
UPDATE users SET password=? WHERE users.id = ?
['f8s7ccs', 1]
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
['wendy', 'Wendy Williams', 'foobar']
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
['mary', 'Mary Contrary', 'xxg527']
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
['fred', 'Fred Flinstone', 'blah']
COMMIT
```

`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
BEGIN
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, u
FROM users
WHERE users.id = ?
[1]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.

2.9 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()
UPDATE users SET name=? WHERE users.id = ?
['Edwardo', 1]
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
['fakeuser', 'Invalid', '12345']
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name IN (?, ?)
['Edwardo', 'fakeuser'][<User('Edwardo','Ed Jones', 'f8s7ccs')>, <User('fakeuser','Invalid'
```

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()
ROLLBACK>>> ed_user.name
BEGIN
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.id = ?
[1]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()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name IN (?, ?)
['ed', 'fakeuser'][<User('ed','Ed Jones', 'f8s7ccs')>]
```

2.10 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):
...     print instance.name, instance.fullname
SELECT users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users ORDER BY users.id
[]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
SELECT users.name AS users_name, users.fullname AS users_fullname
FROM users
[]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
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
[]<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
SELECT users_1.id AS users_1_id, users_1.name AS users_1_name, users_1.fullname AS users_1_
FROM users AS users_1
[]
<User('ed','Ed Jones', 'f8s7ccs')> ed
<User('wendy','Wendy Williams', 'foobar')> wendy
<User('mary','Mary Contrary', 'xxg527')> mary
<User('fred','Fred Flinstone', 'blah')> fred
```

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
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users ORDER BY users.id
LIMIT 2 OFFSET 1
[]<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
SELECT users.name AS users_name FROM users
WHERE users.fullname = ?
['Ed Jones']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
SELECT users.name AS users_name FROM users
WHERE users.fullname = ?
['Ed Jones']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
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, u
FROM users
WHERE users.name = ? AND users.fullname = ?
['ed', 'Ed Jones']<User('ed','Ed Jones', 'f8s7ccs')>
```

2.10.1 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']))
  ```

- IS 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.

2.10.2 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()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name LIKE ? ORDER BY users.id
['%ed'][<User('ed','Ed Jones', 'f8s7ccs')>, <User('fred','Fred Flinstone', 'blah')>]
```

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

```
>>> query.first()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name LIKE ? ORDER BY users.id
 LIMIT 1 OFFSET 0
['%ed']<User('ed','Ed Jones', 'f8s7ccs')>
```

`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
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name LIKE ? ORDER BY users.id
 LIMIT 2 OFFSET 0
['%ed']Multiple rows were found for one()

>>> from sqlalchemy.orm.exc import NoResultFound
>>> try:
...     user = query.filter(User.id == 99).one()
... except NoResultFound, e:
...     print e
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name LIKE ? AND users.id = ? ORDER BY users.id
 LIMIT 2 OFFSET 0
['%ed', 99]No row was found for one()
```

2.10.3 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
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, u
FROM users
WHERE id<224 ORDER BY id
[]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()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, u
FROM users
WHERE id<? and name=? ORDER BY users.id
LIMIT 2 OFFSET 0
[224, 'fred']<User('fred','Fred Flinstone', 'blah')>
```

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=
SELECT * FROM users where name=?
['ed'][<User('ed','Ed Jones', 'f8s7ccs')>]
```

2.11 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)
PRAGMA table_info("users")
()
PRAGMA table_info("addresses")
()
CREATE TABLE addresses (
    id INTEGER NOT NULL,
    email_address VARCHAR NOT NULL,
    user_id INTEGER,
    PRIMARY KEY (id),
     FOREIGN KEY(user_id) REFERENCES users (id)
)
()
COMMIT
```

2.12 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 *Custom 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(
```

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()
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
['jack', 'Jack Bean', 'gjffdd']
INSERT INTO addresses (email_address, user_id) VALUES (?, ?)
['jack@google.com', 5]
INSERT INTO addresses (email_address, user_id) VALUES (?, ?)
['j25@yahoo.com', 5]
COMMIT
```

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()
BEGIN
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name = ?
 LIMIT 2 OFFSET 0
['jack']>>> jack
<User('jack','Jack Bean', 'gjffdd')>
```

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

```
>>> jack.addresses
SELECT addresses.id AS addresses_id, addresses.email_address AS addresses_email_address, ad
FROM addresses
WHERE ? = addresses.user_id ORDER BY addresses.id
[5][<Address('jack@google.com')>, <Address('j25@yahoo.com')>]
```

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()
SELECT anon_1.users_id AS anon_1_users_id, anon_1.users_name AS anon_1_users_name,
anon_1.users_fullname AS anon_1_users_fullname, anon_1.users_password AS anon_1_users_passw
addresses_1.id AS addresses_1_id, addresses_1.email_address AS addresses_1_email_address,
addresses_1.user_id AS addresses_1_user_id
FROM (SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullna
users.password AS users_password
FROM users WHERE users.name = ?
LIMIT 2 OFFSET 0) AS anon_1 LEFT OUTER JOIN addresses AS addresses_1
ON anon_1.users_id = addresses_1.user_id ORDER BY addresses_1.id
['jack']>>> jack
<User('jack','Jack Bean', 'gjffdd')>

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

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 *advdatamapping_relation*.

2.13 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
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname,
users.password AS users_password, addresses.id AS addresses_id,
addresses.email_address AS addresses_email_address, addresses.user_id AS addresses_user_id
FROM users, addresses
WHERE users.id = addresses.user_id AND addresses.email_address = ?
['jack@google.com']<User('jack','Jack Bean', 'gjffdd')> <Address('jack@google.com')>
```

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()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users JOIN addresses ON users.id = addresses.user_id
WHERE addresses.email_address = ?
['jack@google.com'][<User('jack','Jack Bean', 'gjffdd')>]
```

`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()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users JOIN addresses ON users.id = addresses.user_id
WHERE addresses.email_address = ?
['jack@google.com'][<User('jack','Jack Bean', 'gjffdd')>]
```

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>`.

2.13.1 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 \
...     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
SELECT users.name AS users_name, addresses_1.email_address AS addresses_1_email_address,
addresses_2.email_address AS addresses_2_email_address
FROM users JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
JOIN addresses AS addresses_2 ON users.id = addresses_2.user_id
WHERE addresses_1.email_address = ? AND addresses_2.email_address = ?
['jack@google.com', 'j25@yahoo.com']jack jack@google.com j25@yahoo.com
```

2.13.2 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_cou
    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*:

```
>>> 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).\
...     outerjoin((stmt, User.id==stmt.c.user_id)).order_by(User.id):
...     print u, count
SELECT users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password,
anon_1.address_count AS anon_1_address_count
FROM users LEFT OUTER JOIN (SELECT addresses.user_id AS user_id, count(?) AS address_count
FROM addresses GROUP BY addresses.user_id) AS anon_1 ON users.id = anon_1.user_id
ORDER BY users.id
['*']<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
```

2.13.3 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
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname,
users.password AS users_password, anon_1.id AS anon_1_id,
anon_1.email_address AS anon_1_email_address, anon_1.user_id AS anon_1_user_id
FROM users JOIN (SELECT addresses.id AS id, addresses.email_address AS email_address, addre
FROM addresses
WHERE addresses.email_address != ?) AS anon_1 ON users.id = anon_1.user_id
['j25@yahoo.com']<User('jack','Jack Bean', 'gjffdd')> <Address('jack@google.com')>
```

2.13.4 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
SELECT users.name AS users_name
FROM users
WHERE EXISTS (SELECT *
FROM addresses
WHERE addresses.user_id = users.id)
[]jack
```

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
SELECT users.name AS users_name
FROM users
WHERE EXISTS (SELECT 1
FROM addresses
WHERE users.id = addresses.user_id)
[]jack
```

`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
SELECT users.name AS users_name
FROM users
WHERE EXISTS (SELECT 1
FROM addresses
WHERE users.id = addresses.user_id AND addresses.email_address LIKE ?)
['%google%']jack
```

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()
SELECT addresses.id AS addresses_id, addresses.email_address AS addresses_email_address,
addresses.user_id AS addresses_user_id
FROM addresses
WHERE NOT (EXISTS (SELECT 1
FROM users
WHERE users.id = addresses.user_id AND users.name = ?))
['jack'][]
```

2.13.5 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')
  ```

2.14 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()
UPDATE addresses SET user_id=? WHERE addresses.id = ?
[None, 1]
UPDATE addresses SET user_id=? WHERE addresses.id = ?
[None, 2]
DELETE FROM users WHERE users.id = ?
[5]
SELECT count(1) AS count_1
FROM users
WHERE users.name = ?
['jack']0
```

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()
SELECT count(1) AS count_1
FROM addresses
WHERE addresses.email_address IN (?, ?)
['jack@google.com', 'j25@yahoo.com']2
```

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.

2.14.1 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)
BEGIN
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.id = ?
[5]# remove one Address (lazy load fires off)
>>> del jack.addresses[1]
SELECT addresses.id AS addresses_id, addresses.email_address AS addresses_email_address, ad
FROM addresses
WHERE ? = addresses.user_id
[5]# only one address remains
>>> session.query(Address).filter(
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
DELETE FROM addresses WHERE addresses.id = ?
[2]
SELECT count(1) AS count_1
FROM addresses
WHERE addresses.email_address IN (?, ?)
['jack@google.com', 'j25@yahoo.com']1
```

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

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

>>> session.query(User).filter_by(name='jack').count()
DELETE FROM addresses WHERE addresses.id = ?
[1]
DELETE FROM users WHERE users.id = ?
[5]
SELECT count(1) AS count_1
FROM users
WHERE users.name = ?
['jack']0

>>> session.query(Address).filter(
...    Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
SELECT count(1) AS count_1
FROM addresses
WHERE addresses.email_address IN (?, ?)
['jack@google.com', 'j25@yahoo.com']0
```

2.15 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)
PRAGMA table_info("users")
()
PRAGMA table_info("addresses")
()
PRAGMA table_info("posts")
()
PRAGMA table_info("keywords")
()
PRAGMA table_info("post_keywords")
()
CREATE TABLE posts (
    id INTEGER NOT NULL,
    user_id INTEGER,
    headline VARCHAR(255) NOT NULL,
    body TEXT,
    PRIMARY KEY (id),
     FOREIGN KEY(user_id) REFERENCES users (id)
)
()
COMMIT
CREATE TABLE keywords (
    id INTEGER NOT NULL,
    keyword VARCHAR(50) NOT NULL,
    PRIMARY KEY (id),
     UNIQUE (keyword)
)
()
COMMIT
CREATE TABLE post_keywords (
    post_id INTEGER,
    keyword_id INTEGER,
     FOREIGN KEY(post_id) REFERENCES posts (id),
     FOREIGN KEY(keyword_id) REFERENCES keywords (id)
)
()
COMMIT
```

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()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
```

```
FROM users
WHERE users.name = ?
 LIMIT 2 OFFSET 0
['wendy']>>> 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()
INSERT INTO keywords (keyword) VALUES (?)
['wendy']
INSERT INTO keywords (keyword) VALUES (?)
['firstpost']
INSERT INTO posts (user_id, headline, body) VALUES (?, ?, ?)
[2, "Wendy's Blog Post", 'This is a test']
INSERT INTO post_keywords (post_id, keyword_id) VALUES (?, ?)
[[1, 2], [1, 1]]
SELECT posts.id AS posts_id, posts.user_id AS posts_user_id, posts.headline AS posts_headli
FROM posts
WHERE EXISTS (SELECT 1
FROM post_keywords, keywords
WHERE posts.id = post_keywords.post_id AND keywords.id = post_keywords.keyword_id AND keywo
['firstpost'][BlogPost("Wendy's Blog Post", 'This is a test', <User('wendy','Wendy Williams
```

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).\
... filter(BlogPost.keywords.any(keyword='firstpost')).all()
SELECT posts.id AS posts_id, posts.user_id AS posts_user_id, posts.headline AS posts_headli
FROM posts
WHERE ? = posts.user_id AND (EXISTS (SELECT 1
FROM post_keywords, keywords
WHERE posts.id = post_keywords.post_id AND keywords.id = post_keywords.keyword_id AND keywo
[2, 'firstpost'][BlogPost("Wendy's Blog Post", 'This is a test', <User('wendy','Wendy Willi
```

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()
SELECT posts.id AS posts_id, posts.user_id AS posts_user_id, posts.headline AS posts_headli
FROM posts
WHERE ? = posts.user_id AND (EXISTS (SELECT 1
FROM post_keywords, keywords
WHERE posts.id = post_keywords.post_id AND keywords.id = post_keywords.keyword_id AND keywo
[2, 'firstpost'][BlogPost("Wendy's Blog Post", 'This is a test', <User('wendy','Wendy Willi
```

2.16 Further Reference

Query Reference: *Querying*

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

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

CHAPTER

THREE

SQL EXPRESSION LANGUAGE TUTORIAL

This tutorial will cover SQLAlchemy SQL Expressions, which are Python constructs that represent SQL statements. 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. The tutorial has no prerequisites.

3.1 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
```

3.2 Connecting

For this tutorial we will use an in-memory-only SQLite database. This is an easy way to test things without needing to have an actual database defined anywhere. 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.

3.3 Define and Create Tables

The SQL Expression Language constructs its expressions in most cases against table columns. In SQLAlchemy, a column is most often represented by an object called `Column`, and in all cases a `Column` is associated with a `Table`. A collection of `Table` objects and their associated child objects is referred to as **database metadata**. In this tutorial we will explicitly lay out several `Table` objects, but note that SA can also "import" whole sets of `Table` objects automatically from an existing database (this process is called **table reflection**).

We define our tables all within a catalog called `MetaData`, using the `Table` construct, which resembles regular SQL CREATE TABLE statements. We'll make two tables, one of which represents "users" in an application, and another which represents zero or more "email addreses" for each row in the "users" table:

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

>>> addresses = Table('addresses', metadata,
...   Column('id', Integer, primary_key=True),
...   Column('user_id', None, ForeignKey('users.id')),
...   Column('email_address', String, nullable=False)
...  )
```

All about how to define `Table` objects, as well as how to create them from an existing database automatically, is described in *Database Meta Data*.

Next, to tell the `MetaData` we'd actually like to create our selection of tables for real inside the SQLite database, we use `create_all()`, passing it the `engine` instance which points to our database. This will check for the presence of each table first before creating, so it's safe to call multiple times:

```
>>> metadata.create_all(engine)
PRAGMA table_info("users")
{}
PRAGMA table_info("addresses")
{}
CREATE TABLE users (
    id INTEGER NOT NULL,
    name VARCHAR,
    fullname VARCHAR,
    PRIMARY KEY (id)
)
{}
COMMIT
CREATE TABLE addresses (
    id INTEGER NOT NULL,
    user_id INTEGER,
    email_address VARCHAR NOT NULL,
    PRIMARY KEY (id),
     FOREIGN KEY(user_id) REFERENCES users (id)
)
{}
COMMIT
```

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 fields available on `Integer`, `Numeric`, etc. are not referenced by SQLAlchemy other than when creating tables.

3.4 Insert Expressions

The first SQL expression we'll create is the `Insert` construct, which represents an INSERT statement. This is typically created relative to its target table:

```
>>> ins = users.insert()
```

To see a sample of the SQL this construct produces, use the `str()` function:

```
>>> str(ins)
'INSERT INTO users (id, name, fullname) VALUES (:id, :name, :fullname)'
```

Notice above that the INSERT statement names every column in the `users` table. This can be limited by using the `values()` method, which establishes the VALUES clause of the INSERT explicitly:

```
>>> ins = users.insert().values(name='jack', fullname='Jack Jones')
>>> str(ins)
'INSERT INTO users (name, fullname) VALUES (:name, :fullname)'
```

Above, while the `values` method limited the VALUES clause to just two columns, the actual data we placed in `values` didn't get rendered into the string; instead we got named bind parameters. As it turns out, our data *is* stored within our `Insert` construct, but it typically only comes out when the statement is actually executed; since the data consists of literal values, SQLAlchemy automatically generates bind parameters for them. We can peek at this data for now by looking at the compiled form of the statement:

```
>>> ins.compile().params
{'fullname': 'Jack Jones', 'name': 'jack'}
```

3.5 Executing

The interesting part of an `Insert` is executing it. In this tutorial, we will generally focus on the most explicit method of executing a SQL construct, and later touch upon some "shortcut" ways to do it. The `engine` object we created is a repository for database connections capable of issuing SQL to the database. To acquire a connection, we use the `connect()` method:

```
>>> conn = engine.connect()
>>> conn
<sqlalchemy.engine.base.Connection object at 0x...>
```

The `Connection` object represents an actively checked out DBAPI connection resource. Lets feed it our `Insert` object and see what happens:

```
>>> result = conn.execute(ins)
INSERT INTO users (name, fullname) VALUES (?, ?)
['jack', 'Jack Jones']
COMMIT
```

So the INSERT statement was now issued to the database. Although we got positional "qmark" bind parameters instead of "named" bind parameters in the output. How come ? Because when executed, the `Connection` used the SQLite **dialect** to help generate the statement; when we use the `str()` function, the statement isn't aware of this dialect, and falls back onto a default which uses named parameters. We can view this manually as follows:

```
>>> ins.bind = engine
>>> str(ins)
'INSERT INTO users (name, fullname) VALUES (?, ?)'
```

What about the `result` variable we got when we called `execute()` ? As the SQLAlchemy `Connection` object references a DBAPI connection, the result, known as a `ResultProxy` object, is analogous to the DBAPI cursor object. In the case of an INSERT, we can get important information from it, such as the primary key values which were generated from our statement:

```
>>> result.last_inserted_ids()
[1]
```

The value of `1` was automatically generated by SQLite, but only because we did not specify the `id` column in our `Insert` statement; otherwise, our explicit value would have been used. In either case, SQLAlchemy always knows how to get at a newly generated primary key value, even though the method of generating them is different across different databases; each databases' `Dialect` knows the specific steps needed to determine the correct value (or values; note that `last_inserted_ids()` returns a list so that it supports composite primary keys).

3.6 Executing Multiple Statements

Our insert example above was intentionally a little drawn out to show some various behaviors of expression language constructs. In the usual case, an `Insert` statement is usually compiled against the parameters sent to the `execute()` method on `Connection`, so that there's no need to use the `values` keyword with `Insert`. Lets create a generic `Insert` statement again and use it in the "normal" way:

```
>>> ins = users.insert()
>>> conn.execute(ins, id=2, name='wendy', fullname='Wendy Williams')
INSERT INTO users (id, name, fullname) VALUES (?, ?, ?)
[2, 'wendy', 'Wendy Williams']
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>
```

Above, because we specified all three columns in the the `execute()` method, the compiled `Insert` included all three columns. The `Insert` statement is compiled at execution time based on the parameters we specified; if we specified fewer parameters, the `Insert` would have fewer entries in its VALUES clause.

To issue many inserts using DBAPI's `executemany()` method, we can send in a list of dictionaries each containing a distinct set of parameters to be inserted, as we do here to add some email addresses:

```
>>> conn.execute(addresses.insert(), [
...    {'user_id': 1, 'email_address' : 'jack@yahoo.com'},
...    {'user_id': 1, 'email_address' : 'jack@msn.com'},
...    {'user_id': 2, 'email_address' : 'www@www.org'},
...    {'user_id': 2, 'email_address' : 'wendy@aol.com'},
... ])
INSERT INTO addresses (user_id, email_address) VALUES (?, ?)
[[1, 'jack@yahoo.com'], [1, 'jack@msn.com'], [2, 'www@www.org'], [2, 'wendy@aol.com']]
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>
```

Above, we again relied upon SQLite's automatic generation of primary key identifiers for each `addresses` row.

When executing multiple sets of parameters, each dictionary must have the **same** set of keys; i.e. you cant have fewer keys in some dictionaries than others. This is because the `Insert` statement is compiled against the **first** dictionary in the list, and it's assumed that all subsequent argument dictionaries are compatible with that statement.

3.7 Connectionless / Implicit Execution

We're executing our `Insert` using a `Connection`. There's two options that allow you to not have to deal with the connection part. You can execute in the **connectionless** style, using the engine, which opens and closes a connection for you:

```
>>> result = engine.execute(users.insert(), name='fred', fullname="Fred Flintstone")
INSERT INTO users (name, fullname) VALUES (?, ?)
['fred', 'Fred Flintstone']
COMMIT
```

and you can save even more steps than that, if you connect the `Engine` to the `MetaData` object we created earlier. When this is done, all SQL expressions which involve tables within the `MetaData` object will be automatically **bound** to the `Engine`. In this case, we call it **implicit execution**:

```
>>> metadata.bind = engine
>>> result = users.insert().execute(name="mary", fullname="Mary Contrary")
INSERT INTO users (name, fullname) VALUES (?, ?)
['mary', 'Mary Contrary']
COMMIT
```

When the `MetaData` is bound, statements will also compile against the engine's dialect. Since a lot of the examples here assume the default dialect, we'll detach the engine from the metadata which we just attached:

```
>>> metadata.bind = None
```

Detailed examples of connectionless and implicit execution are available in the "Engines" chapter: *dbengine_implicit*.

3.8 Selecting

We began with inserts just so that our test database had some data in it. The more interesting part of the data is selecting it ! We'll cover UPDATE and DELETE statements later. The primary construct used to generate SELECT statements is the `select()` function:

```
>>> from sqlalchemy.sql import select
>>> s = select([users])
>>> result = conn.execute(s)
SELECT users.id, users.name, users.fullname
FROM users
[]
```

Above, we issued a basic `select()` call, placing the `users` table within the COLUMNS clause of the select, and then executing. SQLAlchemy expanded the `users` table into the set of each of its columns, and also generated a FROM clause for us. The result returned is again a `ResultProxy` object, which acts much like a DBAPI cursor, including methods such as `fetchone()` and `fetchall()`. The easiest way to get rows from it is to just iterate:

```
>>> for row in result:
...     print row
(1, u'jack', u'Jack Jones')
(2, u'wendy', u'Wendy Williams')
(3, u'fred', u'Fred Flintstone')
(4, u'mary', u'Mary Contrary')
```

Above, we see that printing each row produces a simple tuple-like result. We have more options at accessing the data in each row. One very common way is through dictionary access, using the string names of columns:

```
>>> result = conn.execute(s)
SELECT users.id, users.name, users.fullname
FROM users
[]>>> row = result.fetchone()
>>> print "name:", row['name'], "; fullname:", row['fullname']
name: jack ; fullname: Jack Jones
```

Integer indexes work as well:

```
>>> row = result.fetchone()
>>> print "name:", row[1], "; fullname:", row[2]
name: wendy ; fullname: Wendy Williams
```

But another way, whose usefulness will become apparent later on, is to use the `Column` objects directly as keys:

```
>>> for row in conn.execute(s):
...     print "name:", row[users.c.name], "; fullname:", row[users.c.fullname]
SELECT users.id, users.name, users.fullname
FROM users
[]name: jack ; fullname: Jack Jones
name: wendy ; fullname: Wendy Williams
name: fred ; fullname: Fred Flintstone
name: mary ; fullname: Mary Contrary
```

Result sets which have pending rows remaining should be explicitly closed before discarding. While the resources referenced by the `ResultProxy` will be closed when the object is garbage collected, it's better to make it explicit as some database APIs are very picky about such things:

```
>>> result.close()
```

If we'd like to more carefully control the columns which are placed in the COLUMNS clause of the select, we reference individual `Column` objects from our `Table`. These are available as named attributes off the `c` attribute of the `Table` object:

```
>>> s = select([users.c.name, users.c.fullname])
>>> result = conn.execute(s)
SELECT users.name, users.fullname
FROM users
[]>>> for row in result:
...     print row
(u'jack', u'Jack Jones')
(u'wendy', u'Wendy Williams')
(u'fred', u'Fred Flintstone')
(u'mary', u'Mary Contrary')
```

Lets observe something interesting about the FROM clause. Whereas the generated statement contains two distinct sections, a "SELECT columns" part and a "FROM table" part, our `select()` construct only has a list containing columns. How does this work ? Let's try putting *two* tables into our `select()` statement:

```
>>> for row in conn.execute(select([users, addresses])):
...     print row
SELECT users.id, users.name, users.fullname, addresses.id, addresses.user_id, addresses.ema
FROM users, addresses
[](1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com')
(1, u'jack', u'Jack Jones', 2, 1, u'jack@msn.com')
(1, u'jack', u'Jack Jones', 3, 2, u'www@www.org')
(1, u'jack', u'Jack Jones', 4, 2, u'wendy@aol.com')
(2, u'wendy', u'Wendy Williams', 1, 1, u'jack@yahoo.com')
(2, u'wendy', u'Wendy Williams', 2, 1, u'jack@msn.com')
(2, u'wendy', u'Wendy Williams', 3, 2, u'www@www.org')
(2, u'wendy', u'Wendy Williams', 4, 2, u'wendy@aol.com')
(3, u'fred', u'Fred Flintstone', 1, 1, u'jack@yahoo.com')
(3, u'fred', u'Fred Flintstone', 2, 1, u'jack@msn.com')
(3, u'fred', u'Fred Flintstone', 3, 2, u'www@www.org')
(3, u'fred', u'Fred Flintstone', 4, 2, u'wendy@aol.com')
(4, u'mary', u'Mary Contrary', 1, 1, u'jack@yahoo.com')
(4, u'mary', u'Mary Contrary', 2, 1, u'jack@msn.com')
(4, u'mary', u'Mary Contrary', 3, 2, u'www@www.org')
(4, u'mary', u'Mary Contrary', 4, 2, u'wendy@aol.com')
```

It placed **both** tables into the FROM clause. But also, it made a real mess. Those who are familiar with SQL joins know that this is a **Cartesian product**; each row from the `users` table is produced against each row from the `addresses` table. So to put some sanity into this statement, we need a WHERE clause. Which brings us to the second argument of `select()`:

```
>>> s = select([users, addresses], users.c.id==addresses.c.user_id)
>>> for row in conn.execute(s):
...     print row
SELECT users.id, users.name, users.fullname, addresses.id, addresses.user_id, addresses.ema
FROM users, addresses
WHERE users.id = addresses.user_id
[](1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com')
(1, u'jack', u'Jack Jones', 2, 1, u'jack@msn.com')
(2, u'wendy', u'Wendy Williams', 3, 2, u'www@www.org')
(2, u'wendy', u'Wendy Williams', 4, 2, u'wendy@aol.com')
```

So that looks a lot better, we added an expression to our `select()` which had the effect of adding `WHERE users.id = addresses.user_id` to our statement, and our results were managed down so that the join of `users` and `addresses` rows made sense. But let's look at that expression? It's using just a Python equality operator between two different `Column` objects. It should be clear that something is up. Saying `1==1` produces `True`, and `1==2` produces `False`, not a WHERE clause. So lets see exactly what that expression is doing:

```
>>> users.c.id==addresses.c.user_id
<sqlalchemy.sql.expression._BinaryExpression object at 0x...>
```

Wow, surprise ! This is neither a `True` nor a `False`. Well what is it ?

```
>>> str(users.c.id==addresses.c.user_id)
'users.id = addresses.user_id'
```

As you can see, the == operator is producing an object that is very much like the `Insert` and `select()` objects we've made so far, thanks to Python's `__eq__()` builtin; you call `str()` on it and it produces SQL. By now, one can that everything we are working with is ultimately the same type of object. SQLAlchemy terms the base class of all of these expressions as `sqlalchemy.sql.ClauseElement`.

3.9 Operators

Since we've stumbled upon SQLAlchemy's operator paradigm, let's go through some of its capabilities. We've seen how to equate two columns to each other:

```
>>> print users.c.id==addresses.c.user_id
users.id = addresses.user_id
```

If we use a literal value (a literal meaning, not a SQLAlchemy clause object), we get a bind parameter:

```
>>> print users.c.id==7
users.id = :id_1
```

The `7` literal is embedded in `ClauseElement`; we can use the same trick we did with the `Insert` object to see it:

```
>>> (users.c.id==7).compile().params
{u'id_1': 7}
```

Most Python operators, as it turns out, produce a SQL expression here, like equals, not equals, etc.:

```
>>> print users.c.id != 7
users.id != :id_1

>>> # None converts to IS NULL
>>> print users.c.name == None
users.name IS NULL

>>> # reverse works too
>>> print 'fred' > users.c.name
users.name < :name_1
```

If we add two integer columns together, we get an addition expression:

```
>>> print users.c.id + addresses.c.id
users.id + addresses.id
```

Interestingly, the type of the `Column` is important ! If we use `+` with two string based columns (recall we put types like `Integer` and `String` on our `Column` objects at the beginning), we get something different:

```
>>> print users.c.name + users.c.fullname
users.name || users.fullname
```

Where `||` is the string concatenation operator used on most databases. But not all of them. MySQL users, fear not:

```
>>> print (users.c.name + users.c.fullname).compile(bind=create_engine('mysql://'))
concat(users.name, users.fullname)
```

The above illustrates the SQL that's generated for an `Engine` that's connected to a MySQL database; the `||` operator now compiles as MySQL's `concat()` function.

If you have come across an operator which really isn't available, you can always use the `op()` method; this generates whatever operator you need:

```
>>> print users.c.name.op('tiddlywinks')('foo')
users.name tiddlywinks :name_1
```

3.10 Conjunctions

We'd like to show off some of our operators inside of `select()` constructs. But we need to lump them together a little more, so let's first introduce some conjunctions. Conjunctions are those little words like AND and OR that put things together. We'll also hit upon NOT. AND, OR and NOT can work from the corresponding functions SQLAlchemy provides (notice we also throw in a LIKE):

```
>>> from sqlalchemy.sql import and_, or_, not_
>>> print and_(users.c.name.like('j%'), users.c.id==addresses.c.user_id,
...     or_(addresses.c.email_address=='wendy@aol.com', addresses.c.email_address=='jack@ya
...     not_(users.c.id>5))
users.name LIKE :name_1 AND users.id = addresses.user_id AND
(addresses.email_address = :email_address_1 OR addresses.email_address = :email_address_2)
AND users.id <= :id_1
```

And you can also use the re-jiggered bitwise AND, OR and NOT operators, although because of Python operator precedence you have to watch your parenthesis:

```
>>> print users.c.name.like('j%') & (users.c.id==addresses.c.user_id) &  \
...     ((addresses.c.email_address=='wendy@aol.com') | (addresses.c.email_address=='jack@y
...     & ~(users.c.id>5)
users.name LIKE :name_1 AND users.id = addresses.user_id AND
(addresses.email_address = :email_address_1 OR addresses.email_address = :email_address_2)
AND users.id <= :id_1
```

So with all of this vocabulary, let's select all users who have an email address at AOL or MSN, whose name starts with a letter between "m" and "z", and we'll also generate a column containing their full name combined with their email address. We will add two new constructs to this statement, `between()` and `label()`. `between()` produces a BETWEEN clause, and `label()` is used in a column expression to produce labels using the `AS` keyword; it's recommended when selecting from expressions that otherwise would not have a name:

```
>>> s = select([(users.c.fullname + ", " + addresses.c.email_address).label('title')],
...        and_(
...            users.c.id==addresses.c.user_id,
...            users.c.name.between('m', 'z'),
...           or_(
...              addresses.c.email_address.like('%@aol.com'),
...              addresses.c.email_address.like('%@msn.com')
...           )
...         )
...     )
>>> print conn.execute(s).fetchall()
SELECT users.fullname || ? || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN ? AND ? AND
(addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
[', ', 'm', 'z', '%@aol.com', '%@msn.com']
[(u'Wendy Williams, wendy@aol.com',)]
```

Once again, SQLAlchemy figured out the FROM clause for our statement. In fact it will determine the FROM clause based on all of its other bits; the columns clause, the where clause, and also some other elements which we haven't covered yet, which include ORDER BY, GROUP BY, and HAVING.

3.11 Using Text

Our last example really became a handful to type. Going from what one understands to be a textual SQL expression into a Python construct which groups components together in a programmatic style can be hard. That's why SQLAlchemy lets you just use strings too. The `text()` construct represents any textual statement. To use bind parameters with `text()`, always use the named colon format. Such as below, we create a `text()` and execute it, feeding in the bind parameters to the `execute()` method:

```
>>> from sqlalchemy.sql import text
>>> s = text("""SELECT users.fullname || ', ' || addresses.email_address AS title
...             FROM users, addresses
...             WHERE users.id = addresses.user_id AND users.name BETWEEN :x AND :y AND
...             (addresses.email_address LIKE :e1 OR addresses.email_address LIKE :e2)
...         """)
>>> print conn.execute(s, x='m', y='z', e1='%@aol.com', e2='%@msn.com').fetchall()
SELECT users.fullname || ', ' || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN ? AND ? AND
(addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
['m', 'z', '%@aol.com', '%@msn.com'][(u'Wendy Williams, wendy@aol.com',)]
```

To gain a "hybrid" approach, any of SA's SQL constructs can have text freely intermingled wherever you like - the `text()` construct can be placed within any other `ClauseElement` construct, and when used in a non-operator context, a direct string may be placed which converts to `text()` automatically. Below we combine the usage of `text()` and strings with our constructed `select()` object, by using the `select()` object to structure the statement, and the `text()`/strings to provide all the content within the structure. For this example, SQLAlchemy is not given any `Column` or `Table` objects in any of its expressions, so it cannot generate a FROM clause. So we also give it the `from_obj` keyword argument, which is a list of `ClauseElements` (or strings) to be placed within the FROM clause:

```
>>> s = select([text("users.fullname || ', ' || addresses.email_address AS title")],
...         and_(
...             "users.id = addresses.user_id",
...              "users.name BETWEEN 'm' AND 'z'",
...              "(addresses.email_address LIKE :x OR addresses.email_address LIKE :y)"
...         ),
...          from_obj=['users', 'addresses']
...     )
```

```
>>> print conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()
SELECT users.fullname || ', ' || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN 'm' AND 'z' AND (addresses.email_
['%@aol.com', '%@msn.com'][(u'Wendy Williams, wendy@aol.com',)]
```

Going from constructed SQL to text, we lose some capabilities. We lose the capability for SQLAlchemy to compile our expression to a specific target database; above, our expression won't work with MySQL since it has no `||` construct. It also becomes more tedious for SQLAlchemy to be made aware of the datatypes in use; for example, if our bind parameters required UTF-8 encoding before going in, or conversion from a Python `datetime` into a string (as is required with SQLite), we would have to add extra information to our `text()` construct. Similar issues arise on the result set side, where SQLAlchemy also performs type-specific data conversion in some cases; still more information can be added to `text()` to work around this. But what we really lose from our statement is the ability to manipulate it, transform it, and analyze it. These features are critical when using the ORM, which makes heavy usage of relational transformations. To show off what we mean, we'll first introduce the ALIAS construct and the JOIN construct, just so we have some juicier bits to play with.

3.12 Using Aliases

The alias corresponds to a "renamed" version of a table or arbitrary relation, which occurs anytime you say "SELECT .. FROM sometable AS someothername". The `AS` creates a new name for the table. Aliases are super important in SQL as they allow you to reference the same table more than once. Scenarios where you need to do this include when you self-join a table to itself, or more commonly when you need to join from a parent table to a child table multiple times. For example, we know that our user `jack` has two email addresses. How can we locate jack based on the combination of those two addresses? We need to join twice to it. Let's construct two distinct aliases for the `addresses` table and join:

```
>>> a1 = addresses.alias('a1')
>>> a2 = addresses.alias('a2')
>>> s = select([users], and_(
...         users.c.id==a1.c.user_id,
...         users.c.id==a2.c.user_id,
...         a1.c.email_address=='jack@msn.com',
...         a2.c.email_address=='jack@yahoo.com'
...    ))
>>> print conn.execute(s).fetchall()
SELECT users.id, users.name, users.fullname
FROM users, addresses AS a1, addresses AS a2
WHERE users.id = a1.user_id AND users.id = a2.user_id AND a1.email_address = ? AND a2.emai
['jack@msn.com', 'jack@yahoo.com'][(1, u'jack', u'Jack Jones')]
```

Easy enough. One thing that we're going for with the SQL Expression Language is the melding of programmatic behavior with SQL generation. Coming up with names like `a1` and `a2` is messy; we really didn't need to use those names anywhere, it's just the database that needed them. Plus, we might write some code that uses alias objects that came from several different places, and it's difficult to ensure that they all have unique names. So instead, we just let SQLAlchemy make the names for us, using "anonymous" aliases:

```
>>> a1 = addresses.alias()
>>> a2 = addresses.alias()
>>> s = select([users], and_(
...         users.c.id==a1.c.user_id,
...         users.c.id==a2.c.user_id,
...         a1.c.email_address=='jack@msn.com',
...         a2.c.email_address=='jack@yahoo.com'
...    ))
>>> print conn.execute(s).fetchall()
SELECT users.id, users.name, users.fullname
FROM users, addresses AS addresses_1, addresses AS addresses_2
WHERE users.id = addresses_1.user_id AND users.id = addresses_2.user_id AND addresses_1.ema
['jack@msn.com', 'jack@yahoo.com'][(1, u'jack', u'Jack Jones')]
```

One super-huge advantage of anonymous aliases is that not only did we not have to guess up a random name, but we can also be guaranteed that the above SQL string is **deterministically** generated to be the same every time. This is important for databases such as Oracle which cache compiled "query plans" for their statements, and need to see the same SQL string in order to make use of it.

Aliases can of course be used for anything which you can SELECT from, including SELECT statements themselves. We can self-join the `users` table back to the `select()` we've created by making an alias of the entire statement. The `correlate(None)` directive is to avoid SQLAlchemy's attempt to "correlate" the inner `users` table with the outer one:

```
>>> a1 = s.correlate(None).alias()
>>> s = select([users.c.name], users.c.id==a1.c.id)
>>> print conn.execute(s).fetchall()
SELECT users.name
FROM users, (SELECT users.id AS id, users.name AS name, users.fullname AS fullname
FROM users, addresses AS addresses_1, addresses AS addresses_2
WHERE users.id = addresses_1.user_id AND users.id = addresses_2.user_id AND addresses_1.ema
WHERE users.id = anon_1.id
['jack@msn.com', 'jack@yahoo.com'][(u'jack',)]
```

3.13 Using Joins

We're halfway along to being able to construct any SELECT expression. The next cornerstone of the SELECT is the JOIN expression. We've already been doing joins in our examples, by just placing two tables in either the columns clause or the where clause of the `select()` construct. But if we want to make a real "JOIN" or "OUTERJOIN" construct, we use the `join()` and `outerjoin()` methods, most commonly accessed from the left table in the join:

```
>>> print users.join(addresses)
users JOIN addresses ON users.id = addresses.user_id
```

The alert reader will see more surprises; SQLAlchemy figured out how to JOIN the two tables ! The ON condition of the join, as it's called, was automatically generated based on the `ForeignKey` object which we placed on the `addresses` table way at the beginning of this tutorial. Already the `join()` construct is looking like a much better way to join tables.

Of course you can join on whatever expression you want, such as if we want to join on all users who use the same name in their email address as their username:

```
>>> print users.join(addresses, addresses.c.email_address.like(users.c.name + '%'))
users JOIN addresses ON addresses.email_address LIKE users.name || :name_1
```

When we create a `select()` construct, SQLAlchemy looks around at the tables we've mentioned and then places them in the FROM clause of the statement. When we use JOINs however, we know what FROM clause we want, so here we make usage of the `from_obj` keyword argument:

```
>>> s = select([users.c.fullname], from_obj=[
...    users.join(addresses, addresses.c.email_address.like(users.c.name + '%'))
...    ])
>>> print conn.execute(s).fetchall()
SELECT users.fullname
FROM users JOIN addresses ON addresses.email_address LIKE users.name || ?
['%'][(u'Jack Jones',), (u'Jack Jones',), (u'Wendy Williams',)]
```

The `outerjoin()` function just creates `LEFT OUTER JOIN` constructs. It's used just like `join()`:

```
>>> s = select([users.c.fullname], from_obj=[users.outerjoin(addresses)])
>>> print s
SELECT users.fullname
FROM users LEFT OUTER JOIN addresses ON users.id = addresses.user_id
```

That's the output `outerjoin()` produces, unless, of course, you're stuck in a gig using Oracle prior to version 9, and you've set up your engine (which would be using `OracleDialect`) to use Oracle-specific SQL:

```
>>> from sqlalchemy.databases.oracle import OracleDialect
>>> print s.compile(dialect=OracleDialect(use_ansi=False))
SELECT users.fullname
FROM users, addresses
WHERE users.id = addresses.user_id(+)
```

If you don't know what that SQL means, don't worry ! The secret tribe of Oracle DBAs don't want their black magic being found out ;).

3.14 Intro to Generative Selects and Transformations

We've now gained the ability to construct very sophisticated statements. We can use all kinds of operators, table constructs, text, joins, and aliases. The point of all of this, as mentioned earlier, is not that it's an "easier" or "better" way to write SQL than just writing a SQL statement yourself; the point is that it's better for writing *programmatically generated* SQL which can be morphed and adapted as needed in automated scenarios.

To support this, the `select()` construct we've been working with supports piecemeal construction, in addition to the "all at once" method we've been doing. Suppose you're writing a search function, which receives criterion and then must construct a select from it. To accomplish this, upon each criterion encountered, you apply "generative" criterion to an existing `select()` construct with new elements, one at a time. We start with a basic `select()` constructed with the shortcut method available on the `users` table:

```
>>> query = users.select()
>>> print query
SELECT users.id, users.name, users.fullname
FROM users
```

We encounter search criterion of "name='jack'". So we apply WHERE criterion stating such:

```
>>> query = query.where(users.c.name=='jack')
```

Next, we encounter that they'd like the results in descending order by full name. We apply ORDER BY, using an extra modifier `desc`:

```
>>> query = query.order_by(users.c.fullname.desc())
```

We also come across that they'd like only users who have an address at MSN. A quick way to tack this on is by using an EXISTS clause, which we correlate to the `users` table in the enclosing SELECT:

```
>>> from sqlalchemy.sql import exists
>>> query = query.where(
...    exists([addresses.c.id],
...        and_(addresses.c.user_id==users.c.id, addresses.c.email_address.like('%@msn.com'
...    ).correlate(users))
```

And finally, the application also wants to see the listing of email addresses at once; so to save queries, we outerjoin the `addresses` table (using an outer join so that users with no addresses come back as well; since we're programmatic, we might not have kept track that we used an EXISTS clause against the `addresses` table too...). Additionally, since the `users` and `addresses` table both have a column named `id`, let's isolate their names from each other in the COLUMNS clause by using labels:

```
>>> query = query.column(addresses).select_from(users.outerjoin(addresses)).apply_labels()
```

Let's bake for .0001 seconds and see what rises:

```
>>> conn.execute(query).fetchall()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, ad
FROM users LEFT OUTER JOIN addresses ON users.id = addresses.user_id
WHERE users.name = ? AND (EXISTS (SELECT addresses.id
FROM addresses
WHERE addresses.user_id = users.id AND addresses.email_address LIKE ?)) ORDER BY users.full
['jack', '%@msn.com'][(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com'), (1, u'jack', u'
```

So we started small, added one little thing at a time, and at the end we have a huge statement..which actually works. Now let's do one more thing; the searching function wants to add another `email_address` criterion on, however it doesn't want to construct an alias of the `addresses` table; suppose many parts of the application are written to deal specifically with the `addresses` table, and to change all those functions to support receiving an arbitrary alias of the address would be cumbersome. We can actually *convert* the `addresses` table within the *existing* statement to be an alias of itself, using `replace_selectable()`:

```
>>> a1 = addresses.alias()
>>> query = query.replace_selectable(addresses, a1)
>>> print query
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, ad
FROM users LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = :name_1 AND (EXISTS (SELECT addresses_1.id
FROM addresses AS addresses_1
WHERE addresses_1.user_id = users.id AND addresses_1.email_address LIKE :email_address_1))
```

One more thing though, with automatic labeling applied as well as anonymous aliasing, how do we retrieve the columns from the rows for this thing ? The label for the `email_addresses` column is now the generated name `addresses_1_email_address`; and in another statement might be something different ! This is where accessing by result columns by `Column` object becomes very useful:

```
>>> for row in conn.execute(query):
...     print "Name:", row[users.c.name], "; Email Address", row[a1.c.email_address]
```

```
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, ad
FROM users LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = ? AND (EXISTS (SELECT addresses_1.id
FROM addresses AS addresses_1
WHERE addresses_1.user_id = users.id AND addresses_1.email_address LIKE ?)) ORDER BY users
['jack', '%@msn.com']Name: jack ; Email Address jack@yahoo.com
Name: jack ; Email Address jack@msn.com
```

The above example, by its end, got significantly more intense than the typical end-user constructed SQL will usually be. However when writing higher-level tools such as ORMs, they become more significant. SQLAlchemy's ORM relies very heavily on techniques like this.

3.15 Everything Else

The concepts of creating SQL expressions have been introduced. What's left are more variants of the same themes. So now we'll catalog the rest of the important things we'll need to know.

3.15.1 Bind Parameter Objects

Throughout all these examples, SQLAlchemy is busy creating bind parameters wherever literal expressions occur. You can also specify your own bind parameters with your own names, and use the same statement repeatedly. The database dialect converts to the appropriate named or positional style, as here where it converts to positional for SQLite:

```
>>> from sqlalchemy.sql import bindparam
>>> s = users.select(users.c.name==bindparam('username'))
>>> conn.execute(s, username='wendy').fetchall()
SELECT users.id, users.name, users.fullname
FROM users
WHERE users.name = ?
['wendy'][(2, u'wendy', u'Wendy Williams')]
```

Another important aspect of bind parameters is that they may be assigned a type. The type of the bind parameter will determine its behavior within expressions and also how the data bound to it is processed before being sent off to the database:

```
>>> s = users.select(users.c.name.like(bindparam('username', type_=String) + text("'%'")))
>>> conn.execute(s, username='wendy').fetchall()
SELECT users.id, users.name, users.fullname
FROM users
WHERE users.name LIKE ? || '%'
['wendy'][(2, u'wendy', u'Wendy Williams')]
```

Bind parameters of the same name can also be used multiple times, where only a single named value is needed in the execute parameters:

```
>>> s = select([users, addresses],
...    users.c.name.like(bindparam('name', type_=String) + text("'%'")) |
...    addresses.c.email_address.like(bindparam('name', type_=String) + text("'@%'")),
...    from_obj=[users.outerjoin(addresses)])
>>> conn.execute(s, name='jack').fetchall()
```

```
SELECT users.id, users.name, users.fullname, addresses.id, addresses.user_id, addresses.ema
FROM users LEFT OUTER JOIN addresses ON users.id = addresses.user_id
WHERE users.name LIKE ? || '%' OR addresses.email_address LIKE ? || '@%'
['jack', 'jack'][(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com'), (1, u'jack', u'Jack
```

3.15.2 Functions

SQL functions are created using the `func` keyword, which generates functions using attribute access:

```
>>> from sqlalchemy.sql import func
>>> print func.now()
now()

>>> print func.concat('x', 'y')
concat(:param_1, :param_2)
```

Certain functions are marked as "ANSI" functions, which mean they don't get the parenthesis added after them, such as CURRENT_TIMESTAMP:

```
>>> print func.current_timestamp()
CURRENT_TIMESTAMP
```

Functions are most typically used in the columns clause of a select statement, and can also be labeled as well as given a type. Labeling a function is recommended so that the result can be targeted in a result row based on a string name, and assigning it a type is required when you need result-set processing to occur, such as for Unicode conversion and date conversions. Below, we use the result function `scalar()` to just read the first column of the first row and then close the result; the label, even though present, is not important in this case:

```
>>> print conn.execute(
...     select([func.max(addresses.c.email_address, type_=String).label('maxemail')])
... ).scalar()
SELECT max(addresses.email_address) AS maxemail
FROM addresses
[]www@www.org
```

Databases such as PostgreSQL and Oracle which support functions that return whole result sets can be assembled into selectable units, which can be used in statements. Such as, a database function `calculate()` which takes the parameters `x` and `y`, and returns three columns which we'd like to name `q`, `z` and `r`, we can construct using "lexical" column objects as well as bind parameters:

```
>>> from sqlalchemy.sql import column
>>> calculate = select([column('q'), column('z'), column('r')],
...     from_obj=[func.calculate(bindparam('x'), bindparam('y'))])

>>> print select([users], users.c.id > calculate.c.z)
SELECT users.id, users.name, users.fullname
FROM users, (SELECT q, z, r
FROM calculate(:x, :y))
WHERE users.id > z
```

If we wanted to use our `calculate` statement twice with different bind parameters, the `unique_params()` function will create copies for us, and mark the bind parameters as "unique" so that conflicting names are isolated. Note we also make two separate aliases of our selectable:

```
>>> s = select([users], users.c.id.between(
...     calculate.alias('c1').unique_params(x=17, y=45).c.z,
...     calculate.alias('c2').unique_params(x=5, y=12).c.z))

>>> print s
SELECT users.id, users.name, users.fullname
FROM users, (SELECT q, z, r
FROM calculate(:x_1, :y_1)) AS c1, (SELECT q, z, r
FROM calculate(:x_2, :y_2)) AS c2
WHERE users.id BETWEEN c1.z AND c2.z

>>> s.compile().params
{u'x_2': 5, u'y_2': 12, u'y_1': 45, u'x_1': 17}
```

See also `sqlalchemy.sql.expression.func`.

3.15.3 Unions and Other Set Operations

Unions come in two flavors, UNION and UNION ALL, which are available via module level functions:

```
>>> from sqlalchemy.sql import union
>>> u = union(
...      addresses.select(addresses.c.email_address=='foo@bar.com'),
...     addresses.select(addresses.c.email_address.like('%@yahoo.com')),
... ).order_by(addresses.c.email_address)

>>> print conn.execute(u).fetchall()
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address = ? UNION SELECT addresses.id, addresses.user_id, addresses.e
FROM addresses
WHERE addresses.email_address LIKE ? ORDER BY addresses.email_address
['foo@bar.com', '%@yahoo.com'][(1, 1, u'jack@yahoo.com')]
```

Also available, though not supported on all databases, are `intersect()`, `intersect_all()`, `except_()`, and `except_all()`:

```
>>> from sqlalchemy.sql import except_
>>> u = except_(
...     addresses.select(addresses.c.email_address.like('%@%.com')),
...     addresses.select(addresses.c.email_address.like('%@msn.com'))
... )

>>> print conn.execute(u).fetchall()
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address LIKE ? EXCEPT SELECT addresses.id, addresses.user_id, address
FROM addresses
WHERE addresses.email_address LIKE ?
['%@%.com', '%@msn.com'][(1, 1, u'jack@yahoo.com'), (4, 2, u'wendy@aol.com')]
```

3.15.4 Scalar Selects

To embed a SELECT in a column expression, use `as_scalar()`:

```
>>> print conn.execute(select([
...       users.c.name,
...       select([func.count(addresses.c.id)], users.c.id==addresses.c.user_id).as_scalar(
...    ])).fetchall()
SELECT users.name, (SELECT count(addresses.id) AS count_1
FROM addresses
WHERE users.id = addresses.user_id) AS anon_1
FROM users
[][(u'jack', 2), (u'wendy', 2), (u'fred', 0), (u'mary', 0)]
```

Alternatively, applying a `label()` to a select evaluates it as a scalar as well:

```
>>> print conn.execute(select([
...       users.c.name,
...       select([func.count(addresses.c.id)], users.c.id==addresses.c.user_id).label('add
...    ])).fetchall()
SELECT users.name, (SELECT count(addresses.id) AS count_1
FROM addresses
WHERE users.id = addresses.user_id) AS address_count
FROM users
[][(u'jack', 2), (u'wendy', 2), (u'fred', 0), (u'mary', 0)]
```

3.15.5 Correlated Subqueries

Notice in the examples on "scalar selects", the FROM clause of each embedded select did not contain the `users` table in its FROM clause. This is because SQLAlchemy automatically attempts to correlate embedded FROM objects to that of an enclosing query. To disable this, or to specify explicit FROM clauses to be correlated, use `correlate()`:

```
>>> s = select([users.c.name], users.c.id==select([users.c.id]).correlate(None))
>>> print s
SELECT users.name
FROM users
WHERE users.id = (SELECT users.id
FROM users)

>>> s = select([users.c.name, addresses.c.email_address], users.c.id==
...        select([users.c.id], users.c.id==addresses.c.user_id).correlate(addresses)
...    )
>>> print s
SELECT users.name, addresses.email_address
FROM users, addresses
WHERE users.id = (SELECT users.id
FROM users
WHERE users.id = addresses.user_id)
```

3.15.6 Ordering, Grouping, Limiting, Offset...ing...

The `select()` function can take keyword arguments `order_by`, `group_by` (as well as `having`), `limit`, and `offset`. There's also `distinct=True`. These are all also available as generative functions. `order_by()`

expressions can use the modifiers `asc()` or `desc()` to indicate ascending or descending.

```
>>> s = select([addresses.c.user_id, func.count(addresses.c.id)]).\
...     group_by(addresses.c.user_id).having(func.count(addresses.c.id)>1)
>>> print conn.execute(s).fetchall()
SELECT addresses.user_id, count(addresses.id) AS count_1
FROM addresses GROUP BY addresses.user_id
HAVING count(addresses.id) > ?
[1][(1, 2), (2, 2)]

>>> s = select([addresses.c.email_address, addresses.c.id]).distinct().\
...     order_by(addresses.c.email_address.desc(), addresses.c.id)
>>> conn.execute(s).fetchall()
SELECT DISTINCT addresses.email_address, addresses.id
FROM addresses ORDER BY addresses.email_address DESC, addresses.id
[][(u'www@www.org', 3), (u'wendy@aol.com', 4), (u'jack@yahoo.com', 1), (u'jack@msn.com', 2

>>> s = select([addresses]).offset(1).limit(1)
>>> print conn.execute(s).fetchall()
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
LIMIT 1 OFFSET 1
[][(2, 1, u'jack@msn.com')]
```

3.16 Updates

Finally, we're back to UPDATE. Updates work a lot like INSERTS, except there is an additional WHERE clause that can be specified.

```
>>> # change 'jack' to 'ed'
>>> conn.execute(users.update().where(users.c.name=='jack').values(name='ed'))
UPDATE users SET name=? WHERE users.name = ?
['ed', 'jack']
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>

>>> # use bind parameters
>>> u = users.update().where(users.c.name==bindparam('oldname')).values(name=bindparam('ne
>>> conn.execute(u, oldname='jack', newname='ed')
UPDATE users SET name=? WHERE users.name = ?
['ed', 'jack']
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>

>>> # update a column to an expression.  Send a dictionary to values():
>>> conn.execute(users.update().values({users.c.fullname:"Fullname: " + users.c.name}))
UPDATE users SET fullname=(? || users.name)
['Fullname: ']
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>
```

3.16.1 Correlated Updates

A correlated update lets you update a table using selection from another table, or the same table:

```
>>> s = select([addresses.c.email_address], addresses.c.user_id==users.c.id).limit(1)
>>> conn.execute(users.update().values({users.c.fullname:s}))
UPDATE users SET fullname=(SELECT addresses.email_address
FROM addresses
WHERE addresses.user_id = users.id
LIMIT 1 OFFSET 0)
[]
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>
```

3.17 Deletes

Finally, a delete. Easy enough:

```
>>> conn.execute(addresses.delete())
DELETE FROM addresses
[]
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>

>>> conn.execute(users.delete().where(users.c.name > 'm'))
DELETE FROM users WHERE users.name > ?
['m']
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>
```

3.18 Further Reference

API docs: `sqlalchemy.sql.expression`

Table Metadata Reference: *Database Meta Data*

Engine/Connection/Execution Reference: *Database Engines*

SQL Types: *Column and Data Types*

CHAPTER

FOUR

MAPPER CONFIGURATION

This section references most major configurational patterns involving the `mapper()` and `relation()` functions. It assumes you've worked through *Object Relational Tutorial* and know how to construct and use rudimentary mappers and relations.

4.1 Mapper Configuration

4.1.1 Customizing Column Properties

The default behavior of a `mapper` is to assemble all the columns in the mapped `Table` into mapped object attributes. This behavior can be modified in several ways, as well as enhanced by SQL expressions.

To load only a part of the columns referenced by a table as attributes, use the `include_properties` and `exclude_properties` arguments:

```
mapper(User, users_table, include_properties=['user_id', 'user_name'])

mapper(Address, addresses_table, exclude_properties=['street', 'city', 'state', 'zip'])
```

To change the name of the attribute mapped to a particular column, place the `Column` object in the `properties` dictionary with the desired key:

```
mapper(User, users_table, properties={
   'id': users_table.c.user_id,
   'name': users_table.c.user_name,
})
```

To change the names of all attributes using a prefix, use the `column_prefix` option. This is useful for classes which wish to add their own `property` accessors:

```
mapper(User, users_table, column_prefix='_')
```

The above will place attribute names such as `_user_id`, `_user_name`, `_password` etc. on the mapped `User` class.

To place multiple columns which are known to be "synonymous" based on foreign key relationship or join condition into the same mapped attribute, put them together using a list, as below where we map to a `Join`:

```
# join users and addresses
usersaddresses = sql.join(users_table, addresses_table, \
    users_table.c.user_id == addresses_table.c.user_id)

mapper(User, usersaddresses, properties={
    'id':[users_table.c.user_id, addresses_table.c.user_id],
})
```

4.1.2 Deferred Column Loading

This feature allows particular columns of a table to not be loaded by default, instead being loaded later on when first referenced. It is essentially "column-level lazy loading". This feature is useful when one wants to avoid loading a large text or binary field into memory when it's not needed. Individual columns can be lazy loaded by themselves or placed into groups that lazy-load together:

```
book_excerpts = Table('books', db,
    Column('book_id', Integer, primary_key=True),
    Column('title', String(200), nullable=False),
    Column('summary', String(2000)),
    Column('excerpt', String),
    Column('photo', Binary)
)

class Book(object):
    pass

# define a mapper that will load each of 'excerpt' and 'photo' in
# separate, individual-row SELECT statements when each attribute
# is first referenced on the individual object instance
mapper(Book, book_excerpts, properties={
   'excerpt': deferred(book_excerpts.c.excerpt),
   'photo': deferred(book_excerpts.c.photo)
})
```

Deferred columns can be placed into groups so that they load together:

```
book_excerpts = Table('books', db,
  Column('book_id', Integer, primary_key=True),
  Column('title', String(200), nullable=False),
  Column('summary', String(2000)),
  Column('excerpt', String),
  Column('photo1', Binary),
  Column('photo2', Binary),
  Column('photo3', Binary)
)

class Book(object):
    pass

# define a mapper with a 'photos' deferred group.  when one photo is referenced,
# all three photos will be loaded in one SELECT statement.  The 'excerpt' will
# be loaded separately when it is first referenced.
mapper(Book, book_excerpts, properties = {
  'excerpt': deferred(book_excerpts.c.excerpt),
  'photo1': deferred(book_excerpts.c.photo1, group='photos'),
  'photo2': deferred(book_excerpts.c.photo2, group='photos'),
  'photo3': deferred(book_excerpts.c.photo3, group='photos')
})
```

You can defer or undefer columns at the `Query` level using the `defer` and `undefer` options:

```
query = session.query(Book)
query.options(defer('summary')).all()
query.options(undefer('excerpt')).all()
```

And an entire "deferred group", i.e. which uses the `group` keyword argument to `deferred()`, can be undeferred using `undefer_group()`, sending in the group name:

```
query = session.query(Book)
query.options(undefer_group('photos')).all()
```

4.1.3 SQL Expressions as Mapped Attributes

To add a SQL clause composed of local or external columns as a read-only, mapped column attribute, use the `column_property()` function. Any scalar-returning `ClauseElement` may be used, as long as it has a `name` attribute; usually, you'll want to call `label()` to give it a specific name:

```
mapper(User, users_table, properties={
    'fullname': column_property(
        (users_table.c.firstname + " " + users_table.c.lastname).label('fullname')
    )
})
```

Correlated subqueries may be used as well:

```
mapper(User, users_table, properties={
    'address_count': column_property(
            select(
                [func.count(addresses_table.c.address_id)],
                addresses_table.c.user_id==users_table.c.user_id
            ).label('address_count')
        )
})
```

4.1.4 Changing Attribute Behavior

Simple Validators

A quick way to add a "validation" routine to an attribute is to use the `validates()` decorator. This is a shortcut for using the `sqlalchemy.orm.util.Validator` attribute extension with individual column or relation based attributes. An attribute validator can raise an exception, halting the process of mutating the attribute's value, or can change the given value into something different. Validators, like all attribute extensions, are only called by normal userland code; they are not issued when the ORM is populating the object.

```
addresses_table = Table('addresses', metadata,
    Column('id', Integer, primary_key=True),
    Column('email', String)
)

class EmailAddress(object):
    @validates('email')
    def validate_email(self, key, address):
        assert '@' in address
        return address

mapper(EmailAddress, addresses_table)
```

Validators also receive collection events, when items are added to a collection:

```
class User(object):
    @validates('addresses')
    def validate_address(self, key, address):
        assert '@' in address.email
        return address
```

Using Descriptors

A more comprehensive way to produce modified behavior for an attribute is to use descriptors. These are commonly used in Python using the `property()` function. The standard SQLAlchemy technique for descriptors is to create a plain descriptor, and to have it read/write from a mapped attribute with a different name. To have the descriptor named the same as a column, map the column under a different name, i.e.:

```
class EmailAddress(object):
   def _set_email(self, email):
      self._email = email
   def _get_email(self):
      return self._email
   email = property(_get_email, _set_email)

mapper(MyAddress, addresses_table, properties={
    '_email': addresses_table.c.email
})
```

However, the approach above is not complete. While our `EmailAddress` object will shuttle the value through the `email` descriptor and into the `_email` mapped attribute, the class level `EmailAddress.email` attribute does not have the usual expression semantics usable with `Query`. To provide these, we instead use the `synonym()` function as follows:

```
mapper(EmailAddress, addresses_table, properties={
    'email': synonym('_email', map_column=True)
})
```

The `email` attribute is now usable in the same way as any other mapped attribute, including filter expressions, get/set operations, etc.:

```
address = session.query(EmailAddress).filter(EmailAddress.email == 'some address').one()

address.email = 'some other address'
```

```
session.flush()

q = session.query(EmailAddress).filter_by(email='some other address')
```

If the mapped class does not provide a property, the `synonym()` construct will create a default getter/setter object automatically.

Custom Comparators

The expressions returned by comparison operations, such as `User.name=='ed'`, can be customized. SQLAlchemy attributes generate these expressions using `PropComparator` objects, which provide common Python expression overrides including `__eq__()`, `__ne__()`, `__lt__()`, and so on. Any mapped attribute can be passed a user-defined class via the `comparator_factory` keyword argument, which subclasses the appropriate `PropComparator` in use, which can provide any or all of these methods:

```
from sqlalchemy.orm.properties import ColumnProperty
class MyComparator(ColumnProperty.Comparator):
    def __eq__(self, other):
        return func.lower(self.__clause_element__()) == func.lower(other)

mapper(EmailAddress, addresses_table, properties={
    'email':column_property(addresses_table.c.email, comparator_factory=MyComparator)
})
```

Above, comparisons on the `email` column are wrapped in the SQL lower() function to produce case-insensitive matching:

```
>>> str(EmailAddress.email == 'SomeAddress@foo.com')
lower(addresses.email) = lower(:lower_1)
```

The `__clause_element__()` method is provided by the base `Comparator` class in use, and represents the SQL element which best matches what this attribute represents. For a column-based attribute, it's the mapped column. For a composite attribute, it's a `ClauseList` consisting of each column represented. For a relation, it's the table mapped by the local mapper (not the remote mapper). `__clause_element__()` should be honored by the custom comparator class in most cases since the resulting element will be applied any translations which are in effect, such as the correctly aliased member when using an `aliased()` construct or certain `with_polymorphic()` scenarios.

There are four kinds of `Comparator` classes which may be subclassed, as according to the type of mapper property configured:

- `column_property()` attribute - `sqlalchemy.orm.properties.ColumnProperty.Comparator`
- `composite()` attribute - `sqlalchemy.orm.properties.CompositeProperty.Comparator`
- `relation()` attribute - `sqlalchemy.orm.properties.RelationProperty.Comparator`
- `comparable_property()` attribute - `sqlalchemy.orm.interfaces.PropComparator`

When using `comparable_property()`, which is a mapper property that isn't tied to any column or mapped table, the `__clause_element__()` method of `PropComparator` should also be implemented.

The `comparator_factory` argument is accepted by all `MapperProperty`-producing functions: `column_property()`, `composite()`, `comparable_property()`, `synonym()`, `relation()`, `backref()`, `deferred()`, and `dynamic_loader()`.

4.1.5 Composite Column Types

Sets of columns can be associated with a single datatype. The ORM treats the group of columns like a single column which accepts and returns objects using the custom datatype you provide. In this example, we'll create a table `vertices` which stores a pair of x/y coordinates, and a custom datatype `Point` which is a composite type of an x and y column:

```
vertices = Table('vertices', metadata,
    Column('id', Integer, primary_key=True),
    Column('x1', Integer),
    Column('y1', Integer),
    Column('x2', Integer),
    Column('y2', Integer),
    )
```

The requirements for the custom datatype class are that it have a constructor which accepts positional arguments corresponding to its column format, and also provides a method `__composite_values__()` which returns the state of the object as a list or tuple, in order of its column-based attributes. It also should supply adequate `__eq__()` and `__ne__()` methods which test the equality of two instances, and may optionally provide a `__set_composite_values__` method which is used to set internal state in some cases (typically when default values have been generated during a flush):

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __composite_values__(self):
        return [self.x, self.y]
    def __set_composite_values__(self, x, y):
        self.x = x
        self.y = y
    def __eq__(self, other):
        return other.x == self.x and other.y == self.y
    def __ne__(self, other):
        return not self.__eq__(other)
```

If `__set_composite_values__()` is not provided, the names of the mapped columns are taken as the names of attributes on the object, and `setattr()` is used to set data.

Setting up the mapping uses the `composite()` function:

```
class Vertex(object):
    pass

mapper(Vertex, vertices, properties={
    'start': composite(Point, vertices.c.x1, vertices.c.y1),
    'end': composite(Point, vertices.c.x2, vertices.c.y2)
})
```

We can now use the `Vertex` instances as well as querying as though the `start` and `end` attributes are regular scalar attributes:

```
session = Session()
v = Vertex(Point(3, 4), Point(5, 6))
```

```
session.save(v)

v2 = session.query(Vertex).filter(Vertex.start == Point(3, 4))
```

The "equals" comparison operation by default produces an AND of all corresponding columns equated to one another. This can be changed using the `comparator_factory`, described in *Custom Comparators*:

```
from sqlalchemy.orm.properties import CompositeProperty
from sqlalchemy import sql

class PointComparator(CompositeProperty.Comparator):
    def __gt__(self, other):
        """define the 'greater than' operation"""

        return sql.and_(*[a>b for a, b in
                          zip(self.__clause_element__().clauses,
                              other.__composite_values__())])

maper(Vertex, vertices, properties={
    'start': composite(Point, vertices.c.x1, vertices.c.y1, comparator_factory=PointCompara
    'end': composite(Point, vertices.c.x2, vertices.c.y2, comparator_factory=PointComparato
})
```

4.1.6 Controlling Ordering

As of version 0.5, the ORM does not generate ordering for any query unless explicitly configured.

The "default" ordering for a collection, which applies to list-based collections, can be configured using the `order_by` keyword argument on `relation()`:

```
mapper(Address, addresses_table)

# order address objects by address id
mapper(User, users_table, properties={
    'addresses': relation(Address, order_by=addresses_table.c.address_id)
})
```

Note that when using eager loaders with relations, the tables used by the eager load's join are anonymously aliased. You can only order by these columns if you specify it at the `relation()` level. To control ordering at the query level based on a related table, you `join()` to that relation, then order by it:

```
session.query(User).join('addresses').order_by(Address.street)
```

Ordering for rows loaded through `Query` is usually specified using the `order_by()` generative method. There is also an option to set a default ordering for Queries which are against a single mapped entity and where there was no explicit `order_by()` stated, which is the `order_by` keyword argument to `mapper()`:

```
# order by a column
mapper(User, users_table, order_by=users_table.c.user_id)

# order by multiple items
mapper(User, users_table, order_by=[users_table.c.user_id, users_table.c.user_name.desc()]
```

Above, a `Query` issued for the `User` class will use the value of the mapper's `order_by` setting if the `Query` itself has no ordering specified.

4.1.7 Mapping Class Inheritance Hierarchies

SQLAlchemy supports three forms of inheritance: *single table inheritance*, where several types of classes are stored in one table, *concrete table inheritance*, where each type of class is stored in its own table, and *joined table inheritance*, where the parent/child classes are stored in their own tables that are joined together in a select. Whereas support for single and joined table inheritance is strong, concrete table inheritance is a less common scenario with some particular problems so is not quite as flexible.

When mappers are configured in an inheritance relationship, SQLAlchemy has the ability to load elements "polymorphically", meaning that a single query can return objects of multiple types.

For the following sections, assume this class relationship:

```
class Employee(object):
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return self.__class__.__name__ + " " + self.name

class Manager(Employee):
    def __init__(self, name, manager_data):
        self.name = name
        self.manager_data = manager_data
    def __repr__(self):
        return self.__class__.__name__ + " " + self.name + " " +  self.manager_data

class Engineer(Employee):
    def __init__(self, name, engineer_info):
        self.name = name
        self.engineer_info = engineer_info
    def __repr__(self):
        return self.__class__.__name__ + " " + self.name + " " +  self.engineer_info
```

Joined Table Inheritance

In joined table inheritance, each class along a particular classes' list of parents is represented by a unique table. The total set of attributes for a particular instance is represented as a join along all tables in its inheritance path. Here, we first define a table to represent the `Employee` class. This table will contain a primary key column (or columns), and a column for each attribute that's represented by `Employee`. In this case it's just `name`:

```
employees = Table('employees', metadata,
   Column('employee_id', Integer, primary_key=True),
   Column('name', String(50)),
   Column('type', String(30), nullable=False)
)
```

The table also has a column called `type`. It is strongly advised in both single- and joined- table inheritance scenarios that the root table contains a column whose sole purpose is that of the **discriminator**; it stores a value which indicates the type of object represented within the row. The column may be of any desired datatype. While there are some "tricks" to work around the requirement that there be a discriminator column, they are more complicated to configure when one wishes to load polymorphically.

Next we define individual tables for each of `Engineer` and `Manager`, which contain columns that represent the attributes unique to the subclass they represent. Each table also must contain a primary key column (or columns), and in most cases a foreign key reference to the parent table. It is standard practice that the same column is used for both of these roles, and that the column is also named the same as that of the parent table. However this is optional in SQLAlchemy; separate columns may be used for primary key and parent-relation, the column may be named differently than that of the parent, and even a custom join condition can be specified between parent and child tables instead of using a foreign key:

```
engineers = Table('engineers', metadata,
   Column('employee_id', Integer, ForeignKey('employees.employee_id'), primary_key=True),
   Column('engineer_info', String(50)),
)

managers = Table('managers', metadata,
   Column('employee_id', Integer, ForeignKey('employees.employee_id'), primary_key=True),
   Column('manager_data', String(50)),
)
```

One natural effect of the joined table inheritance configuration is that the identity of any mapped object can be determined entirely from the base table. This has obvious advantages, so SQLAlchemy always considers the primary key columns of a joined inheritance class to be those of the base table only, unless otherwise manually configured. In other words, the `employee_id` column of both the `engineers` and `managers` table is not used to locate the `Engineer` or `Manager` object itself - only the value in `employees.employee_id` is considered, and the primary key in this case is non-composite. `engineers.employee_id` and `managers.employee_id` are still of course critical to the proper operation of the pattern overall as they are used to locate the joined row, once the parent row has been determined, either through a distinct SELECT statement or all at once within a JOIN.

We then configure mappers as usual, except we use some additional arguments to indicate the inheritance relationship, the polymorphic discriminator column, and the **polymorphic identity** of each class; this is the value that will be stored in the polymorphic discriminator column.

```
mapper(Employee, employees, polymorphic_on=employees.c.type, polymorphic_identity='employee
mapper(Engineer, engineers, inherits=Employee, polymorphic_identity='engineer')
mapper(Manager, managers, inherits=Employee, polymorphic_identity='manager')
```

And that's it. Querying against `Employee` will return a combination of `Employee`, `Engineer` and `Manager` objects. Newly saved `Engineer`, `Manager`, and `Employee` objects will automatically populate the `employees.type` column with `engineer`, `manager`, or `employee`, as appropriate.

Controlling Which Tables are Queried

The `with_polymorphic()` method of `Query` affects the specific subclass tables which the Query selects from. Normally, a query such as this:

```
session.query(Employee).all()
```

...selects only from the `employees` table. When loading fresh from the database, our joined-table setup will query from the parent table only, using SQL such as this:

```
SELECT employees.employee_id AS employees_employee_id, employees.name AS employees_name, em
FROM employees
[]
```

As attributes are requested from those `Employee` objects which are represented in either the `engineers` or `managers` child tables, a second load is issued for the columns in that related row, if the data was not already loaded. So above, after accessing the objects you'd see further SQL issued along the lines of:

```
SELECT managers.employee_id AS managers_employee_id, managers.manager_data AS managers_mana
FROM managers
WHERE ? = managers.employee_id
[5]
SELECT engineers.employee_id AS engineers_employee_id, engineers.engineer_info AS engineers
FROM engineers
WHERE ? = engineers.employee_id
[2]
```

This behavior works well when issuing searches for small numbers of items, such as when using `get()`, since the full range of joined tables are not pulled in to the SQL statement unnecessarily. But when querying a larger span of rows which are known to be of many types, you may want to actively join to some or all of the joined tables. The `with_polymorphic` feature of `Query` and `mapper` provides this.

Telling our query to polymorphically load `Engineer` and `Manager` objects:

```
query = session.query(Employee).with_polymorphic([Engineer, Manager])
```

produces a query which joins the `employees` table to both the `engineers` and `managers` tables like the following:

```
query.all()

SELECT employees.employee_id AS employees_employee_id, engineers.employee_id AS engineers_e
FROM employees LEFT OUTER JOIN engineers ON employees.employee_id = engineers.employee_id L
[]
```

`with_polymorphic()` accepts a single class or mapper, a list of classes/mappers, or the string `'*'` to indicate all subclasses:

```
# join to the engineers table
query.with_polymorphic(Engineer)

# join to the engineers and managers tables
query.with_polymorphic([Engineer, Manager])

# join to all subclass tables
query.with_polymorphic('*')
```

It also accepts a second argument `selectable` which replaces the automatic join creation and instead selects directly from the selectable given. This feature is normally used with "concrete" inheritance, described later, but can be used with any kind of inheritance setup in the case that specialized SQL should be used to load polymorphically:

```
# custom selectable
query.with_polymorphic([Engineer, Manager], employees.outerjoin(managers).outerjoin(enginee
```

`with_polymorphic()` is also needed when you wish to add filter criterion that is specific to one or more subclasses, so that those columns are available to the WHERE clause:

```
session.query(Employee).with_polymorphic([Engineer, Manager]).\
    filter(or_(Engineer.engineer_info=='w', Manager.manager_data=='q'))
```

Note that if you only need to load a single subtype, such as just the `Engineer` objects, `with_polymorphic()` is not needed since you would query against the `Engineer` class directly.

The mapper also accepts `with_polymorphic` as a configurational argument so that the joined-style load will be issued automatically. This argument may be the string `'*'`, a list of classes, or a tuple consisting of either, followed by a selectable.

```
mapper(Employee, employees, polymorphic_on=employees.c.type, \
    polymorphic_identity='employee', with_polymorphic='*')
mapper(Engineer, engineers, inherits=Employee, polymorphic_identity='engineer')
mapper(Manager, managers, inherits=Employee, polymorphic_identity='manager')
```

The above mapping will produce a query similar to that of `with_polymorphic('*')` for every query of `Employee` objects.

Using `with_polymorphic()` with `Query` will override the mapper-level `with_polymorphic` setting.

Creating Joins to Specific Subtypes

The `of_type()` method is a helper which allows the construction of joins along `relation` paths while narrowing the criterion to specific subclasses. Suppose the `employees` table represents a collection of employees which are associated with a `Company` object. We'll add a `company_id` column to the `employees` table and a new table `companies`:

```
companies = Table('companies', metadata,
   Column('company_id', Integer, primary_key=True),
   Column('name', String(50))
   )

employees = Table('employees', metadata,
  Column('employee_id', Integer, primary_key=True),
  Column('name', String(50)),
  Column('type', String(30), nullable=False),
  Column('company_id', Integer, ForeignKey('companies.company_id'))
)

class Company(object):
    pass

mapper(Company, companies, properties={
    'employees': relation(Employee)
})
```

When querying from `Company` onto the `Employee` relation, the `join()` method as well as the `any()` and `has()` operators will create a join from `companies` to `employees`, without including `engineers` or `managers` in the mix. If we wish to have criterion which is specifically against the `Engineer` class, we can tell those methods to join or subquery against the joined table representing the subclass using the `of_type()` operator:

```
session.query(Company).join(Company.employees.of_type(Engineer)).filter(Engineer.engineer_i
```

A longhand version of this would involve spelling out the full target selectable within a 2-tuple:

```
session.query(Company).join((employees.join(engineers), Company.employees)).filter(Enginee
```

Currently, `of_type()` accepts a single class argument. It may be expanded later on to accept multiple classes. For now, to join to any group of subclasses, the longhand notation allows this flexibility:

```
session.query(Company).join((employees.outerjoin(engineers).outerjoin(managers), Company.em
    filter(or_(Engineer.engineer_info=='someinfo', Manager.manager_data=='somedata'))
```

The `any()` and `has()` operators also can be used with `of_type()` when the embedded criterion is in terms of a subclass:

```
session.query(Company).filter(Company.employees.of_type(Engineer).any(Engineer.engineer_in
```

Note that the `any()` and `has()` are both shorthand for a correlated EXISTS query. To build one by hand looks like:

```
session.query(Company).filter(
    exists([1],
        and_(Engineer.engineer_info=='someinfo', employees.c.company_id==companies.c.compa
        from_obj=employees.join(engineers)
    )
).all()
```

The EXISTS subquery above selects from the join of `employees` to `engineers`, and also specifies criterion which correlates the EXISTS subselect back to the parent `companies` table.

Single Table Inheritance

Single table inheritance is where the attributes of the base class as well as all subclasses are represented within a single table. A column is present in the table for every attribute mapped to the base class and all subclasses; the columns which correspond to a single subclass are nullable. This configuration looks much like joined-table inheritance except there's only one table. In this case, a `type` column is required, as there would be no other way to discriminate between classes. The table is specified in the base mapper only; for the inheriting classes, leave their `table` parameter blank:

```
employees_table = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('manager_data', String(50)),
    Column('engineer_info', String(50)),
    Column('type', String(20), nullable=False)
)

employee_mapper = mapper(Employee, employees_table, \
    polymorphic_on=employees_table.c.type, polymorphic_identity='employee')
manager_mapper = mapper(Manager, inherits=employee_mapper, polymorphic_identity='manager')
engineer_mapper = mapper(Engineer, inherits=employee_mapper, polymorphic_identity='enginee
```

Note that the mappers for the derived classes Manager and Engineer omit the specification of their associated table, as it is inherited from the employee_mapper. Omitting the table specification for derived mappers in single-table inheritance is required.

Concrete Table Inheritance

This form of inheritance maps each class to a distinct table, as below:

```
employees_table = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
)

managers_table = Table('managers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('manager_data', String(50)),
)

engineers_table = Table('engineers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('engineer_info', String(50)),
)
```

Notice in this case there is no `type` column. If polymorphic loading is not required, there's no advantage to using `inherits` here; you just define a separate mapper for each class.

```
mapper(Employee, employees_table)
mapper(Manager, managers_table)
mapper(Engineer, engineers_table)
```

To load polymorphically, the `with_polymorphic` argument is required, along with a selectable indicating how rows should be loaded. In this case we must construct a UNION of all three tables. SQLAlchemy includes a helper function to create these called `polymorphic_union`, which will map all the different columns into a structure of selects with the same numbers and names of columns, and also generate a virtual `type` column for each subselect:

```
pjoin = polymorphic_union({
    'employee': employees_table,
    'manager': managers_table,
    'engineer': engineers_table
}, 'type', 'pjoin')

employee_mapper = mapper(Employee, employees_table, with_polymorphic=('*', pjoin), \
    polymorphic_on=pjoin.c.type, polymorphic_identity='employee')
manager_mapper = mapper(Manager, managers_table, inherits=employee_mapper, \
    concrete=True, polymorphic_identity='manager')
engineer_mapper = mapper(Engineer, engineers_table, inherits=employee_mapper, \
    concrete=True, polymorphic_identity='engineer')
```

Upon select, the polymorphic union produces a query like this:

```
session.query(Employee).all()

SELECT pjoin.type AS pjoin_type, pjoin.manager_data AS pjoin_manager_data, pjoin.employee_
pjoin.name AS pjoin_name, pjoin.engineer_info AS pjoin_engineer_info
```

```
FROM (
    SELECT employees.employee_id AS employee_id, CAST(NULL AS VARCHAR(50)) AS manager_data,
    CAST(NULL AS VARCHAR(50)) AS engineer_info, 'employee' AS type
    FROM employees
UNION ALL
    SELECT managers.employee_id AS employee_id, managers.manager_data AS manager_data, mana
    CAST(NULL AS VARCHAR(50)) AS engineer_info, 'manager' AS type
    FROM managers
UNION ALL
    SELECT engineers.employee_id AS employee_id, CAST(NULL AS VARCHAR(50)) AS manager_data,
    engineers.engineer_info AS engineer_info, 'engineer' AS type
    FROM engineers
) AS pjoin
[]
```

Using Relations with Inheritance

Both joined-table and single table inheritance scenarios produce mappings which are usable in `relation()` functions; that is, it's possible to map a parent object to a child object which is polymorphic. Similarly, inheriting mappers can have `relation()` objects of their own at any level, which are inherited to each child class. The only requirement for relations is that there is a table relationship between parent and child. An example is the following modification to the joined table inheritance example, which sets a bi-directional relationship between `Employee` and `Company`:

```
employees_table = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('company_id', Integer, ForeignKey('companies.company_id'))
)

companies = Table('companies', metadata,
   Column('company_id', Integer, primary_key=True),
   Column('name', String(50)))

class Company(object):
    pass

mapper(Company, companies, properties={
   'employees': relation(Employee, backref='company')
})
```

SQLAlchemy has a lot of experience in this area; the optimized "outer join" approach can be used freely for parent and child relationships, eager loads are fully useable, `aliased()` objects and other techniques are fully supported as well.

In a concrete inheritance scenario, mapping relations is more difficult since the distinct classes do not share a table. In this case, you *can* establish a relationship from parent to child if a join condition can be constructed from parent to child, if each child table contains a foreign key to the parent:

```
companies = Table('companies', metadata,
   Column('id', Integer, primary_key=True),
   Column('name', String(50)))

employees_table = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('company_id', Integer, ForeignKey('companies.id'))
)

managers_table = Table('managers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('manager_data', String(50)),
    Column('company_id', Integer, ForeignKey('companies.id'))
)

engineers_table = Table('engineers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('engineer_info', String(50)),
    Column('company_id', Integer, ForeignKey('companies.id'))
)

mapper(Employee, employees_table, with_polymorphic=('*', pjoin), polymorphic_on=pjoin.c.typ
mapper(Manager, managers_table, inherits=employee_mapper, concrete=True, polymorphic_identi
mapper(Engineer, engineers_table, inherits=employee_mapper, concrete=True, polymorphic_ider
mapper(Company, companies, properties={
    'employees': relation(Employee)
})
```

The big limitation with concrete table inheritance is that `relation()` objects placed on each concrete mapper do **not** propagate to child mappers. If you want to have the same `relation()` objects set up on all concrete mappers, they must be configured manually on each. To configure back references in such a configuration the `back_populates` keyword may be used instead of `backref`, such as below where both `A(object)` and `B(A)` bidirectionally reference `C`:

```
ajoin = polymorphic_union({
        'a':a_table,
        'b':b_table
    }, 'type', 'ajoin')

mapper(A, a_table, with_polymorphic=('*', ajoin),
    polymorphic_on=ajoin.c.type, polymorphic_identity='a',
    properties={
        'some_c':relation(C, back_populates='many_a')
})
mapper(B, b_table,inherits=A, concrete=True,
    polymorphic_identity='b',
    properties={
        'some_c':relation(C, back_populates='many_a')
})
mapper(C, c_table, properties={
    'many_a':relation(A, collection_class=set, back_populates='some_c'),
})
```

4.1.8 Mapping a Class against Multiple Tables

Mappers can be constructed against arbitrary relational units (called `Selectables`) as well as plain `Tables`. For example, The `join` keyword from the SQL package creates a neat selectable unit comprised of multiple tables, complete with its own composite primary key, which can be passed in to a mapper as the table.

```
# a class
class AddressUser(object):
    pass

# define a Join
j = join(users_table, addresses_table)

# map to it - the identity of an AddressUser object will be
# based on (user_id, address_id) since those are the primary keys involved
mapper(AddressUser, j, properties={
    'user_id': [users_table.c.user_id, addresses_table.c.user_id]
})
```

A second example:

```
# many-to-many join on an association table
j = join(users_table, userkeywords,
        users_table.c.user_id==userkeywords.c.user_id).join(keywords,
           userkeywords.c.keyword_id==keywords.c.keyword_id)

# a class
class KeywordUser(object):
    pass

# map to it - the identity of a KeywordUser object will be
# (user_id, keyword_id) since those are the primary keys involved
mapper(KeywordUser, j, properties={
    'user_id': [users_table.c.user_id, userkeywords.c.user_id],
    'keyword_id': [userkeywords.c.keyword_id, keywords.c.keyword_id]
})
```

In both examples above, “composite” columns were added as properties to the mappers; these are aggregations of multiple columns into one mapper property, which instructs the mapper to keep both of those columns set at the same value.

4.1.9 Mapping a Class against Arbitrary Selects

Similar to mapping against a join, a plain select() object can be used with a mapper as well. Below, an example select which contains two aggregate functions and a group_by is mapped to a class:

```
s = select([customers,
            func.count(orders).label('order_count'),
            func.max(orders.price).label('highest_order')],
            customers.c.customer_id==orders.c.customer_id,
            group_by=[c for c in customers.c]
            ).alias('somealias')
```

```
class Customer(object):
    pass

mapper(Customer, s)
```

Above, the "customers" table is joined against the "orders" table to produce a full row for each customer row, the total count of related rows in the "orders" table, and the highest price in the "orders" table, grouped against the full set of columns in the "customers" table. That query is then mapped against the Customer class. New instances of Customer will contain attributes for each column in the "customers" table as well as an "order_count" and "highest_order" attribute. Updates to the Customer object will only be reflected in the "customers" table and not the "orders" table. This is because the primary key columns of the "orders" table are not represented in this mapper and therefore the table is not affected by save or delete operations.

4.1.10 Multiple Mappers for One Class

The first mapper created for a certain class is known as that class's "primary mapper." Other mappers can be created as well on the "load side" - these are called **secondary mappers**. This is a mapper that must be constructed with the keyword argument `non_primary=True`, and represents a load-only mapper. Objects that are loaded with a secondary mapper will have their save operation processed by the primary mapper. It is also invalid to add new `relation()` objects to a non-primary mapper. To use this mapper with the Session, specify it to the `query` method:

example:

```
# primary mapper
mapper(User, users_table)

# make a secondary mapper to load User against a join
othermapper = mapper(User, users_table.join(someothertable), non_primary=True)

# select
result = session.query(othermapper).select()
```

The "non primary mapper" is a rarely needed feature of SQLAlchemy; in most cases, the `Query` object can produce any kind of query that's desired. It's recommended that a straight `Query` be used in place of a non-primary mapper unless the mapper approach is absolutely needed. Current use cases for the "non primary mapper" are when you want to map the class to a particular select statement or view to which additional query criterion can be added, and for when the particular mapped select statement or view is to be placed in a `relation()` of a parent mapper.

Versions of SQLAlchemy previous to 0.5 included another mapper flag called "entity_name", as of version 0.5.0 this feature has been removed (it never worked very well).

4.1.11 Constructors and Object Initialization

Mapping imposes no restrictions or requirements on the constructor (`__init__`) method for the class. You are free to require any arguments for the function that you wish, assign attributes to the instance that are unknown to the ORM, and generally do anything else you would normally do when writing a constructor for a Python class.

The SQLAlchemy ORM does not call `__init__` when recreating objects from database rows. The ORM's process is somewhat akin to the Python standard library's `pickle` module, invoking the low level `__new__` method and then quietly restoring attributes directly on the instance rather than calling `__init__`.

If you need to do some setup on database-loaded instances before they're ready to use, you can use the `@reconstructor` decorator to tag a method as the ORM counterpart to `__init__`. SQLAlchemy will call this method with no arguments every time it loads or reconstructs one of your instances. This is useful for recreating transient properties that are normally assigned in your `__init__`:

```
from sqlalchemy import orm

class MyMappedClass(object):
    def __init__(self, data):
        self.data = data
        # we need stuff on all instances, but not in the database.
        self.stuff = []

    @orm.reconstructor
    def init_on_load(self):
        self.stuff = []
```

When `obj = MyMappedClass()` is executed, Python calls the `__init__` method as normal and the `data` argument is required. When instances are loaded during a `Query` operation as in `query(MyMappedClass).one()`, `init_on_load` is called instead.

Any method may be tagged as the `reconstructor`, even the `__init__` method. SQLAlchemy will call the reconstructor method with no arguments. Scalar (non-collection) database-mapped attributes of the instance will be available for use within the function. Eagerly-loaded collections are generally not yet available and will usually only contain the first element. ORM state changes made to objects at this stage will not be recorded for the next flush() operation, so the activity within a reconstructor should be conservative.

While the ORM does not call your `__init__` method, it will modify the class's `__init__` slightly. The method is lightly wrapped to act as a trigger for the ORM, allowing mappers to be compiled automatically and will fire a `init_instance` event that `MapperExtension` objectss may listen for. `MapperExtension` objects can also listen for a `reconstruct_instance` event, analogous to the `reconstructor` decorator above.

4.1.12 Extending Mapper

Mappers can have functionality augmented or replaced at many points in its execution via the usage of the MapperExtension class. This class is just a series of "hooks" where various functionality takes place. An application can make its own MapperExtension objects, overriding only the methods it needs. Methods that are not overridden return the special value `sqlalchemy.orm.EXT_CONTINUE` to allow processing to continue to the next MapperExtension or simply proceed normally if there are no more extensions.

API documentation for MapperExtension: `sqlalchemy.orm.interfaces.MapperExtension`

To use MapperExtension, make your own subclass of it and just send it off to a mapper:

```
m = mapper(User, users_table, extension=MyExtension())
```

Multiple extensions will be chained together and processed in order; they are specified as a list:

```
m = mapper(User, users_table, extension=[ext1, ext2, ext3])
```

4.2 Relation Configuration

4.2.1 Basic Relational Patterns

A quick walkthrough of the basic relational patterns.

One To Many

A one to many relationship places a foreign key in the child table referencing the parent. SQLAlchemy creates the relationship as a collection on the parent object containing instances of the child object.

```
parent_table = Table('parent', metadata,
    Column('id', Integer, primary_key=True))

child_table = Table('child', metadata,
    Column('id', Integer, primary_key=True),
    Column('parent_id', Integer, ForeignKey('parent.id')))

class Parent(object):
    pass

class Child(object):
    pass

mapper(Parent, parent_table, properties={
    'children': relation(Child)
})

mapper(Child, child_table)
```

To establish a bi-directional relationship in one-to-many, where the "reverse" side is a many to one, specify the `backref` option:

```
mapper(Parent, parent_table, properties={
    'children': relation(Child, backref='parent')
})

mapper(Child, child_table)
```

`Child` will get a `parent` attribute with many-to-one semantics.

Many To One

Many to one places a foreign key in the parent table referencing the child. The mapping setup is identical to one-to-many, however SQLAlchemy creates the relationship as a scalar attribute on the parent object referencing a single instance of the child object.

```
parent_table = Table('parent', metadata,
    Column('id', Integer, primary_key=True),
    Column('child_id', Integer, ForeignKey('child.id')))

child_table = Table('child', metadata,
    Column('id', Integer, primary_key=True),
    )

class Parent(object):
    pass
```

```
class Child(object):
    pass

mapper(Parent, parent_table, properties={
    'child': relation(Child)
})

mapper(Child, child_table)
```

Backref behavior is available here as well, where `backref="parents"` will place a one-to-many collection on the `Child` class.

One To One

One To One is essentially a bi-directional relationship with a scalar attribute on both sides. To achieve this, the `uselist=False` flag indicates the placement of a scalar attribute instead of a collection on the "many" side of the relationship. To convert one-to-many into one-to-one:

```
mapper(Parent, parent_table, properties={
    'child': relation(Child, uselist=False, backref='parent')
})
```

Or to turn many-to-one into one-to-one:

```
mapper(Parent, parent_table, properties={
    'child': relation(Child, backref=backref('parent', uselist=False))
})
```

Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the `secondary` argument to `relation()`.

```
left_table = Table('left', metadata,
    Column('id', Integer, primary_key=True))

right_table = Table('right', metadata,
    Column('id', Integer, primary_key=True))

association_table = Table('association', metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id')),
    )

mapper(Parent, left_table, properties={
    'children': relation(Child, secondary=association_table)
})

mapper(Child, right_table)
```

For a bi-directional relationship, both sides of the relation contain a collection by default, which can be modified on either side via the `uselist` flag to be scalar. The `backref` keyword will automatically use the same `secondary` argument for the reverse relation:

```
mapper(Parent, left_table, properties={
    'children': relation(Child, secondary=association_table, backref='parents')
})
```

Association Object

The association object pattern is a variant on many-to-many: it specifically is used when your association table contains additional columns beyond those which are foreign keys to the left and right tables. Instead of using the `secondary` argument, you map a new class directly to the association table. The left side of the relation references the association object via one-to-many, and the association class references the right side via many-to-one.

```
left_table = Table('left', metadata,
    Column('id', Integer, primary_key=True))

right_table = Table('right', metadata,
    Column('id', Integer, primary_key=True))

association_table = Table('association', metadata,
    Column('left_id', Integer, ForeignKey('left.id'), primary_key=True),
    Column('right_id', Integer, ForeignKey('right.id'), primary_key=True),
    Column('data', String(50))
    )

mapper(Parent, left_table, properties={
    'children':relation(Association)
})

mapper(Association, association_table, properties={
    'child':relation(Child)
})

mapper(Child, right_table)
```

The bi-directional version adds backrefs to both relations:

```
mapper(Parent, left_table, properties={
    'children':relation(Association, backref="parent")
})

mapper(Association, association_table, properties={
    'child':relation(Child, backref="parent_assocs")
})

mapper(Child, right_table)
```

Working with the association pattern in its direct form requires that child objects are associated with an association instance before being appended to the parent; similarly, access from parent to child goes through the association object:

```
# create parent, append a child via association
p = Parent()
a = Association()
a.child = Child()
```

```
p.children.append(a)

# iterate through child objects via association, including association
# attributes
for assoc in p.children:
    print assoc.data
    print assoc.child
```

To enhance the association object pattern such that direct access to the `Association` object is optional, SQLAlchemy provides the *associationproxy*.

Important Note: it is strongly advised that the `secondary` table argument not be combined with the Association Object pattern, unless the `relation()` which contains the `secondary` argument is marked `viewonly=True`. Otherwise, SQLAlchemy may persist conflicting data to the underlying association table since it is represented by two conflicting mappings. The Association Proxy pattern should be favored in the case where access to the underlying association data is only sometimes needed.

4.2.2 Adjacency List Relationships

The **adjacency list** pattern is a common relational pattern whereby a table contains a foreign key reference to itself. This is the most common and simple way to represent hierarchical data in flat tables. The other way is the "nested sets" model, sometimes called "modified preorder". Despite what many online articles say about modified preorder, the adjacency list model is probably the most appropriate pattern for the large majority of hierarchical storage needs, for reasons of concurrency, reduced complexity, and that modified preorder has little advantage over an application which can fully load subtrees into the application space.

SQLAlchemy commonly refers to an adjacency list relation as a **self-referential mapper**. In this example, we'll work with a single table called `treenodes` to represent a tree structure:

```
nodes = Table('treenodes', metadata,
    Column('id', Integer, primary_key=True),
    Column('parent_id', Integer, ForeignKey('treenodes.id')),
    Column('data', String(50)),
    )
```

A graph such as the following:

```
root --+---> child1
       +---> child2 --+--> subchild1
       |              +--> subchild2
       +---> child3
```

Would be represented with data such as:

```
id       parent_id     data
---      -------       ----
1        NULL          root
2        1             child1
3        1             child2
4        3             subchild1
5        3             subchild2
6        1             child3
```

SQLAlchemy's `mapper()` configuration for a self-referential one-to-many relationship is exactly like a "normal" one-to-many relationship. When SQLAlchemy encounters the foreign key relation from `treenodes` to `treenodes`, it assumes one-to-many unless told otherwise:

```
# entity class
class Node(object):
    pass

mapper(Node, nodes, properties={
    'children': relation(Node)
})
```

To create a many-to-one relationship from child to parent, an extra indicator of the "remote side" is added, which contains the `Column` object or objects indicating the remote side of the relation:

```
mapper(Node, nodes, properties={
    'parent': relation(Node, remote_side=[nodes.c.id])
})
```

And the bi-directional version combines both:

```
mapper(Node, nodes, properties={
    'children': relation(Node, backref=backref('parent', remote_side=[nodes.c.id]))
})
```

There are several examples included with SQLAlchemy illustrating self-referential strategies; these include basic_tree.py and optimized_al.py, the latter of which illustrates how to persist and search XML documents in conjunction with ElementTree.

Self-Referential Query Strategies

Querying self-referential structures is done in the same way as any other query in SQLAlchemy, such as below, we query for any node whose `data` attribute stores the value `child2`:

```
# get all nodes named 'child2'
session.query(Node).filter(Node.data=='child2')
```

On the subject of joins, i.e. those described in *datamapping_joins*, self-referential structures require the usage of aliases so that the same table can be referenced multiple times within the FROM clause of the query. Aliasing can be done either manually using the `nodes` `Table` object as a source of aliases:

```
# get all nodes named 'subchild1' with a parent named 'child2'
nodealias = nodes.alias()
session.query(Node).filter(Node.data=='subchild1').\
    filter(and_(Node.parent_id==nodealias.c.id, nodealias.c.data=='child2')).all()
SELECT treenodes.id AS treenodes_id, treenodes.parent_id AS treenodes_parent_id, treenodes
FROM treenodes, treenodes AS treenodes_1
WHERE treenodes.data = ? AND treenodes.parent_id = treenodes_1.id AND treenodes_1.data = ?
['subchild1', 'child2']
```

or automatically, using `join()` with `aliased=True`:

```
# get all nodes named 'subchild1' with a parent named 'child2'
session.query(Node).filter(Node.data=='subchild1').\
    join('parent', aliased=True).filter(Node.data=='child2').all()
SELECT treenodes.id AS treenodes_id, treenodes.parent_id AS treenodes_parent_id, treenodes
FROM treenodes JOIN treenodes AS treenodes_1 ON treenodes_1.id = treenodes.parent_id
WHERE treenodes.data = ? AND treenodes_1.data = ?
['subchild1', 'child2']
```

To add criterion to multiple points along a longer join, use `from_joinpoint=True`:

```
# get all nodes named 'subchild1' with a parent named 'child2' and a grandparent 'root'
session.query(Node).filter(Node.data=='subchild1').\
    join('parent', aliased=True).filter(Node.data=='child2').\
    join('parent', aliased=True, from_joinpoint=True).filter(Node.data=='root').all()
SELECT treenodes.id AS treenodes_id, treenodes.parent_id AS treenodes_parent_id, treenodes
FROM treenodes JOIN treenodes AS treenodes_1 ON treenodes_1.id = treenodes.parent_id JOIN t
WHERE treenodes.data = ? AND treenodes_1.data = ? AND treenodes_2.data = ?
['subchild1', 'child2', 'root']
```

Configuring Eager Loading

Eager loading of relations occurs using joins or outerjoins from parent to child table during a normal query operation, such that the parent and its child collection can be populated from a single SQL statement. SQLAlchemy's eager loading uses aliased tables in all cases when joining to related items, so it is compatible with self-referential joining. However, to use eager loading with a self-referential relation, SQLAlchemy needs to be told how many levels deep it should join; otherwise the eager load will not take place. This depth setting is configured via `join_depth`:

```
mapper(Node, nodes, properties={
    'children': relation(Node, lazy=False, join_depth=2)
})

session.query(Node).all()
SELECT treenodes_1.id AS treenodes_1_id, treenodes_1.parent_id AS treenodes_1_parent_id, t
FROM treenodes LEFT OUTER JOIN treenodes AS treenodes_2 ON treenodes.id = treenodes_2.pare
[]
```

4.2.3 Specifying Alternate Join Conditions to relation()

The `relation()` function uses the foreign key relationship between the parent and child tables to formulate the **primary join condition** between parent and child; in the case of a many-to-many relationship it also formulates the **secondary join condition**. If you are working with a `Table` which has no `ForeignKey` objects on it (which can be the case when using reflected tables with MySQL), or if the join condition cannot be expressed by a simple foreign key relationship, use the `primaryjoin` and possibly `secondaryjoin` conditions to create the appropriate relationship.

In this example we create a relation `boston_addresses` which will only load the user addresses with a city of "Boston":

```
class User(object):
    pass
class Address(object):
    pass
```

```
mapper(Address, addresses_table)
mapper(User, users_table, properties={
    'boston_addresses': relation(Address, primaryjoin=
                and_(users_table.c.user_id==addresses_table.c.user_id,
                addresses_table.c.city=='Boston'))
})
```

Many to many relationships can be customized by one or both of `primaryjoin` and `secondaryjoin`, shown below with just the default many-to-many relationship explicitly set:

```
class User(object):
    pass
class Keyword(object):
    pass
mapper(Keyword, keywords_table)
mapper(User, users_table, properties={
    'keywords': relation(Keyword, secondary=userkeywords_table,
        primaryjoin=users_table.c.user_id==userkeywords_table.c.user_id,
        secondaryjoin=userkeywords_table.c.keyword_id==keywords_table.c.keyword_id
        )
})
```

Specifying Foreign Keys

When using `primaryjoin` and `secondaryjoin`, SQLAlchemy also needs to be aware of which columns in the relation reference the other. In most cases, a `Table` construct will have `ForeignKey` constructs which take care of this; however, in the case of reflected tables on a database that does not report FKs (like MySQL ISAM) or when using join conditions on columns that don't have foreign keys, the `relation()` needs to be told specifically which columns are "foreign" using the `foreign_keys` collection:

```
mapper(Address, addresses_table)
mapper(User, users_table, properties={
    'addresses': relation(Address, primaryjoin=
                users_table.c.user_id==addresses_table.c.user_id,
                foreign_keys=[addresses_table.c.user_id])
})
```

Building Query-Enabled Properties

Very ambitious custom join conditions may fail to be directly persistable, and in some cases may not even load correctly. To remove the persistence part of the equation, use the flag `viewonly=True` on the `relation()`, which establishes it as a read-only attribute (data written to the collection will be ignored on flush()). However, in extreme cases, consider using a regular Python property in conjunction with `Query` as follows:

```
class User(object):
    def _get_addresses(self):
        return object_session(self).query(Address).with_parent(self).filter(...).all()
    addresses = property(_get_addresses)
```

Multiple Relations against the Same Parent/Child

Theres no restriction on how many times you can relate from parent to child. SQLAlchemy can usually figure out what you want, particularly if the join conditions are straightforward. Below we add a `newyork_addresses` attribute to complement the `boston_addresses` attribute:

```
mapper(User, users_table, properties={
    'boston_addresses': relation(Address, primaryjoin=
                and_(users_table.c.user_id==addresses_table.c.user_id,
                addresses_table.c.city=='Boston')),
    'newyork_addresses': relation(Address, primaryjoin=
                and_(users_table.c.user_id==addresses_table.c.user_id,
                addresses_table.c.city=='New York')),
})
```

4.2.4 Alternate Collection Implementations

Mapping a one-to-many or many-to-many relationship results in a collection of values accessible through an attribute on the parent instance. By default, this collection is a `list`:

```
mapper(Parent, properties={
    children = relation(Child)
})

parent = Parent()
parent.children.append(Child())
print parent.children[0]
```

Collections are not limited to lists. Sets, mutable sequences and almost any other Python object that can act as a container can be used in place of the default list.

```
# use a set
mapper(Parent, properties={
    children = relation(Child, collection_class=set)
})

parent = Parent()
child = Child()
parent.children.add(child)
assert child in parent.children
```

Custom Collection Implementations

You can use your own types for collections as well. For most cases, simply inherit from `list` or `set` and add the custom behavior.

Collections in SQLAlchemy are transparently *instrumented*. Instrumentation means that normal operations on the collection are tracked and result in changes being written to the database at flush time. Additionally, collection operations can fire *events* which indicate some secondary operation must take place. Examples of a secondary operation include saving the child item in the parent's `Session` (i.e. the `save-update` cascade), as well as synchronizing the state of a bi-directional relationship (i.e. a `backref`).

The collections package understands the basic interface of lists, sets and dicts and will automatically apply instrumentation to those built-in types and their subclasses. Object-derived types that implement a basic collection interface are detected and instrumented via duck-typing:

```
class ListLike(object):
    def __init__(self):
        self.data = []
    def append(self, item):
        self.data.append(item)
    def remove(self, item):
        self.data.remove(item)
    def extend(self, items):
        self.data.extend(items)
    def __iter__(self):
        return iter(self.data)
    def foo(self):
        return 'foo'
```

`append`, `remove`, and `extend` are known list-like methods, and will be instrumented automatically. `__iter__` is not a mutator method and won't be instrumented, and `foo` won't be either.

Duck-typing (i.e. guesswork) isn't rock-solid, of course, so you can be explicit about the interface you are implementing by providing an `__emulates__` class attribute:

```
class SetLike(object):
    __emulates__ = set

    def __init__(self):
        self.data = set()
    def append(self, item):
        self.data.add(item)
    def remove(self, item):
        self.data.remove(item)
    def __iter__(self):
        return iter(self.data)
```

This class looks list-like because of `append`, but `__emulates__` forces it to set-like. `remove` is known to be part of the set interface and will be instrumented.

But this class won't work quite yet: a little glue is needed to adapt it for use by SQLAlchemy. The ORM needs to know which methods to use to append, remove and iterate over members of the collection. When using a type like `list` or `set`, the appropriate methods are well-known and used automatically when present. This set-like class does not provide the expected `add` method, so we must supply an explicit mapping for the ORM via a decorator.

Annotating Custom Collections via Decorators

Decorators can be used to tag the individual methods the ORM needs to manage collections. Use them when your class doesn't quite meet the regular interface for its container type, or you simply would like to use a different method to get the job done.

```
from sqlalchemy.orm.collections import collection

class SetLike(object):
    __emulates__ = set

    def __init__(self):
        self.data = set()

    @collection.appender
    def append(self, item):
        self.data.add(item)

    def remove(self, item):
        self.data.remove(item)

    def __iter__(self):
        return iter(self.data)
```

And that's all that's needed to complete the example. SQLAlchemy will add instances via the `append` method. `remove` and `__iter__` are the default methods for sets and will be used for removing and iteration. Default methods can be changed as well:

```
from sqlalchemy.orm.collections import collection

class MyList(list):
    @collection.remover
    def zark(self, item):
        # do something special...

    @collection.iterator
    def hey_use_this_instead_for_iteration(self):
        # ...
```

There is no requirement to be list-, or set-like at all. Collection classes can be any shape, so long as they have the append, remove and iterate interface marked for SQLAlchemy's use. Append and remove methods will be called with a mapped entity as the single argument, and iterator methods are called with no arguments and must return an iterator.

Dictionary-Based Collections

A `dict` can be used as a collection, but a keying strategy is needed to map entities loaded by the ORM to key, value pairs. The *sqlalchemy.orm.collections* package provides several built-in types for dictionary-based collections:

```
from sqlalchemy.orm.collections import column_mapped_collection, attribute_mapped_collectio

mapper(Item, items_table, properties={
    # key by column
    'notes': relation(Note, collection_class=column_mapped_collection(notes_table.c.keyword
    # or named attribute
    'notes2': relation(Note, collection_class=attribute_mapped_collection('keyword')),
    # or any callable
    'notes3': relation(Note, collection_class=mapped_collection(lambda entity: entity.a + e
})

# ...
item = Item()
```

```
item.notes['color'] = Note('color', 'blue')
print item.notes['color']
```

These functions each provide a `dict` subclass with decorated `set` and `remove` methods and the keying strategy of your choice.

The *sqlalchemy.orm.collections.MappedCollection* class can be used as a base class for your custom types or as a mix-in to quickly add `dict` collection support to other classes. It uses a keying function to delegate to `__setitem__` and `__delitem__`:

```
from sqlalchemy.util import OrderedDict
from sqlalchemy.orm.collections import MappedCollection

class NodeMap(OrderedDict, MappedCollection):
    """Holds 'Node' objects, keyed by the 'name' attribute with insert order maintained.""

    def __init__(self, *args, **kw):
        MappedCollection.__init__(self, keyfunc=lambda node: node.name)
        OrderedDict.__init__(self, *args, **kw)
```

The ORM understands the `dict` interface just like lists and sets, and will automatically instrument all dict-like methods if you choose to subclass `dict` or provide dict-like collection behavior in a duck-typed class. You must decorate appender and remover methods, however- there are no compatible methods in the basic dictionary interface for SQLAlchemy to use by default. Iteration will go through `itervalues()` unless otherwise decorated.

Instrumentation and Custom Types

Many custom types and existing library classes can be used as a entity collection type as-is without further ado. However, it is important to note that the instrumentation process _will_ modify the type, adding decorators around methods automatically.

The decorations are lightweight and no-op outside of relations, but they do add unneeded overhead when triggered elsewhere. When using a library class as a collection, it can be good practice to use the "trivial subclass" trick to restrict the decorations to just your usage in relations. For example:

```
class MyAwesomeList(some.great.library.AwesomeList):
    pass

# ... relation(..., collection_class=MyAwesomeList)
```

The ORM uses this approach for built-ins, quietly substituting a trivial subclass when a `list`, `set` or `dict` is used directly.

The collections package provides additional decorators and support for authoring custom types. See the *sqlalchemy.orm.collections* for more information and discussion of advanced usage and Python 2.3-compatible decoration options.

4.2.5 Configuring Loader Strategies: Lazy Loading, Eager Loading

In the *datamapping*, we introduced the concept of **Eager Loading**. We used an `option` in conjunction with the `Query` object in order to indicate that a relation should be loaded at the same time as the parent, within a single SQL query:

```
>>> jack = session.query(User).options(eagerload('addresses')).filter_by(name='jack').all(
SELECT addresses_1.id AS addresses_1_id, addresses_1.email_address AS addresses_1_email_add
addresses_1.user_id AS addresses_1_user_id, users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = ?
['jack']
```

By default, all relations are **lazy loading**. The scalar or collection attribute associated with a `relation()` contains a trigger which fires the first time the attribute is accessed, which issues a SQL call at that point:

```
>>> jack.addresses
SELECT addresses.id AS addresses_id, addresses.email_address AS addresses_email_address, ad
FROM addresses
WHERE ? = addresses.user_id
[5][<Address(u'jack@google.com')>, <Address(u'j25@yahoo.com')>]
```

The default **loader strategy** for any `relation()` is configured by the `lazy` keyword argument, which defaults to `True`. Below we set it as `False` so that the `children` relation is eager loading:

```
# eager load 'children' attribute
mapper(Parent, parent_table, properties={
    'children': relation(Child, lazy=False)
})
```

The loader strategy can be changed from lazy to eager as well as eager to lazy using the `eagerload()` and `lazyload()` query options:

```
# set children to load lazily
session.query(Parent).options(lazyload('children')).all()

# set children to load eagerly
session.query(Parent).options(eagerload('children')).all()
```

To reference a relation that is deeper than one level, separate the names by periods:

```
session.query(Parent).options(eagerload('foo.bar.bat')).all()
```

When using dot-separated names with `eagerload()`, option applies **only** to the actual attribute named, and **not** its ancestors. For example, suppose a mapping from `A` to `B` to `C`, where the relations, named `atob` and `btoc`, are both lazy-loading. A statement like the following:

```
session.query(A).options(eagerload('atob.btoc')).all()
```

will load only `A` objects to start. When the `atob` attribute on each `A` is accessed, the returned `B` objects will *eagerly* load their `C` objects.

Therefore, to modify the eager load to load both `atob` as well as `btoc`, place eagerloads for both:

```
session.query(A).options(eagerload('atob'), eagerload('atob.btoc')).all()
```

or more simply just use `eagerload_all()`:

```
session.query(A).options(eagerload_all('atob.btoc')).all()
```

There are two other loader strategies available, **dynamic loading** and **no loading**; these are described in *Working with Large Collections*.

Routing Explicit Joins/Statements into Eagerly Loaded Collections

The behavior of `eagerload()` is such that joins are created automatically, the results of which are routed into collections and scalar references on loaded objects. It is often the case that a query already includes the necessary joins which represent a particular collection or scalar reference, and the joins added by the eagerload feature are redundant - yet you'd still like the collections/references to be populated.

For this SQLAlchemy supplies the `contains_eager()` option. This option is used in the same manner as the `eagerload()` option except it is assumed that the `Query` will specify the appropriate joins explicitly. Below it's used with a `from_statement` load:

```
# mapping is the users->addresses mapping
mapper(User, users_table, properties={
    'addresses': relation(Address, addresses_table)
})

# define a query on USERS with an outer join to ADDRESSES
statement = users_table.outerjoin(addresses_table).select().apply_labels()

# construct a Query object which expects the "addresses" results
query = session.query(User).options(contains_eager('addresses'))

# get results normally
r = query.from_statement(statement)
```

It works just as well with an inline `Query.join()` or `Query.outerjoin()`:

```
session.query(User).outerjoin(User.addresses).options(contains_eager(User.addresses)).all(
```

If the "eager" portion of the statement is "aliased", the `alias` keyword argument to `contains_eager()` may be used to indicate it. This is a string alias name or reference to an actual `Alias` (or other selectable) object:

```
# use an alias of the Address entity
adalias = aliased(Address)

# construct a Query object which expects the "addresses" results
query = session.query(User).\
    outerjoin((adalias, User.addresses)).\
    options(contains_eager(User.addresses, alias=adalias))

# get results normally
r = query.all()
SELECT users.user_id AS users_user_id, users.user_name AS users_user_name, adalias.address_
adalias.user_id AS adalias_user_id, adalias.email_address AS adalias_email_address, (...otl
FROM users LEFT OUTER JOIN email_addresses AS email_addresses_1 ON users.user_id = email_ad
```

The `alias` argument is used only as a source of columns to match up to the result set. You can use it even to match up the result to arbitrary label names in a string SQL statement, by passing a selectable() which links those labels to the mapped `Table`:

```
# label the columns of the addresses table
eager_columns = select([
                    addresses.c.address_id.label('a1'),
                    addresses.c.email_address.label('a2'),
                    addresses.c.user_id.label('a3')])

# select from a raw SQL statement which uses those label names for the
# addresses table.  contains_eager() matches them up.
query = session.query(User).\
    from_statement("select users.*, addresses.address_id as a1, "
            "addresses.email_address as a2, addresses.user_id as a3 "
            "from users left outer join addresses on users.user_id=addresses.user_id").\
    options(contains_eager(User.addresses, alias=eager_columns))
```

The path given as the argument to `contains_eager()` needs to be a full path from the starting entity. For example if we were loading `Users->orders->Order->items->Item`, the string version would look like:

```
query(User).options(contains_eager('orders', 'items'))
```

Or using the class-bound descriptor:

```
query(User).options(contains_eager(User.orders, Order.items))
```

A variant on `contains_eager()` is the `contains_alias()` option, which is used in the rare case that the parent object is loaded from an alias within a user-defined SELECT statement:

```
# define an aliased UNION called 'ulist'
statement = users.select(users.c.user_id==7).union(users.select(users.c.user_id>7)).alias('

# add on an eager load of "addresses"
statement = statement.outerjoin(addresses).select().apply_labels()

# create query, indicating "ulist" is an alias for the main table, "addresses" property sho
# be eager loaded
query = session.query(User).options(contains_alias('ulist'), contains_eager('addresses'))

# results
r = query.from_statement(statement)
```

4.2.6 Working with Large Collections

The default behavior of `relation()` is to fully load the collection of items in, as according to the loading strategy of the relation. Additionally, the Session by default only knows how to delete objects which are actually present within the session. When a parent instance is marked for deletion and flushed, the Session loads its full list of child items in so that they may either be deleted as well, or have their foreign key value set to null; this is to avoid constraint violations. For large collections of child items, there are several strategies to bypass full loading of child items both at load time as well as deletion time.

Dynamic Relation Loaders

The most useful by far is the `dynamic_loader()` relation. This is a variant of `relation()` which returns a `Query` object in place of a collection when accessed. `filter()` criterion may be applied as well as limits and offsets, either explicitly or via array slices:

```
mapper(User, users_table, properties={
    'posts': dynamic_loader(Post)
})

jack = session.query(User).get(id)

# filter Jack's blog posts
posts = jack.posts.filter(Post.headline=='this is a post')

# apply array slices
posts = jack.posts[5:20]
```

The dynamic relation supports limited write operations, via the `append()` and `remove()` methods. Since the read side of the dynamic relation always queries the database, changes to the underlying collection will not be visible until the data has been flushed:

```
oldpost = jack.posts.filter(Post.headline=='old post').one()
jack.posts.remove(oldpost)

jack.posts.append(Post('new post'))
```

To place a dynamic relation on a backref, use `lazy='dynamic'`:

```
mapper(Post, posts_table, properties={
    'user': relation(User, backref=backref('posts', lazy='dynamic'))
})
```

Note that eager/lazy loading options cannot be used in conjunction dynamic relations at this time.

Setting Noload

The opposite of the dynamic relation is simply "noload", specified using `lazy=None`:

```
mapper(MyClass, table, properties={
    'children': relation(MyOtherClass, lazy=None)
})
```

Above, the `children` collection is fully writeable, and changes to it will be persisted to the database as well as locally available for reading at the time they are added. However when instances of `MyClass` are freshly loaded from the database, the `children` collection stays empty.

Using Passive Deletes

Use `passive_deletes=True` to disable child object loading on a DELETE operation, in conjunction with "ON DELETE (CASCADE|SET NULL)" on your database to automatically cascade deletes to child objects. Note that "ON DELETE" is not supported on SQLite, and requires `InnoDB` tables when using MySQL:

```
mytable = Table('mytable', meta,
    Column('id', Integer, primary_key=True),
    )
```

```
myothertable = Table('myothertable', meta,
    Column('id', Integer, primary_key=True),
    Column('parent_id', Integer),
    ForeignKeyConstraint(['parent_id'], ['mytable.id'], ondelete="CASCADE"),
    )

mapper(MyOtherClass, myothertable)

mapper(MyClass, mytable, properties={
    'children': relation(MyOtherClass, cascade="all, delete-orphan", passive_deletes=True)
})
```

When `passive_deletes` is applied, the `children` relation will not be loaded into memory when an instance of `MyClass` is marked for deletion. The `cascade="all, delete-orphan"` *will* take effect for instances of `MyOtherClass` which are currently present in the session; however for instances of `MyOtherClass` which are not loaded, SQLAlchemy assumes that "ON DELETE CASCADE" rules will ensure that those rows are deleted by the database and that no foreign key violation will occur.

4.2.7 Mutable Primary Keys / Update Cascades

As of SQLAlchemy 0.4.2, the primary key attributes of an instance can be changed freely, and will be persisted upon flush. When the primary key of an entity changes, related items which reference the primary key must also be updated as well. For databases which enforce referential integrity, it's required to use the database's ON UPDATE CASCADE functionality in order to propagate primary key changes. For those which don't, the `passive_cascades` flag can be set to `False` which instructs SQLAlchemy to issue UPDATE statements individually. The `passive_cascades` flag can also be `False` in conjunction with ON UPDATE CASCADE functionality, although in that case it issues UPDATE statements unnecessarily.

A typical mutable primary key setup might look like:

```
users = Table('users', metadata,
    Column('username', String(50), primary_key=True),
    Column('fullname', String(100)))

addresses = Table('addresses', metadata,
    Column('email', String(50), primary_key=True),
    Column('username', String(50), ForeignKey('users.username', onupdate="cascade")))

class User(object):
    pass
class Address(object):
    pass

mapper(User, users, properties={
    'addresses': relation(Address, passive_updates=False)
})
mapper(Address, addresses)
```

passive_updates is set to `True` by default. Foreign key references to non-primary key columns are supported as well.

CHAPTER

FIVE

USING THE SESSION

The *Mapper* is the entrypoint to the configurational API of the SQLAlchemy object relational mapper. But the primary object one works with when using the ORM is the `Session`.

5.1 What does the Session do ?

In the most general sense, the `Session` establishes all conversations with the database and represents a "holding zone" for all the mapped instances which you've loaded or created during its lifespan. It implements the Unit of Work pattern, which means it keeps track of all changes which occur, and is capable of **flushing** those changes to the database as appropriate. Another important facet of the `Session` is that it's also maintaining **unique** copies of each instance, where "unique" means "only one object with a particular primary key" - this pattern is called the Identity Map.

Beyond that, the `Session` implements an interface which lets you move objects in or out of the session in a variety of ways, it provides the entryway to a `Query` object which is used to query the database for data, and it also provides a transactional context for SQL operations which rides on top of the transactional capabilities of `Engine` and `Connection` objects.

5.2 Getting a Session

`Session` is a regular Python class which can be directly instantiated. However, to standardize how sessions are configured and acquired, the `sessionmaker()` function is normally used to create a top level `Session` configuration which can then be used throughout an application without the need to repeat the configurational arguments.

5.2.1 Using a sessionmaker() Configuration

The usage of `sessionmaker()` is illustrated below:

```
from sqlalchemy.orm import sessionmaker

# create a configured "Session" class
Session = sessionmaker(bind=some_engine)

# create a Session
session = Session()

# work with sess
myobject = MyObject('foo', 'bar')
```

```
session.add(myobject)
session.commit()

# close when finished
session.close()
```

Above, the `sessionmaker` call creates a class for us, which we assign to the name `Session`. This class is a subclass of the actual `sqlalchemy.orm.session.Session` class, which will instantiate with a particular bound engine.

When you write your application, place the call to `sessionmaker()` somewhere global, and then make your new `Session` class available to the rest of your application.

5.2.2 Binding Session to an Engine

In our previous example regarding `sessionmaker()`, we specified a `bind` for a particular `Engine`. If we'd like to construct a `sessionmaker()` without an engine available and bind it later on, or to specify other options to an existing `sessionmaker()`, we may use the `configure()` method:

```
# configure Session class with desired options
Session = sessionmaker()

# later, we create the engine
engine = create_engine('postgres://...')

# associate it with our custom Session class
Session.configure(bind=engine)

# work with the session
session = Session()
```

It's actually entirely optional to bind a Session to an engine. If the underlying mapped `Table` objects use "bound" metadata, the `Session` will make use of the bound engine instead (or will even use multiple engines if multiple binds are present within the mapped tables). "Bound" metadata is described at *Binding MetaData to an Engine or Connection*.

The `Session` also has the ability to be bound to multiple engines explicitly. Descriptions of these scenarios are described in *Partitioning Strategies*.

5.2.3 Binding Session to a Connection

The `Session` can also be explicitly bound to an individual database `Connection`. Reasons for doing this may include to join a `Session` with an ongoing transaction local to a specific `Connection` object, or to bypass connection pooling by just having connections persistently checked out and associated with distinct, long running sessions:

```
# global application scope.  create Session class, engine
Session = sessionmaker()

engine = create_engine('postgres://...')

...

# local scope, such as within a controller function
# connect to the database
connection = engine.connect()

# bind an individual Session to the connection
session = Session(bind=connection)
```

5.2.4 Using create_session()

As an alternative to `sessionmaker()`, `create_session()` is a function which calls the normal `Session` constructor directly. All arguments are passed through and the new `Session` object is returned:

```
session = create_session(bind=myengine, autocommit=True, autoflush=False)
```

Note that `create_session()` disables all optional "automation" by default. Called with no arguments, the session produced is not autoflushing, does not auto-expire, and does not maintain a transaction (i.e. it begins and commits a new transaction for each `flush()`). SQLAlchemy uses `create_session()` extensively within its own unit tests.

5.2.5 Configurational Arguments

Configurational arguments accepted by `sessionmaker()` and `create_session()` are the same as that of the `Session` class itself, and are described at `sqlalchemy.orm.sessionmaker()`.

Note that the defaults of `create_session()` are the opposite of that of `sessionmaker()`: autoflush and expire_on_commit are False, autocommit is True. It is recommended to use the `sessionmaker()` function instead of `create_session()`. `create_session()` is used to get a session with no automation turned on and is useful for testing.

5.3 Using the Session

5.3.1 Quickie Intro to Object States

It's helpful to know the states which an instance can have within a session:

- *Transient* - an instance that's not in a session, and is not saved to the database; i.e. it has no database identity. The only relationship such an object has to the ORM is that its class has a `mapper()` associated with it.
- *Pending* - when you `add()` a transient instance, it becomes pending. It still wasn't actually flushed to the database yet, but it will be when the next flush occurs.
- *Persistent* - An instance which is present in the session and has a record in the database. You get persistent instances by either flushing so that the pending instances become persistent, or by querying the database for existing instances (or moving persistent instances from other sessions into your local session).
- *Detached* - an instance which has a record in the database, but is not in any session. There's nothing wrong with this, and you can use objects normally when they're detached, **except** they will not be able to issue any SQL in order to load collections or attributes which are not yet loaded, or were marked as "expired".

Knowing these states is important, since the `Session` tries to be strict about ambiguous operations (such as trying to save the same object to two different sessions at the same time).

5.3.2 Frequently Asked Questions

- When do I make a `sessionmaker` ?

  Just one time, somewhere in your application's global scope. It should be looked upon as part of your application's configuration. If your application has three .py files in a package, you could, for example, place the `sessionmaker` line in your `__init__.py` file; from that point on your other modules say "from mypackage import Session". That way, everyone else just uses `Session()`, and the configuration of that session is controlled by that central point.

  If your application starts up, does imports, but does not know what database it's going to be connecting to, you can bind the `Session` at the "class" level to the engine later on, using `configure()`.

  In the examples in this section, we will frequently show the `sessionmaker` being created right above the line where we actually invoke `Session()`. But that's just for example's sake ! In reality, the `sessionmaker` would be somewhere at the module level, and your individual `Session()` calls would be sprinkled all throughout your app, such as in a web application within each controller method.

- When do I make a `Session` ?

  You typically invoke `Session()` when you first need to talk to your database, and want to save some objects or load some existing ones. Then, you work with it, save your changes, and then dispose of it....or at the very least `close()` it. It's not a "global" kind of object, and should be handled more like a "local variable", as it's generally **not** safe to use with concurrent threads. Sessions are very inexpensive to make, and don't use any resources whatsoever until they are first used...so create some !

  There is also a pattern whereby you're using a **contextual session**, this is described later in *unitof-work_contextual*. In this pattern, a helper object is maintaining a `Session` for you, most commonly one that is local to the current thread (and sometimes also local to an application instance). SQLAlchemy has worked this pattern out such that it still *looks* like you're creating a new session as you need one...so in that case, it's still a guaranteed win to just say `Session()` whenever you want a session.

- Is the Session a cache ?

  Yeee...no. It's somewhat used as a cache, in that it implements the identity map pattern, and stores objects keyed to their primary key. However, it doesn't do any kind of query caching. This means, if you say `session.query(Foo).filter_by(name='bar')`, even if `Foo(name='bar')` is right there, in the identity map, the session has no idea about that. It has to issue SQL to the database, get the rows back, and then when it sees the primary key in the row, *then* it can look in the local identity map and see that the object is already there. It's only when you say `query.get({some primary key})` that the `Session` doesn't have to issue a query.

  Additionally, the Session stores object instances using a weak reference by default. This also defeats the purpose of using the Session as a cache, unless the `weak_identity_map` flag is set to `False`.

  The `Session` is not designed to be a global object from which everyone consults as a "registry" of objects. That is the job of a **second level cache**. A good library for implementing second level caching is Memcached. It *is* possible to "sort of" use the `Session` in this manner, if you set it to be non-transactional and it never flushes any SQL, but it's not a terrific solution, since if concurrent threads load the same objects at the same time, you may have multiple copies of the same objects present in collections.

- How can I get the `Session` for a certain object ?

  Use the `object_session()` classmethod available on `Session`:

  ```
  session = Session.object_session(someobject)
  ```

- Is the session thread-safe?

  Nope. It has no thread synchronization of any kind built in, and particularly when you do a flush operation, it definitely is not open to concurrent threads accessing it, because it holds onto a single database connection at that point. If you use a session which is non-transactional for read operations only, it's still not thread-"safe", but you also wont get any catastrophic failures either, since it opens and closes connections on an as-needed basis; it's just that different threads might load the same objects independently of each other, but only one will wind up in the identity map (however, the other one might still live in a collection somewhere).

  But the bigger point here is, you should not *want* to use the session with multiple concurrent threads. That would be like having everyone at a restaurant all eat from the same plate. The session is a local "workspace" that you use for a specific set of tasks; you don't want to, or need to, share that session with other threads who are doing some other task. If, on the other hand, there are other threads participating in the same task you are, such as in a desktop graphical application, then you would be sharing the session with those threads, but you also will have implemented a proper locking scheme (or your graphical framework does) so that those threads do not collide.

5.3.3 Querying

The `query()` function takes one or more *entities* and returns a new `Query` object which will issue mapper queries within the context of this Session. An entity is defined as a mapped class, a `Mapper` object, an orm-enabled *descriptor*, or an `AliasedClass` object:

```
# query from a class
session.query(User).filter_by(name='ed').all()

# query with multiple classes, returns tuples
session.query(User, Address).join('addresses').filter_by(name='ed').all()

# query using orm-enabled descriptors
session.query(User.name, User.fullname).all()

# query from a mapper
user_mapper = class_mapper(User)
session.query(user_mapper)
```

When `Query` returns results, each object instantiated is stored within the identity map. When a row matches an object which is already present, the same object is returned. In the latter case, whether or not the row is populated onto an existing object depends upon whether the attributes of the instance have been *expired* or not. As of 0.5, a default-configured `Session` automatically expires all instances along transaction boundaries, so that with a normally isolated transaction, there shouldn't be any issue of instances representing data which is stale with regards to the current transaction.

5.3.4 Adding New or Existing Items

`add()` is used to place instances in the session. For *transient* (i.e. brand new) instances, this will have the effect of an INSERT taking place for those instances upon the next flush. For instances which are *persistent* (i.e. were loaded by this session), they are already present and do not need to be added. Instances which are *detached* (i.e. have been removed from a session) may be re-associated with a session using this method:

```
user1 = User(name='user1')
user2 = User(name='user2')
```

```
session.add(user1)
session.add(user2)

session.commit()     # write changes to the database
```

To add a list of items to the session at once, use `add_all()`:

```
session.add_all([item1, item2, item3])
```

The `add()` operation **cascades** along the `save-update` cascade. For more details see the section *unitof-work_cascades*.

5.3.5 Merging

`merge()` reconciles the current state of an instance and its associated children with existing data in the database, and returns a copy of the instance associated with the session. Usage is as follows:

```
merged_object = session.merge(existing_object)
```

When given an instance, it follows these steps:

- It examines the primary key of the instance. If it's present, it attempts to load an instance with that primary key (or pulls from the local identity map).
- If there's no primary key on the given instance, or the given primary key does not exist in the database, a new instance is created.
- The state of the given instance is then copied onto the located/newly created instance.
- The operation is cascaded to associated child items along the `merge` cascade. Note that all changes present on the given instance, including changes to collections, are merged.
- The new instance is returned.

With `merge()`, the given instance is not placed within the session, and can be associated with a different session or detached. `merge()` is very useful for taking the state of any kind of object structure without regard for its origins or current session associations and placing that state within a session. Here's two examples:

- An application which reads an object structure from a file and wishes to save it to the database might parse the file, build up the structure, and then use `merge()` to save it to the database, ensuring that the data within the file is used to formulate the primary key of each element of the structure. Later, when the file has changed, the same process can be re-run, producing a slightly different object structure, which can then be `merged()` in again, and the `Session` will automatically update the database to reflect those changes.
- A web application stores mapped entities within an HTTP session object. When each request starts up, the serialized data can be merged into the session, so that the original entity may be safely shared among requests and threads.

`merge()` is frequently used by applications which implement their own second level caches. This refers to an application which uses an in memory dictionary, or an tool like Memcached to store objects over long running spans of time. When such an object needs to exist within a `Session`, `merge()` is a good choice since it leaves the original cached object untouched. For this use case, merge provides a keyword option called `dont_load=True`. When this boolean flag is set to `True`, `merge()` will not issue any SQL to reconcile the given object against the current state of the database, thereby reducing query overhead. The limitation is that the given object and all of its children may not contain any pending changes, and it's also of course possible that newer information in the database will not be present on the merged object, since no load is issued.

5.3.6 Deleting

The `delete` method places an instance into the Session's list of objects to be marked as deleted:

```
# mark two objects to be deleted
session.delete(obj1)
session.delete(obj2)

# commit (or flush)
session.commit()
```

The big gotcha with `delete()` is that **nothing is removed from collections**. Such as, if a `User` has a collection of three `Addresses`, deleting an `Address` will not remove it from `user.addresses`:

```
>>> address = user.addresses[1]
>>> session.delete(address)
>>> session.flush()
>>> address in user.addresses
True
```

The solution is to use proper cascading:

```
mapper(User, users_table, properties={
    'addresses':relation(Address, cascade="all, delete, delete-orphan")
})
del user.addresses[1]
session.flush()
```

5.3.7 Flushing

When the `Session` is used with its default configuration, the flush step is nearly always done transparently. Specifically, the flush occurs before any individual `Query` is issued, as well as within the `commit()` call before the transaction is committed. It also occurs before a SAVEPOINT is issued when `begin_nested()` is used. The "flush-on-Query" aspect of the behavior can be disabled by constructing `sessionmaker()` with the flag `autoflush=False`.

Regardless of the autoflush setting, a flush can always be forced by issuing `flush()`:

```
session.flush()
```

`flush()` also supports the ability to flush a subset of objects which are present in the session, by passing a list of objects:

```
# saves only user1 and address2.  all other modified
# objects remain present in the session.
session.flush([user1, address2])
```

This second form of flush should be used carefully as it currently does not cascade, meaning that it will not necessarily affect other objects directly associated with the objects given.

The flush process *always* occurs within a transaction, even if the `Session` has been configured with `autocommit=True`, a setting that disables the session's persistent transactional state. If no transaction is present, `flush()` creates its own transaction and commits it. Any failures during flush will always result in a rollback of whatever transaction is present.

5.3.8 Committing

`commit()` is used to commit the current transaction. It always issues `flush()` beforehand to flush any remaining state to the database; this is independent of the "autoflush" setting. If no transaction is present, it raises an error. Note that the default behavior of the `Session` is that a transaction is always present; this behavior can be disabled by setting `autocommit=True`. In autocommit mode, a transaction can be initiated by calling the `begin()` method.

Another behavior of `commit()` is that by default it expires the state of all instances present after the commit is complete. This is so that when the instances are next accessed, either through attribute access or by them being present in a `Query` result set, they receive the most recent state. To disable this behavior, configure `sessionmaker()` with `expire_on_commit=False`.

Normally, instances loaded into the `Session` are never changed by subsequent queries; the assumption is that the current transaction is isolated so the state most recently loaded is correct as long as the transaction continues. Setting `autocommit=True` works against this model to some degree since the `Session` behaves in exactly the same way with regard to attribute state, except no transaction is present.

5.3.9 Rolling Back

`rollback()` rolls back the current transaction. With a default configured session, the post-rollback state of the session is as follows:

- All connections are rolled back and returned to the connection pool, unless the Session was bound directly to a Connection, in which case the connection is still maintained (but still rolled back).
- Objects which were initially in the *pending* state when they were added to the `Session` within the lifespan of the transaction are expunged, corresponding to their INSERT statement being rolled back. The state of their attributes remains unchanged.
- Objects which were marked as *deleted* within the lifespan of the transaction are promoted back to the *persistent* state, corresponding to their DELETE statement being rolled back. Note that if those objects were first *pending* within the transaction, that operation takes precedence instead.
- All objects not expunged are fully expired.

With that state understood, the `Session` may safely continue usage after a rollback occurs (note that this is a new feature as of version 0.5).

When a `flush()` fails, typically for reasons like primary key, foreign key, or "not nullable" constraint violations, a `rollback()` is issued automatically (it's currently not possible for a flush to continue after a partial failure). However, the flush process always uses its own transactional demarcator called a *subtransaction*, which is described more fully in the docstrings for `Session`. What it means here is that even though the database transaction has been rolled back, the end user must still issue `rollback()` to fully reset the state of the `Session`.

5.3.10 Expunging

Expunge removes an object from the Session, sending persistent instances to the detached state, and pending instances to the transient state:

```
session.expunge(obj1)
```

To remove all items, call `session.expunge_all()` (this method was formerly known as `clear()`).

5.3.11 Closing

The `close()` method issues a `expunge_all()`, and releases any transactional/connection resources. When connections are returned to the connection pool, transactional state is rolled back as well.

5.3.12 Refreshing / Expiring

To assist with the Session's "sticky" behavior of instances which are present, individual objects can have all of their attributes immediately re-loaded from the database, or marked as "expired" which will cause a re-load to occur upon the next access of any of the object's mapped attributes. This includes all relationships, so lazy-loaders will be re-initialized, eager relationships will be repopulated. Any changes marked on the object are discarded:

```
# immediately re-load attributes on obj1, obj2
session.refresh(obj1)
session.refresh(obj2)

# expire objects obj1, obj2, attributes will be reloaded
# on the next access:
session.expire(obj1)
session.expire(obj2)
```

`refresh()` and `expire()` also support being passed a list of individual attribute names in which to be refreshed. These names can reference any attribute, column-based or relation based:

```
# immediately re-load the attributes 'hello', 'world' on obj1, obj2
session.refresh(obj1, ['hello', 'world'])
session.refresh(obj2, ['hello', 'world'])

# expire the attributes 'hello', 'world' objects obj1, obj2, attributes will be reloaded
# on the next access:
session.expire(obj1, ['hello', 'world'])
session.expire(obj2, ['hello', 'world'])
```

The full contents of the session may be expired at once using `expire_all()`:

```
session.expire_all()
```

`refresh()` and `expire()` are usually not needed when working with a default-configured `Session`. The usual need is when an UPDATE or DELETE has been issued manually within the transaction using `Session.execute()`.

5.3.13 Session Attributes

The `Session` itself acts somewhat like a set-like collection. All items present may be accessed using the iterator interface:

```
for obj in session:
    print obj
```

And presence may be tested for using regular "contains" semantics:

```
if obj in session:
    print "Object is present"
```

The session is also keeping track of all newly created (i.e. pending) objects, all objects which have had changes since they were last loaded or saved (i.e. "dirty"), and everything that's been marked as deleted:

```
# pending objects recently added to the Session
session.new

# persistent objects which currently have changes detected
# (this collection is now created on the fly each time the property is called)
session.dirty

# persistent objects that have been marked as deleted via session.delete(obj)
session.deleted
```

Note that objects within the session are by default *weakly referenced*. This means that when they are dereferenced in the outside application, they fall out of scope from within the `Session` as well and are subject to garbage collection by the Python interpreter. The exceptions to this include objects which are pending, objects which are marked as deleted, or persistent objects which have pending changes on them. After a full flush, these collections are all empty, and all objects are again weakly referenced. To disable the weak referencing behavior and force all objects within the session to remain until explicitly expunged, configure `sessionmaker()` with the `weak_identity_map=False` setting.

5.4 Cascades

Mappers support the concept of configurable *cascade* behavior on `relation()` constructs. This behavior controls how the Session should treat the instances that have a parent-child relationship with another instance that is operated upon by the Session. Cascade is indicated as a comma-separated list of string keywords, with the possible values `all`, `delete`, `save-update`, `refresh-expire`, `merge`, `expunge`, and `delete-orphan`.

Cascading is configured by setting the `cascade` keyword argument on a `relation()`:

```
mapper(Order, order_table, properties={
    'items' : relation(Item, items_table, cascade="all, delete-orphan"),
    'customer' : relation(User, users_table, user_orders_table, cascade="save-update"),
})
```

The above mapper specifies two relations, `items` and `customer`. The `items` relationship specifies "all, delete-orphan" as its `cascade` value, indicating that all `add`, `merge`, `expunge`, `refresh` `delete` and `expire` operations performed on a parent `Order` instance should also be performed on the child `Item` instances attached to it. The `delete-orphan` cascade value additionally indicates that if an `Item` instance is no longer associated with an `Order`, it should also be deleted. The "all, delete-orphan" cascade argument allows a so-called *lifecycle* relationship between an `Order` and an `Item` object.

The `customer` relationship specifies only the "save-update" cascade value, indicating most operations will not be cascaded from a parent `Order` instance to a child `User` instance except for the `add()` operation. "save-update" cascade indicates that an `add()` on the parent will cascade to all child items, and also that items added to a parent which is already present in the session will also be added.

Note that the `delete-orphan` cascade only functions for relationships where the target object can have a single parent at a time, meaning it is only appropriate for one-to-one or one-to-many relationships. For a `relation()`

which establishes one-to-one via a local foreign key, i.e. a many-to-one that stores only a single parent, or one-to-one/one-to-many via a "secondary" (association) table, a warning will be issued if `delete-orphan` is configured. To disable this warning, also specify the `single_parent=True` flag on the relationship, which constrains objects to allow attachment to only one parent at a time.

The default value for `cascade` on `relation()` is `save-update, merge`.

5.5 Managing Transactions

The `Session` manages transactions across all engines associated with it. As the `Session` receives requests to execute SQL statements using a particular `Engine` or `Connection`, it adds each individual `Engine` encountered to its transactional state and maintains an open connection for each one (note that a simple application normally has just one `Engine`). At commit time, all unflushed data is flushed, and each individual transaction is committed. If the underlying databases support two-phase semantics, this may be used by the Session as well if two-phase transactions are enabled.

Normal operation ends the transactional state using the `rollback()` or `commit()` methods. After either is called, the `Session` starts a new transaction:

```
Session = sessionmaker()
session = Session()
try:
    item1 = session.query(Item).get(1)
    item2 = session.query(Item).get(2)
    item1.foo = 'bar'
    item2.bar = 'foo'

    # commit- will immediately go into a new transaction afterwards
    session.commit()
except:
    # rollback - will immediately go into a new transaction afterwards.
    session.rollback()
```

A session which is configured with `autocommit=True` may be placed into a transaction using `begin()`. With an `autocommit=True` session that's been placed into a transaction using `begin()`, the session releases all connection resources after a `commit()` or `rollback()` and remains transaction-less (with the exception of flushes) until the next `begin()` call:

```
Session = sessionmaker(autocommit=True)
session = Session()
session.begin()
try:
    item1 = session.query(Item).get(1)
    item2 = session.query(Item).get(2)
    item1.foo = 'bar'
    item2.bar = 'foo'
    session.commit()
except:
    session.rollback()
    raise
```

The `begin()` method also returns a transactional token which is compatible with the Python 2.6 `with` statement:

```
Session = sessionmaker(autocommit=True)
session = Session()
with session.begin():
    item1 = session.query(Item).get(1)
    item2 = session.query(Item).get(2)
    item1.foo = 'bar'
    item2.bar = 'foo'
```

5.5.1 Using SAVEPOINT

SAVEPOINT transactions, if supported by the underlying engine, may be delineated using the `begin_nested()` method:

```
Session = sessionmaker()
session = Session()
session.add(u1)
session.add(u2)

session.begin_nested() # establish a savepoint
session.add(u3)
session.rollback()  # rolls back u3, keeps u1 and u2

session.commit() # commits u1 and u2
```

`begin_nested()` may be called any number of times, which will issue a new SAVEPOINT with a unique identifier for each call. For each `begin_nested()` call, a corresponding `rollback()` or `commit()` must be issued.

When `begin_nested()` is called, a `flush()` is unconditionally issued (regardless of the `autoflush` setting). This is so that when a `rollback()` occurs, the full state of the session is expired, thus causing all subsequent attribute/instance access to reference the full state of the `Session` right before `begin_nested()` was called.

5.5.2 Enabling Two-Phase Commit

Finally, for MySQL, PostgreSQL, and soon Oracle as well, the session can be instructed to use two-phase commit semantics. This will coordinate the committing of transactions across databases so that the transaction is either committed or rolled back in all databases. You can also `prepare()` the session for interacting with transactions not managed by SQLAlchemy. To use two phase transactions set the flag `twophase=True` on the session:

```
engine1 = create_engine('postgres://db1')
engine2 = create_engine('postgres://db2')

Session = sessionmaker(twophase=True)

# bind User operations to engine 1, Account operations to engine 2
Session.configure(binds={User:engine1, Account:engine2})

session = Session()

# .... work with accounts and users

# commit.  session will issue a flush to all DBs, and a prepare step to all DBs,
# before committing both transactions
session.commit()
```

5.6 Embedding SQL Insert/Update Expressions into a Flush

This feature allows the value of a database column to be set to a SQL expression instead of a literal value. It's especially useful for atomic updates, calling stored procedures, etc. All you do is assign an expression to an attribute:

```
class SomeClass(object):
    pass
mapper(SomeClass, some_table)

someobject = session.query(SomeClass).get(5)

# set 'value' attribute to a SQL expression adding one
someobject.value = some_table.c.value + 1

# issues "UPDATE some_table SET value=value+1"
session.commit()
```

This technique works both for INSERT and UPDATE statements. After the flush/commit operation, the `value` attribute on `someobject` above is expired, so that when next accessed the newly generated value will be loaded from the database.

5.7 Using SQL Expressions with Sessions

SQL expressions and strings can be executed via the `Session` within its transactional context. This is most easily accomplished using the `execute()` method, which returns a `ResultProxy` in the same manner as an `Engine` or `Connection`:

```
Session = sessionmaker(bind=engine)
session = Session()

# execute a string statement
result = session.execute("select * from table where id=:id", {'id':7})

# execute a SQL expression construct
result = session.execute(select([mytable]).where(mytable.c.id==7))
```

The current `Connection` held by the `Session` is accessible using the `connection()` method:

```
connection = session.connection()
```

The examples above deal with a `Session` that's bound to a single `Engine` or `Connection`. To execute statements using a `Session` which is bound either to multiple engines, or none at all (i.e. relies upon bound metadata), both `execute()` and `connection()` accept a `mapper` keyword argument, which is passed a mapped class or `Mapper` instance, which is used to locate the proper context for the desired engine:

```
Session = sessionmaker()
session = Session()

# need to specify mapper or class when executing
result = session.execute("select * from table where id=:id", {'id':7}, mapper=MyMappedClas
result = session.execute(select([mytable], mytable.c.id==7), mapper=MyMappedClass)

connection = session.connection(MyMappedClass)
```

5.8 Joining a Session into an External Transaction

If a `Connection` is being used which is already in a transactional state (i.e. has a `Transaction`), a `Session` can be made to participate within that transaction by just binding the `Session` to that `Connection`:

```
Session = sessionmaker()

# non-ORM connection + transaction
conn = engine.connect()
trans = conn.begin()

# create a Session, bind to the connection
session = Session(bind=conn)

# ... work with session

session.commit() # commit the session
session.close()  # close it out, prohibit further actions

trans.commit() # commit the actual transaction
```

Note that above, we issue a `commit()` both on the `Session` as well as the `Transaction`. This is an example of where we take advantage of `Connection`'s ability to maintain *subtransactions*, or nested begin/commit pairs. The `Session` is used exactly as though it were managing the transaction on its own; its `commit()` method issues its `flush()`, and commits the subtransaction. The subsequent transaction the `Session` starts after commit will not begin until it's next used. Above we issue a `close()` to prevent this from occurring. Finally, the actual transaction is committed using `Transaction.commit()`.

When using the `threadlocal` engine context, the process above is simplified; the `Session` uses the same connection/transaction as everyone else in the current thread, whether or not you explicitly bind it:

```
engine = create_engine('postgres://mydb', strategy="threadlocal")
engine.begin()

session = Session()  # session takes place in the transaction like everyone else

# ... go nuts

engine.commit() # commit the transaction
```

5.9 Contextual/Thread-local Sessions

A common need in applications, particularly those built around web frameworks, is the ability to "share" a `Session` object among disparate parts of an application, without needing to pass the object explicitly to all method and function calls. What you're really looking for is some kind of "global" session object, or at least "global" to all the parts of an application which are tasked with servicing the current request. For this pattern, SQLAlchemy provides the ability to enhance the `Session` class generated by `sessionmaker()` to provide auto-contextualizing support. This means that whenever you create a `Session` instance with its constructor, you get an *existing* `Session` object which is bound to some "context". By default, this context is the current thread. This feature is what previously was accomplished using the `sessioncontext` SQLAlchemy extension.

5.9.1 Creating a Thread-local Context

The `scoped_session()` function wraps around the `sessionmaker()` function, and produces an object which behaves the same as the `Session` subclass returned by `sessionmaker()`:

```
from sqlalchemy.orm import scoped_session, sessionmaker
Session = scoped_session(sessionmaker())
```

However, when you instantiate this `Session` "class", in reality the object is pulled from a threadlocal variable, or if it doesn't exist yet, it's created using the underlying class generated by `sessionmaker()`:

```
>>> # call Session() the first time.  the new Session instance is created.
>>> session = Session()

>>> # later, in the same application thread, someone else calls Session()
>>> session2 = Session()

>>> # the two Session objects are *the same* object
>>> session is session2
True
```

Since the `Session()` constructor now returns the same `Session` object every time within the current thread, the object returned by `scoped_session()` also implements most of the `Session` methods and properties at the "class" level, such that you don't even need to instantiate `Session()`:

```
# create some objects
u1 = User()
u2 = User()

# save to the contextual session, without instantiating
Session.add(u1)
Session.add(u2)

# view the "new" attribute
assert u1 in Session.new

# commit changes
Session.commit()
```

The contextual session may be disposed of by calling `Session.remove()`:

```
# remove current contextual session
Session.remove()
```

After `remove()` is called, the next operation with the contextual session will start a new `Session` for the current thread.

5.9.2 Lifespan of a Contextual Session

A (really, really) common question is when does the contextual session get created, when does it get disposed ? We'll consider a typical lifespan as used in a web application:

```
Web Server          Web Framework        User-defined Controller Call
--------------      --------------       ------------------------------
web request    ->
                    call controller ->   # call Session().  this establishes a new,
                                         # contextual Session.
                                         session = Session()

                                         # load some objects, save some changes
                                         objects = session.query(MyClass).all()

                                         # some other code calls Session, it's the
                                         # same contextual session as "sess"
                                         session2 = Session()
                                         session2.add(foo)
                                         session2.commit()

                                         # generate content to be returned
                                         return generate_content()
                    Session.remove() <-
web response   <-
```

The above example illustrates an explicit call to `Session.remove()`. This has the effect such that each web request starts fresh with a brand new session. When integrating with a web framework, there's actually many options on how to proceed for this step, particularly as of version 0.5:

- Session.remove() - this is the most cut and dry approach; the `Session` is thrown away, all of its transactional/connection resources are closed out, everything within it is explicitly gone. A new `Session` will be used on the next request.
- Session.close() - Similar to calling `remove()`, in that all objects are explicitly expunged and all transactional/connection resources closed, except the actual `Session` object hangs around. It doesn't make too much difference here unless the start of the web request would like to pass specific options to the initial construction of `Session()`, such as a specific `Engine` to bind to.
- Session.commit() - In this case, the behavior is that any remaining changes pending are flushed, and the transaction is committed. The full state of the session is expired, so that when the next web request is started, all data will be reloaded. In reality, the contents of the `Session` are weakly referenced anyway so its likely that it will be empty on the next request in any case.
- Session.rollback() - Similar to calling commit, except we assume that the user would have called commit explicitly if that was desired; the `rollback()` ensures that no transactional state remains and expires all data, in the case that the request was aborted and did not roll back itself.
- do nothing - this is a valid option as well. The controller code is responsible for doing one of the above steps at the end of the request.

Scoped Session API docs: `sqlalchemy.orm.scoped_session()`

5.10 Partitioning Strategies

5.10.1 Vertical Partitioning

Vertical partitioning places different kinds of objects, or different tables, across multiple databases:

```
engine1 = create_engine('postgres://db1')
engine2 = create_engine('postgres://db2')

Session = sessionmaker(twophase=True)

# bind User operations to engine 1, Account operations to engine 2
Session.configure(binds={User:engine1, Account:engine2})

session = Session()
```

5.10.2 Horizontal Partitioning

Horizontal partitioning partitions the rows of a single table (or a set of tables) across multiple databases.

See the "sharding" example in attribute_shard.py

5.11 Extending Session

Extending the session can be achieved through subclassing as well as through a simple extension class, which resembles the style of *Extending Mapper* called `SessionExtension`. See the docstrings for more information on this class' methods.

Basic usage is similar to `MapperExtension`:

```
class MySessionExtension(SessionExtension):
    def before_commit(self, session):
        print "before commit!"

Session = sessionmaker(extension=MySessionExtension())
```

or with `create_session()`:

```
session = create_session(extension=MySessionExtension())
```

The same `SessionExtension` instance can be used with any number of sessions.

CHAPTER

SIX

DATABASE ENGINES

The **Engine** is the starting point for any SQLAlchemy application. It's "home base" for the actual database and its DBAPI, delivered to the SQLAlchemy application through a connection pool and a **Dialect**, which describes how to talk to a specific kind of database/DBAPI combination.

The general structure is this:

```
                                     +------------+                        __________
                                 /---|   Pool     |---\                   (__________)
             +--------------+    /    +------------+    \    +---------+   |          |
connect() <--|   Engine     |---x                        x----| DBAPI   |---| database |
             +--------------+    \    +------------+    /    +---------+   |          |
                                 \---|  Dialect   |---/                   |__________|
                                     +------------+                       (__________)
```

Where above, a `Engine` references both a `Dialect` and `Pool`, which together interpret the DBAPI's module functions as well as the behavior of the database.

Creating an engine is just a matter of issuing a single call, `create_engine()`:

```
engine = create_engine('postgres://scott:tiger@localhost:5432/mydatabase')
```

The above engine invokes the `postgres` dialect and a connection pool which references `localhost:5432`.

The engine can be used directly to issue SQL to the database. The most generic way is to use connections, which you get via the `connect()` method:

```
connection = engine.connect()
result = connection.execute("select username from users")
for row in result:
    print "username:", row['username']
connection.close()
```

The connection is an instance of `Connection`, which is a **proxy** object for an actual DBAPI connection. The returned result is an instance of `ResultProxy`, which acts very much like a DBAPI cursor.

When you say `engine.connect()`, a new `Connection` object is created, and a DBAPI connection is retrieved from the connection pool. Later, when you call `connection.close()`, the DBAPI connection is returned to the pool; nothing is actually "closed" from the perspective of the database.

To execute some SQL more quickly, you can skip the `Connection` part and just say:

```
result = engine.execute("select username from users")
for row in result:
    print "username:", row['username']
result.close()
```

Where above, the `execute()` method on the `Engine` does the `connect()` part for you, and returns the `ResultProxy` directly. The actual `Connection` is *inside* the `ResultProxy`, waiting for you to finish reading the result. In this case, when you `close()` the `ResultProxy`, the underlying `Connection` is closed, which returns the DBAPI connection to the pool.

To summarize the above two examples, when you use a `Connection` object, it's known as **explicit execution**. When you don't see the `Connection` object, but you still use the `execute()` method on the `Engine`, it's called **explicit, connectionless execution**. A third variant of execution also exists called **implicit execution**; this will be described later.

The `Engine` and `Connection` can do a lot more than what we illustrated above; SQL strings are only its most rudimentary function. Later chapters will describe how "constructed SQL" expressions can be used with engines; in many cases, you don't have to deal with the `Engine` at all after it's created. The Object Relational Mapper (ORM), an optional feature of SQLAlchemy, also uses the `Engine` in order to get at connections; that's also a case where you can often create the engine once, and then forget about it.

6.1 Supported Databases

Recall that the `Dialect` is used to describe how to talk to a specific kind of database. Dialects are included with SQLAlchemy for SQLite, Postgres, MySQL, MS-SQL, Firebird, Informix, and Oracle; these can each be seen as a Python module present in the :mod:`~sqlalchemy.databases` package. Each dialect requires the appropriate DBAPI drivers to be installed separately.

Downloads for each DBAPI at the time of this writing are as follows:

- Postgres: psycopg2
- SQLite: sqlite3 (included in Python 2.5 or greater) pysqlite
- MySQL: MySQLDB
- Oracle: cx_Oracle
- MS-SQL, MSAccess: pyodbc (recommended) adodbapi pymssql
- Firebird: kinterbasdb
- Informix: informixdb
- DB2/Informix IDS: ibm-db
- Sybase: TODO
- MAXDB: TODO

The SQLAlchemy Wiki contains a page of database notes, describing whatever quirks and behaviors have been observed. Its a good place to check for issues with specific databases. Database Notes

6.2 create_engine() URL Arguments

SQLAlchemy indicates the source of an Engine strictly via RFC-1738 style URLs, combined with optional keyword arguments to specify options for the Engine. The form of the URL is:

> driver://username:password@host:port/database

Available drivernames are `sqlite`, `mysql`, `postgres`, `oracle`, `mssql`, and `firebird`. For sqlite, the database name is the filename to connect to, or the special name ":memory:" which indicates an in-memory database. The URL is typically sent as a string to the `create_engine()` function:

```
# postgres
pg_db = create_engine('postgres://scott:tiger@localhost:5432/mydatabase')

# sqlite (note the four slashes for an absolute path)
sqlite_db = create_engine('sqlite:////absolute/path/to/database.txt')
sqlite_db = create_engine('sqlite:///relative/path/to/database.txt')
sqlite_db = create_engine('sqlite://')  # in-memory database
sqlite_db = create_engine('sqlite://:memory:')  # the same

# mysql
mysql_db = create_engine('mysql://localhost/foo')

# oracle
oracle_db = create_engine('oracle://scott:tiger@host:port/dbname?key1=value1&key2=value2')

# oracle via TNS name
oracle_db = create_engine('oracle://scott:tiger@tnsname')
oracle_db = create_engine('oracle://scott:tiger@tnsname/?key1=value1&key2=value2')

# oracle will feed host/port/SID into cx_oracle.makedsn
oracle_db = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

# mssql
mssql_db = create_engine('mssql://username:password@localhost/database')

# mssql via a DSN connection
mssql_db = create_engine('mssql://mydsn')
mssql_db = create_engine('mssql://username:password@mydsn')
```

The `Engine` will ask the connection pool for a connection when the `connect()` or `execute()` methods are called. The default connection pool, `QueuePool`, as well as the default connection pool used with SQLite, `SingletonThreadPool`, will open connections to the database on an as-needed basis. As concurrent statements are executed, `QueuePool` will grow its pool of connections to a default size of five, and will allow a default "overflow" of ten. Since the `Engine` is essentially "home base" for the connection pool, it follows that you should keep a single `Engine` per database established within an application, rather than creating a new one for each connection.

6.2.1 Custom DBAPI connect() arguments

Custom arguments used when issuing the `connect()` call to the underlying DBAPI may be issued in three distinct ways. String-based arguments can be passed directly from the URL string as query arguments:

```
db = create_engine('postgres://scott:tiger@localhost/test?argument1=foo&argument2=bar')
```

If SQLAlchemy's database connector is aware of a particular query argument, it may convert its type from string to its proper type.

`create_engine` also takes an argument `connect_args` which is an additional dictionary that will be passed to `connect()`. This can be used when arguments of a type other than string are required, and SQLAlchemy's database connector has no type conversion logic present for that parameter:

```
db = create_engine('postgres://scott:tiger@localhost/test', connect_args = {'argument1':17,
```

The most customizable connection method of all is to pass a `creator` argument, which specifies a callable that returns a DBAPI connection:

```
def connect():
    return psycopg.connect(user='scott', host='localhost')

db = create_engine('postgres://', creator=connect)
```

6.3 Database Engine Options

Keyword options can also be specified to `create_engine()`, following the string URL as follows:

```
db = create_engine('postgres://...', encoding='latin1', echo=True)
```

Options common to all database dialects are described at `create_engine()`.

6.4 More On Connections

Recall from the beginning of this section that the Engine provides a `connect()` method which returns a `Connection` object. `Connection` is a *proxy* object which maintains a reference to a DBAPI connection instance. The `close()` method on `Connection` does not actually close the DBAPI connection, but instead returns it to the connection pool referenced by the `Engine`. `Connection` will also automatically return its resources to the connection pool when the object is garbage collected, i.e. its `__del__()` method is called. When using the standard C implementation of Python, this method is usually called immediately as soon as the object is dereferenced. With other Python implementations such as Jython, this is not so guaranteed.

The `execute()` methods on both `Engine` and `Connection` can also receive SQL clause constructs as well:

```
connection = engine.connect()
result = connection.execute(select([table1], table1.c.col1==5))
for row in result:
    print row['col1'], row['col2']
connection.close()
```

The above SQL construct is known as a `select()`. The full range of SQL constructs available are described in *sql*.

Both `Connection` and `Engine` fulfill an interface known as `Connectable` which specifies common functionality between the two objects, namely being able to call `connect()` to return a `Connection` object (`Connection` just returns itself), and being able to call `execute()` to get a result set. Following this, most SQLAlchemy functions and objects which accept an `Engine` as a parameter or attribute with which to execute SQL will also accept a `Connection`. As of SQLAlchemy 0.3.9, this argument is named `bind`:

```
engine = create_engine('sqlite:///:memory:')

# specify some Table metadata
metadata = MetaData()
table = Table('sometable', metadata, Column('col1', Integer))

# create the table with the Engine
table.create(bind=engine)

# drop the table with a Connection off the Engine
connection = engine.connect()
table.drop(bind=connection)
```

Connection facts:

- the Connection object is **not thread-safe**. While a Connection can be shared among threads using properly synchronized access, this is also not recommended as many DBAPIs have issues with, if not outright disallow, sharing of connection state between threads.
- The Connection object represents a single dbapi connection checked out from the connection pool. In this state, the connection pool has no affect upon the connection, including its expiration or timeout state. For the connection pool to properly manage connections, **connections should be returned to the connection pool (i.e. ``connection.close()``) whenever the connection is not in use**. If your application has a need for management of multiple connections or is otherwise long running (this includes all web applications, threaded or not), don't hold a single connection open at the module level.

6.5 Using Transactions with Connection

The `Connection` object provides a `begin()` method which returns a `Transaction` object. This object is usually used within a try/except clause so that it is guaranteed to `rollback()` or `commit()`:

```
trans = connection.begin()
try:
    r1 = connection.execute(table1.select())
    connection.execute(table1.insert(), col1=7, col2='this is some data')
    trans.commit()
except:
    trans.rollback()
    raise
```

The `Transaction` object also handles "nested" behavior by keeping track of the outermost begin/commit pair. In this example, two functions both issue a transaction on a Connection, but only the outermost Transaction object actually takes effect when it is committed.

```
# method_a starts a transaction and calls method_b
def method_a(connection):
    trans = connection.begin() # open a transaction
    try:
        method_b(connection)
        trans.commit()  # transaction is committed here
    except:
        trans.rollback() # this rolls back the transaction unconditionally
        raise

# method_b also starts a transaction
def method_b(connection):
    trans = connection.begin() # open a transaction - this runs in the context of method_a
    try:
        connection.execute("insert into mytable values ('bat', 'lala')")
        connection.execute(mytable.insert(), col1='bat', col2='lala')
        trans.commit()  # transaction is not committed yet
    except:
        trans.rollback() # this rolls back the transaction unconditionally
        raise

# open a Connection and call method_a
conn = engine.connect()
method_a(conn)
conn.close()
```

Above, `method_a` is called first, which calls `connection.begin()`. Then it calls `method_b`. When `method_b` calls `connection.begin()`, it just increments a counter that is decremented when it calls `commit()`. If either `method_a` or `method_b` calls `rollback()`, the whole transaction is rolled back. The transaction is not committed until `method_a` calls the `commit()` method. This "nesting" behavior allows the creation of functions which "guarantee" that a transaction will be used if one was not already available, but will automatically participate in an enclosing transaction if one exists.

Note that SQLAlchemy's Object Relational Mapper also provides a way to control transaction scope at a higher level; this is described in *unitofwork_transaction*. Transaction Facts:

- the Transaction object, just like its parent Connection, is **not thread-safe**.
- SQLAlchemy 0.4 will feature transactions with two-phase commit capability as well as SAVEPOINT capability.

6.5.1 Understanding Autocommit

The above transaction example illustrates how to use `Transaction` so that several executions can take part in the same transaction. What happens when we issue an INSERT, UPDATE or DELETE call without using `Transaction`? The answer is **autocommit**. While many DBAPIs implement a flag called `autocommit`, the current SQLAlchemy behavior is such that it implements its own autocommit. This is achieved by detecting statements which represent data-changing operations, i.e. INSERT, UPDATE, DELETE, etc., and then issuing a COMMIT automatically if no transaction is in progress. The detection is based on compiled statement attributes, or in the case of a text-only statement via regular expressions.

```
conn = engine.connect()
conn.execute("INSERT INTO users VALUES (1, 'john')")  # autocommits
```

6.6 Connectionless Execution, Implicit Execution

Recall from the first section we mentioned executing with and without a `Connection`. `Connectionless` execution refers to calling the `execute()` method on an object which is not a `Connection`, which could be on the `Engine` itself, or could be a constructed SQL object. When we say "implicit", we mean that we are calling the `execute()` method on an object which is neither a `Connection` nor an `Engine` object; this can only be used with constructed SQL objects which have their own `execute()` method, and can be "bound" to an `Engine`. A description of "constructed SQL objects" may be found in *sql*.

A summary of all three methods follows below. First, assume the usage of the following `MetaData` and `Table` objects; while we haven't yet introduced these concepts, for now you only need to know that we are representing a database table, and are creating an "executable" SQL construct which issues a statement to the database. These objects are described in *metadata*.

```
meta = MetaData()
users_table = Table('users', meta,
    Column('id', Integer, primary_key=True),
    Column('name', String(50))
)
```

Explicit execution delivers the SQL text or constructed SQL expression to the `execute()` method of `Connection`:

```
engine = create_engine('sqlite:///file.db')
connection = engine.connect()
result = connection.execute(users_table.select())
for row in result:
    # ....
connection.close()
```

Explicit, connectionless execution delivers the expression to the `execute()` method of `Engine`:

```
engine = create_engine('sqlite:///file.db')
result = engine.execute(users_table.select())
for row in result:
    # ....
result.close()
```

Implicit execution is also connectionless, and calls the `execute()` method on the expression itself, utilizing the fact that either an `Engine` or `Connection` has been *bound* to the expression object (binding is discussed further in the next section, *metadata*):

```
engine = create_engine('sqlite:///file.db')
meta.bind = engine
result = users_table.select().execute()
for row in result:
    # ....
result.close()
```

In both "connectionless" examples, the `Connection` is created behind the scenes; the `ResultProxy` returned by the `execute()` call references the `Connection` used to issue the SQL statement. When we issue `close()` on the `ResultProxy`, or if the result set object falls out of scope and is garbage collected, the underlying `Connection` is closed for us, resulting in the DBAPI connection being returned to the pool.

6.6.1 Using the Threadlocal Execution Strategy

The "threadlocal" engine strategy is used by non-ORM applications which wish to bind a transaction to the current thread, such that all parts of the application can participate in that transaction implicitly without the need to explicitly reference a `Connection`. "threadlocal" is designed for a very specific pattern of use, and is not appropriate unless this very specfic pattern, described below, is what's desired. It has **no impact** on the "thread safety" of SQLAlchemy components or one's application. It also should not be used when using an ORM `Session` object, as the `Session` itself represents an ongoing transaction and itself handles the job of maintaining connection and transactional resources.

Enabling `threadlocal` is achieved as follows:

```
db = create_engine('mysql://localhost/test', strategy='threadlocal')
```

When the engine above is used in a "connectionless" style, meaning `engine.execute()` is called, a DBAPI connection is retrieved from the connection pool and then associated with the current thread. Subsequent operations on the `Engine` while the DBAPI connection remains checked out will make use of the *same* DBAPI connection object. The connection stays allocated until all returned `ResultProxy` objects are closed, which occurs for a particular `ResultProxy` after all pending results are fetched, or immediately for an operation which returns no rows (such as an INSERT).

```
# execute one statement and receive results.  r1 now references a DBAPI connection resourc
r1 = db.execute("select * from table1")

# execute a second statement and receive results.  r2 now references the *same* resource a
r2 = db.execute("select * from table2")

# fetch a row on r1 (assume more results are pending)
row1 = r1.fetchone()

# fetch a row on r2 (same)
row2 = r2.fetchone()

# close r1.  the connection is still held by r2.
r1.close()

# close r2.  with no more references to the underlying connection resources, they
# are returned to the pool.
r2.close()
```

The above example does not illustrate any pattern that is particularly useful, as it is not a frequent occurence that two execute/result fetching operations "leapfrog" one another. There is a slight savings of connection pool checkout overhead between the two operations, and an implicit sharing of the same transactional context, but since there is no explicitly declared transaction, this association is short lived.

The real usage of "threadlocal" comes when we want several operations to occur within the scope of a shared transaction. The `Engine` now has `begin()`, `commit()` and `rollback()` methods which will retrieve a connection resource from the pool and establish a new transaction, maintaining the connection against the current thread until the transaction is committed or rolled back:

```
db.begin()
try:
    call_operation1()
    call_operation2()
    db.commit()
except:
    db.rollback()
```

`call_operation1()` and `call_operation2()` can make use of the `Engine` as a global variable, using the "connectionless" execution style, and their operations will participate in the same transaction:

```
def call_operation1():
    engine.execute("insert into users values (?, ?)", 1, "john")

def call_operation2():
    users.update(users.c.user_id==5).execute(name='ed')
```

When using threadlocal, operations that do call upon the `engine.connect()` method will receive a `Connection` that is **outside** the scope of the transaction. This can be used for operations such as logging the status of an operation regardless of transaction success:

```
db.begin()
conn = db.connect()
try:
    conn.execute(log_table.insert(), message="Operation started")
    call_operation1()
    call_operation2()
    db.commit()
    conn.execute(log_table.insert(), message="Operation succeeded")
except:
    db.rollback()
    conn.execute(log_table.insert(), message="Operation failed")
finally:
    conn.close()
```

Functions which are written to use an explicit `Connection` object, but wish to participate in the threadlocal transaction, can receive their `Connection` object from the `contextual_connect()` method, which returns a `Connection` that is **inside** the scope of the transaction:

```
conn = db.contextual_connect()
call_operation3(conn)
conn.close()
```

Calling `close()` on the "contextual" connection does not release the connection resources to the pool if other resources are making use of it. A resource-counting mechanism is employed so that the connection is released back to the pool only when all users of that connection, including the transaction established by `engine.begin()`, have been completed.

So remember - if you're not sure if you need to use `strategy="threadlocal"` or not, the answer is **no** ! It's driven by a specific programming pattern that is generally not the norm.

6.7 Configuring Logging

Python's standard logging module is used to implement informational and debug log output with SQLAlchemy. This allows SQLAlchemy's logging to integrate in a standard way with other applications and libraries. The `echo` and `echo_pool` flags that are present on `create_engine()`, as well as the `echo_uow` flag used on `Session`, all interact with regular loggers.

This section assumes familiarity with the above linked logging module. All logging performed by SQLAlchemy exists underneath the `sqlalchemy` namespace, as used by `logging.getLogger('sqlalchemy')`. When logging has been configured (i.e. such as via `logging.basicConfig()`), the general namespace of SA loggers that can be turned on is as follows:

- `sqlalchemy.engine` - controls SQL echoing. set to `logging.INFO` for SQL query output, `logging.DEBUG` for query + result set output.
- `sqlalchemy.pool` - controls connection pool logging. set to `logging.INFO` or lower to log connection pool checkouts/checkins.
- **`sqlalchemy.orm` - controls logging of various ORM functions. set to `logging.INFO` for configurational logging as we**
  `sqlalchemy.orm.attributes` - logs certain instrumented attribute operations, such as triggered callables
  - `sqlalchemy.orm.mapper` - logs Mapper configuration and operations
  - `sqlalchemy.orm.unitofwork` - logs flush() operations, including dependency sort graphs and other operations
  - `sqlalchemy.orm.strategies` - logs relation loader operations (i.e. lazy and eager loads)
  - `sqlalchemy.orm.sync` - logs synchronization of attributes from parent to child instances during a flush()

For example, to log SQL queries as well as unit of work debugging:

```
import logging

logging.basicConfig()
logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)
logging.getLogger('sqlalchemy.orm.unitofwork').setLevel(logging.DEBUG)
```

By default, the log level is set to `logging.ERROR` within the entire `sqlalchemy` namespace so that no log operations occur, even within an application that has logging enabled otherwise.

The `echo` flags present as keyword arguments to `create_engine()` and others as well as the `echo` property on `Engine`, when set to `True`, will first attempt to ensure that logging is enabled. Unfortunately, the `logging` module provides no way of determining if output has already been configured (note we are referring to if a logging configuration has been set up, not just that the logging level is set). For this reason, any `echo=True` flags will result in a call to `logging.basicConfig()` using sys.stdout as the destination. It also sets up a default format using the level name, timestamp, and logger name. Note that this configuration has the affect of being configured **in addition** to any existing logger configurations. Therefore, **when using Python logging, ensure all echo flags are set to False at all times**, to avoid getting duplicate log lines.

CHAPTER

SEVEN

DATABASE META DATA

7.1 Describing Databases with MetaData

The core of SQLAlchemy's query and object mapping operations are supported by **database metadata**, which is comprised of Python objects that describe tables and other schema-level objects. These objects can be created by explicitly naming the various components and their properties, using the Table, Column, ForeignKey, Index, and Sequence objects imported from `sqlalchemy.schema`. There is also support for **reflection** of some entities, which means you only specify the *name* of the entities and they are recreated from the database automatically.

A collection of metadata entities is stored in an object aptly named `MetaData`:

```
from sqlalchemy import *

metadata = MetaData()
```

To represent a Table, use the `Table` class:

```
users = Table('users', metadata,
    Column('user_id', Integer, primary_key = True),
    Column('user_name', String(16), nullable = False),
    Column('email_address', String(60), key='email'),
    Column('password', String(20), nullable = False)
)

user_prefs = Table('user_prefs', metadata,
    Column('pref_id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey("users.user_id"), nullable=False),
    Column('pref_name', String(40), nullable=False),
    Column('pref_value', String(100))
)
```

The specific datatypes for each Column, such as Integer, String, etc. are described in *types*, and exist within the module `sqlalchemy.types` as well as the global `sqlalchemy` namespace.

7.1.1 Defining Foreign Keys

Foreign keys are most easily specified by the `ForeignKey` object within a `Column` object. For a composite foreign key, i.e. a foreign key that contains multiple columns referencing multiple columns to a composite primary key, an explicit syntax is provided which allows the correct table CREATE statements to be generated:

```
# a table with a composite primary key
invoices = Table('invoices', metadata,
    Column('invoice_id', Integer, primary_key=True),
    Column('ref_num', Integer, primary_key=True),
    Column('description', String(60), nullable=False)
)

# a table with a composite foreign key referencing the parent table
invoice_items = Table('invoice_items', metadata,
    Column('item_id', Integer, primary_key=True),
    Column('item_name', String(60), nullable=False),
    Column('invoice_id', Integer, nullable=False),
    Column('ref_num', Integer, nullable=False),
    ForeignKeyConstraint(['invoice_id', 'ref_num'], ['invoices.invoice_id', 'invoices.ref_
)
```

Above, the `invoice_items` table will have `ForeignKey` objects automatically added to the `invoice_id` and `ref_num` `Column` objects as a result of the additional `ForeignKeyConstraint` object.

7.1.2 Accessing Tables and Columns

The `MetaData` object supports some handy methods, such as getting a list of Tables in the order (or reverse) of their dependency:

```
>>> for t in metadata.table_iterator(reverse=False):
...    print t.name
users
user_prefs
```

And `Table` provides an interface to the table's properties as well as that of its columns:

```
employees = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('employee_name', String(60), nullable=False, key='name'),
    Column('employee_dept', Integer, ForeignKey("departments.department_id"))
)

# access the column "EMPLOYEE_ID":
employees.columns.employee_id

# or just
employees.c.employee_id

# via string
employees.c['employee_id']

# iterate through all columns
for c in employees.c:
    print c

# get the table's primary key columns
for primary_key in employees.primary_key:
    print primary_key

# get the table's foreign key objects:
for fkey in employees.foreign_keys:
    print fkey

# access the table's MetaData:
employees.metadata

# access the table's bound Engine or Connection, if its MetaData is bound:
employees.bind

# access a column's name, type, nullable, primary key, foreign key
employees.c.employee_id.name
employees.c.employee_id.type
employees.c.employee_id.nullable
employees.c.employee_id.primary_key
employees.c.employee_dept.foreign_key

# get the "key" of a column, which defaults to its name, but can
# be any user-defined string:
employees.c.name.key

# access a column's table:
employees.c.employee_id.table is employees

# get the table related by a foreign key
fcolumn = employees.c.employee_dept.foreign_key.column.table
```

7.1.3 Binding MetaData to an Engine or Connection

A `MetaData` object can be associated with an `Engine` or an individual `Connection`; this process is called **binding**. The term used to describe "an engine or a connection" is often referred to as a **connectable**. Binding allows the `MetaData` and the elements which it contains to perform operations against the database directly, using the connection resources to which it's bound. Common operations which are made more convenient through binding include being able to generate SQL constructs which know how to execute themselves, creating `Table` objects which query the database for their column and constraint information, and issuing CREATE or DROP statements.

To bind `MetaData` to an `Engine`, use the `bind` attribute:

```
engine = create_engine('sqlite://', **kwargs)

# create MetaData
meta = MetaData()

# bind to an engine
meta.bind = engine
```

Once this is done, the `MetaData` and its contained `Table` objects can access the database directly:

```
meta.create_all()  # issue CREATE statements for all tables

# describe a table called 'users', query the database for its columns
```

```
users_table = Table('users', meta, autoload=True)

# generate a SELECT statement and execute
result = users_table.select().execute()
```

Note that the feature of binding engines is **completely optional**. All of the operations which take advantage of "bound" `MetaData` also can be given an `Engine` or `Connection` explicitly with which to perform the operation. The equivalent "non-bound" of the above would be:

```
meta.create_all(engine)  # issue CREATE statements for all tables

# describe a table called 'users',  query the database for its columns
users_table = Table('users', meta, autoload=True, autoload_with=engine)

# generate a SELECT statement and execute
result = engine.execute(users_table.select())
```

7.1.4 Reflecting Tables

A `Table` object can be created without specifying any of its contained attributes, using the argument `autoload=True` in conjunction with the table's name and possibly its schema (if not the databases "default" schema). (You can also specify a list or set of column names to autoload as the kwarg include_columns, if you only want to load a subset of the columns in the actual database.) This will issue the appropriate queries to the database in order to locate all properties of the table required for SQLAlchemy to use it effectively, including its column names and datatypes, foreign and primary key constraints, and in some cases its default-value generating attributes. To use `autoload=True`, the table's `MetaData` object need be bound to an `Engine` or `Connection`, or alternatively the `autoload_with=<some connectable>` argument can be passed. Below we illustrate autoloading a table and then iterating through the names of its columns:

```
>>> messages = Table('messages', meta, autoload=True)
>>> [c.name for c in messages.columns]
['message_id', 'message_name', 'date']
```

Note that if a reflected table has a foreign key referencing another table, the related `Table` object will be automatically created within the `MetaData` object if it does not exist already. Below, suppose table `shopping_cart_items` references a table `shopping_carts`. After reflecting, the `shopping carts` table is present:

```
>>> shopping_cart_items = Table('shopping_cart_items', meta, autoload=True)
>>> 'shopping_carts' in meta.tables:
True
```

To get direct access to 'shopping_carts', simply instantiate it via the `Table` constructor. `Table` uses a special constructor that will return the already created `Table` instance if it's already present:

```
shopping_carts = Table('shopping_carts', meta)
```

Of course, it's a good idea to use `autoload=True` with the above table regardless. This is so that if it hadn't been loaded already, the operation will load the table. The autoload operation only occurs for the table if it hasn't already been loaded; once loaded, new calls to `Table` will not re-issue any reflection queries.

Overriding Reflected Columns

Individual columns can be overridden with explicit values when reflecting tables; this is handy for specifying custom datatypes, constraints such as primary keys that may not be configured within the database, etc.:

```
>>> mytable = Table('mytable', meta,
... Column('id', Integer, primary_key=True),   # override reflected 'id' to have primary ke
... Column('mydata', Unicode(50)),    # override reflected 'mydata' to be Unicode
... autoload=True)
```

Reflecting All Tables at Once

The `MetaData` object can also get a listing of tables and reflect the full set. This is achieved by using the `reflect()` method. After calling it, all located tables are present within the `MetaData` object's dictionary of tables:

```
meta = MetaData()
meta.reflect(bind=someengine)
users_table = meta.tables['users']
addresses_table = meta.tables['addresses']
```

`metadata.reflect()` is also a handy way to clear or drop all tables in a database:

```
meta = MetaData()
meta.reflect(bind=someengine)
for table in reversed(meta.sorted_tables):
    someengine.execute(table.delete())
```

7.1.5 Specifying the Schema Name

Some databases support the concept of multiple schemas. A `Table` can reference this by specifying the `schema` keyword argument:

```
financial_info = Table('financial_info', meta,
    Column('id', Integer, primary_key=True),
    Column('value', String(100), nullable=False),
    schema='remote_banks'
)
```

Within the `MetaData` collection, this table will be identified by the combination of `financial_info` and `remote_banks`. If another table called `financial_info` is referenced without the `remote_banks` schema, it will refer to a different `Table`. `ForeignKey` objects can reference columns in this table using the form `remote_banks.financial_info.id`.

7.1.6 ON UPDATE and ON DELETE

`ON UPDATE` and `ON DELETE` clauses to a table create are specified within the `ForeignKeyConstraint` object, using the `onupdate` and `ondelete` keyword arguments:

```
foobar = Table('foobar', meta,
    Column('id', Integer, primary_key=True),
    Column('lala', String(40)),
    ForeignKeyConstraint(['lala'],['hoho.lala'], onupdate="CASCADE", ondelete="CASCADE"))
```

Note that these clauses are not supported on SQLite, and require InnoDB tables when used with MySQL. They may also not be supported on other databases.

7.1.7 Other Options

Tables may support database-specific options, such as MySQL's engine option that can specify "MyISAM", "InnoDB", and other backends for the table:

```
addresses = Table('engine_email_addresses', meta,
    Column('address_id', Integer, primary_key = True),
    Column('remote_user_id', Integer, ForeignKey(users.c.user_id)),
    Column('email_address', String(20)),
    mysql_engine='InnoDB'
)
```

7.2 Creating and Dropping Database Tables

Creating and dropping individual tables can be done via the create() and drop() methods of Table; these methods take an optional bind parameter which references an Engine or a Connection. If not supplied, the Engine bound to the MetaData will be used, else an error is raised:

```
meta = MetaData()
meta.bind = 'sqlite:///:memory:'

employees = Table('employees', meta,
    Column('employee_id', Integer, primary_key=True),
    Column('employee_name', String(60), nullable=False, key='name'),
    Column('employee_dept', Integer, ForeignKey("departments.department_id"))
)
employees.create()
CREATE TABLE employees(
employee_id SERIAL NOT NULL PRIMARY KEY,
employee_name VARCHAR(60) NOT NULL,
employee_dept INTEGER REFERENCES departments(department_id)
)
```

drop() method:

```
employees.drop(bind=e)
DROP TABLE employee
```

The create() and drop() methods also support an optional keyword argument checkfirst which will issue the database's appropriate pragma statements to check if the table exists before creating or dropping:

```
employees.create(bind=e, checkfirst=True)
employees.drop(checkfirst=False)
```

Entire groups of Tables can be created and dropped directly from the MetaData object with create_all() and drop_all(). These methods always check for the existence of each table before creating or dropping. Each method takes an optional bind keyword argument which can reference an Engine or a Connection. If no engine is specified, the underlying bound Engine, if any, is used:

```
engine = create_engine('sqlite:///:memory:')

metadata = MetaData()

users = Table('users', metadata,
    Column('user_id', Integer, primary_key = True),
    Column('user_name', String(16), nullable = False),
    Column('email_address', String(60), key='email'),
    Column('password', String(20), nullable = False)
)

user_prefs = Table('user_prefs', metadata,
    Column('pref_id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey("users.user_id"), nullable=False),
    Column('pref_name', String(40), nullable=False),
    Column('pref_value', String(100))
)

metadata.create_all(bind=engine)
PRAGMA table_info(users){}
CREATE TABLE users(
        user_id INTEGER NOT NULL PRIMARY KEY,
        user_name VARCHAR(16) NOT NULL,
        email_address VARCHAR(60),
        password VARCHAR(20) NOT NULL
)
PRAGMA table_info(user_prefs){}
CREATE TABLE user_prefs(
        pref_id INTEGER NOT NULL PRIMARY KEY,
        user_id INTEGER NOT NULL REFERENCES users(user_id),
        pref_name VARCHAR(40) NOT NULL,
        pref_value VARCHAR(100)
)
```

7.3 Column Insert/Update Defaults

SQLAlchemy includes several constructs which provide default values provided during INSERT and UPDATE statements. The defaults may be provided as Python constants, Python functions, or SQL expressions, and the SQL expressions themselves may be "pre-executed", executed inline within the insert/update statement itself, or can be created as a SQL level "default" placed on the table definition itself. A "default" value by definition is only invoked if no explicit value is passed into the INSERT or UPDATE statement.

7.3.1 Pre-Executed Python Functions

The "default" keyword argument on Column can reference a Python value or callable which is invoked at the time of an insert:

```
# a function which counts upwards
i = 0
def mydefault():
    global i
    i += 1
    return i

t = Table("mytable", meta,
    # function-based default
    Column('id', Integer, primary_key=True, default=mydefault),

    # a scalar default
    Column('key', String(10), default="default")
)
```

Similarly, the "onupdate" keyword does the same thing for update statements:

```
import datetime

t = Table("mytable", meta,
    Column('id', Integer, primary_key=True),

    # define 'last_updated' to be populated with datetime.now()
    Column('last_updated', DateTime, onupdate=datetime.datetime.now),
)
```

7.3.2 Pre-executed and Inline SQL Expressions

The "default" and "onupdate" keywords may also be passed SQL expressions, including select statements or direct function calls:

```
t = Table("mytable", meta,
    Column('id', Integer, primary_key=True),

    # define 'create_date' to default to now()
    Column('create_date', DateTime, default=func.now()),

    # define 'key' to pull its default from the 'keyvalues' table
    Column('key', String(20), default=keyvalues.select(keyvalues.c.type='type1', limit=1))

    # define 'last_modified' to use the current_timestamp SQL function on update
    Column('last_modified', DateTime, onupdate=func.current_timestamp())
    )
```

The above SQL functions are usually executed "inline" with the INSERT or UPDATE statement being executed. In some cases, the function is "pre-executed" and its result pre-fetched explicitly. This happens under the following circumstances:

- the column is a primary key column
- the database dialect does not support a usable `cursor.lastrowid` accessor (or equivalent); this currently includes Postgres, Oracle, and Firebird.
- the statement is a single execution, i.e. only supplies one set of parameters and doesn't use "executemany" behavior
- the `inline=True` flag is not set on the `Insert()` or `Update()` construct.

For a statement execution which is not an executemany, the returned `ResultProxy` will contain a collection accessible via `result.postfetch_cols()` which contains a list of all `Column` objects which had an inline-executed default. Similarly, all parameters which were bound to the statement, including all Python and SQL expressions which were pre-executed, are present in the `last_inserted_params()` or `last_updated_params()` collections on `ResultProxy`. The `last_inserted_ids()` collection contains a list of primary key values for the row inserted.

7.3.3 DDL-Level Defaults

A variant on a SQL expression default is the `server_default`, which gets placed in the CREATE TABLE statement during a `create()` operation:

```
t = Table('test', meta,
    Column('abc', String(20), server_default='abc'),
    Column('created_at', DateTime, server_default=text("sysdate"))
)
```

A create call for the above table will produce:

```
CREATE TABLE test (
    abc varchar(20) default 'abc',
    created_at datetime default sysdate
)
```

The behavior of `server_default` is similar to that of a regular SQL default; if it's placed on a primary key column for a database which doesn't have a way to "postfetch" the ID, and the statement is not "inlined", the SQL expression is pre-executed; otherwise, SQLAlchemy lets the default fire off on the database side normally.

7.3.4 Triggered Columns

Columns with values set by a database trigger or other external process may be called out with a marker:

```
t = Table('test', meta,
    Column('abc', String(20), server_default=FetchedValue())
    Column('def', String(20), server_onupdate=FetchedValue())
)
```

These markers do not emit a ``` ``default`` ``` clause when the table is created, however they do set the same internal flags as a static `server_default` clause, providing hints to higher-level tools that a "post-fetch" of these rows should be performed after an insert or update.

7.3.5 Defining Sequences

A table with a sequence looks like:

```
table = Table("cartitems", meta,
    Column("cart_id", Integer, Sequence('cart_id_seq'), primary_key=True),
    Column("description", String(40)),
    Column("createdate", DateTime())
)
```

The `Sequence` object works a lot like the `default` keyword on `Column`, except that it only takes effect on a database which supports sequences. When used with a database that does not support sequences, the `Sequence` object has no effect; therefore it's safe to place on a table which is used against multiple database backends. The same rules for pre- and inline execution apply.

When the `Sequence` is associated with a table, CREATE and DROP statements issued for that table will also issue CREATE/DROP for the sequence object as well, thus "bundling" the sequence object with its parent table.

The flag `optional=True` on `Sequence` will produce a sequence that is only used on databases which have no "autoincrementing" capability. For example, Postgres supports primary key generation using the SERIAL keyword, whereas Oracle has no such capability. Therefore, a `Sequence` placed on a primary key column with `optional=True` will only be used with an Oracle backend but not Postgres.

A sequence can also be executed standalone, using an `Engine` or `Connection`, returning its next value in a database-independent fashion:

```
seq = Sequence('some_sequence')
nextid = connection.execute(seq)
```

7.4 Defining Constraints and Indexes

7.4.1 UNIQUE Constraint

Unique constraints can be created anonymously on a single column using the `unique` keyword on `Column`. Explicitly named unique constraints and/or those with multiple columns are created via the `UniqueConstraint` table-level construct.

```
meta = MetaData()
mytable = Table('mytable', meta,

    # per-column anonymous unique constraint
    Column('col1', Integer, unique=True),

    Column('col2', Integer),
    Column('col3', Integer),

    # explicit/composite unique constraint.  'name' is optional.
    UniqueConstraint('col2', 'col3', name='uix_1')
    )
```

7.4.2 CHECK Constraint

Check constraints can be named or unnamed and can be created at the Column or Table level, using the `CheckConstraint` construct. The text of the check constraint is passed directly through to the database, so there is limited "database independent" behavior. Column level check constraints generally should only refer to the column to which they are placed, while table level constraints can refer to any columns in the table.

Note that some databases do not actively support check constraints such as MySQL and SQLite.

```
meta = MetaData()
mytable = Table('mytable', meta,

    # per-column CHECK constraint
    Column('col1', Integer, CheckConstraint('col1>5')),

    Column('col2', Integer),
    Column('col3', Integer),

    # table level CHECK constraint.  'name' is optional.
    CheckConstraint('col2 > col3 + 5', name='check1')
    )
```

7.4.3 Indexes

Indexes can be created anonymously (using an auto-generated name "ix_&lt;column label&gt;") for a single column using the inline `index` keyword on `Column`, which also modifies the usage of `unique` to apply the uniqueness to the index itself, instead of adding a separate UNIQUE constraint. For indexes with specific names or which encompass more than one column, use the `Index` construct, which requires a name.

Note that the `Index` construct is created **externally** to the table which it corresponds, using `Column` objects and not strings.

```
meta = MetaData()
mytable = Table('mytable', meta,
    # an indexed column, with index "ix_mytable_col1"
    Column('col1', Integer, index=True),

    # a uniquely indexed column with index "ix_mytable_col2"
    Column('col2', Integer, index=True, unique=True),

    Column('col3', Integer),
    Column('col4', Integer),

    Column('col5', Integer),
    Column('col6', Integer),
    )

# place an index on col3, col4
Index('idx_col34', mytable.c.col3, mytable.c.col4)

# place a unique index on col5, col6
Index('myindex', mytable.c.col5, mytable.c.col6, unique=True)
```

The `Index` objects will be created along with the CREATE statements for the table itself. An index can also be created on its own independently of the table:

```
# create a table
sometable.create()

# define an index
i = Index('someindex', sometable.c.col5)

# create the index, will use the table's bound connectable if the ``bind`` keyword argumen
i.create()
```

7.5 Adapting Tables to Alternate Metadata

A `Table` object created against a specific `MetaData` object can be re-created against a new MetaData using the `tometadata` method:

```
# create two metadata
meta1 = MetaData('sqlite:///querytest.db')
meta2 = MetaData()

# load 'users' from the sqlite engine
users_table = Table('users', meta1, autoload=True)

# create the same Table object for the plain metadata
users_table_2 = users_table.tometadata(meta2)
```

CHAPTER

EIGHT

API REFERENCE

8.1 sqlalchemy

8.1.1 Connections

Creating Engines

`create_engine``(`**args, **kwargs*`)`

Create a new Engine instance.

The standard method of specifying the engine is via URL as the first positional argument, to indicate the appropriate database dialect and connection arguments, with additional keyword arguments sent as options to the dialect and resulting Engine.

The URL is a string in the form `dialect://user:password@host/dbname[?key=value..]`, where `dialect` is a name such as `mysql`, `oracle`, `postgres`, etc. Alternatively, the URL can be an instance of `URL`.

***kwargs* takes a wide variety of options which are routed towards their appropriate components. Arguments may be specific to the Engine, the underlying Dialect, as well as the Pool. Specific dialects also accept keyword arguments that are unique to that dialect. Here, we describe the parameters that are common to most `create_engine()` usage.

Parameters

- *assert_unicode=False* – When set to `True` alongside convert_unicode=‘‘True‘‘, asserts that incoming string bind parameters are instances of `unicode`, otherwise raises an error. Only takes effect when `convert_unicode==True`. This flag is also available on the `String` type and its descendants. New in 0.4.2.
- *connect_args* – a dictionary of options which will be passed directly to the DBAPI's `connect()` method as additional keyword arguments.
- *convert_unicode=False* – if set to True, all String/character based types will convert Unicode values to raw byte values going into the database, and all raw byte values to Python Unicode coming out in result sets. This is an engine-wide method to provide unicode conversion across the board. For unicode conversion on a column-by-column level, use the `Unicode` column type instead, described in *types*.
- *creator* – a callable which returns a DBAPI connection. This creation function will be passed to the underlying connection pool and will be used to create all new database connections. Usage of this function causes connection parameters specified in the URL argument to be bypassed.
- *echo=False* – if True, the Engine will log all statements as well as a repr() of their parameter lists to the engines logger, which defaults to sys.stdout. The `echo` attribute of `Engine` can be modified at any time to turn logging on and off. If set to the string `"debug"`, result rows will be printed to the standard output as well. This flag ultimately controls a Python logger;

see *dbengine_logging* at the end of this chapter for information on how to configure logging directly.

- *echo_pool=False* – if True, the connection pool will log all checkouts/checkins to the logging stream, which defaults to sys.stdout. This flag ultimately controls a Python logger; see *dbengine_logging* for information on how to configure logging directly.
- *encoding='utf-8'* – the encoding to use for all Unicode translations, both by engine-wide unicode conversion as well as the `Unicode` type object.
- *label_length=None* – optional integer value which limits the size of dynamically generated column labels to that many characters. If less than 6, labels are generated as "_(counter)". If `None`, the value of `dialect.max_identifier_length` is used instead.
- *module=None* – used by database implementations which support multiple DBAPI modules, this is a reference to a DBAPI2 module to be used instead of the engine's default module. For Postgres, the default is psycopg2. For Oracle, it's cx_Oracle.
- *pool=None* – an already-constructed instance of `Pool`, such as a `QueuePool` instance. If non-None, this pool will be used directly as the underlying connection pool for the engine, bypassing whatever connection parameters are present in the URL argument. For information on constructing connection pools manually, see *pooling*.
- *poolclass=None* – a `Pool` subclass, which will be used to create a connection pool instance using the connection parameters given in the URL. Note this differs from `pool` in that you don't actually instantiate the pool in this case, you just indicate what type of pool to be used.
- *max_overflow=10* – the number of connections to allow in connection pool "overflow", that is connections that can be opened above and beyond the pool_size setting, which defaults to five. this is only used with `QueuePool`.
- *pool_size=5* – the number of connections to keep open inside the connection pool. This used with `QueuePool` as well as `SingletonThreadPool`.
- *pool_recycle=-1* – this setting causes the pool to recycle connections after the given number of seconds has passed. It defaults to -1, or no timeout. For example, setting to 3600 means connections will be recycled after one hour. Note that MySQL in particular will `disconnect automatically` if no activity is detected on a connection for eight hours (although this is configurable with the MySQLDB connection itself and the server configuration as well).
- *pool_timeout=30* – number of seconds to wait before giving up on getting a connection from the pool. This is only used with `QueuePool`.
- *strategy='plain'* – used to invoke alternate implementations. Currently available is the `threadlocal` strategy, which is described in *Using the Threadlocal Execution Strategy*.

`engine_from_config`(*configuration, prefix='sqlalchemy.', **kwargs*)

Create a new Engine instance using a configuration dictionary.

The dictionary is typically produced from a config file where keys are prefixed, such as sqlalchemy.url, sqlalchemy.echo, etc. The 'prefix' argument indicates the prefix to be searched for.

A select set of keyword arguments will be "coerced" to their expected type based on string values. In a future release, this functionality will be expanded and include dialect-specific arguments.

class `URL`(*drivername, username=None, password=None, host=None, port=None, database=None, query=None*)

Represent the components of a URL used to connect to a database.

This object is suitable to be passed directly to a `create_engine()` call. The fields of the URL are parsed from a string by the `module-level make_url()` function. the string format of the URL is an RFC-1738-style string.

All initialization parameters are available as public attributes.

Parameters
- *drivername* – the name of the database backend. This name will correspond to a module in sqlalchemy/databases or a third party plug-in.

- *username* – The user name.
- *password* – database password.
- *host* – The name of the host.
- *port* – The port number.
- *database* – The database name.
- *query* – A dictionary of options to be passed to the dialect and/or the DBAPI upon connect.

`__init__`(*drivername, username=None, password=None, host=None, port=None, database=None, query=None*)

`get_dialect`()
Return the SQLAlchemy database dialect class corresponding to this URL's driver name.

`translate_connect_args`(*names=, [], **kw*)
Translate url attributes into a dictionary of connection arguments.

Returns attributes of this url (*host*, *database*, *username*, *password*, *port*) as a plain dictionary. The attribute names are used as the keys by default. Unset or false attributes are omitted from the final dictionary.

Parameters
- ***kw* – Optional, alternate key names for url attributes.
- *names* – Deprecated. Same purpose as the keyword-based alternate names, but correlates the name to the original positionally.

Connectables

class `Engine`(*pool, dialect, url, echo=None, proxy=None*)
Connects a `Pool` and `Dialect` together to provide a source of database connectivity and behavior.

`__init__`(*pool, dialect, url, echo=None, proxy=None*)

`connect`(***kwargs*)
Return a newly allocated Connection object.

`contextual_connect`(*close_with_result=False, **kwargs*)
Return a Connection object which may be newly allocated, or may be part of some ongoing context.

This Connection is meant to be used by the various "auto-connecting" operations.

`create`(*entity, connection=None, **kwargs*)
Create a table or index within this engine's database connection given a schema.Table object.

`drop`(*entity, connection=None, **kwargs*)
Drop a table or index within this engine's database connection given a schema.Table object.

`echo`
When `True`, enable log output for this element.

This has the effect of setting the Python logging level for the namespace of this element's class and object reference. A value of boolean `True` indicates that the loglevel `logging.INFO` will be set for the logger, whereas the string value `debug` will set the loglevel to `logging.DEBUG`.

`name`
String name of the `Dialect` in use by this `Engine`.

`raw_connection`()
Return a DB-API connection.

`reflecttable`(*table, connection=None, include_columns=None*)
Given a Table object, reflects its columns and properties from the database.

`table_names`(*schema=None, connection=None*)
Return a list of all table names available in the database.

schema: Optional, retrieve names from a non-default schema.

connection: Optional, use a specified connection. Default is the `contextual_connect` for this `Engine`.

`text`(*text, *args, **kwargs*)
Return a sql.text() object for performing literal queries.

`transaction`(*callable_, connection=None, *args, **kwargs*)
Execute the given function within a transaction boundary.

This is a shortcut for explicitly calling *begin()* and *commit()* and optionally *rollback()* when exceptions are raised. The given **args* and ***kwargs* will be passed to the function, as well as the Connection used in the transaction.

class `Connection`(*engine, connection=None, close_with_result=False, _branch=False*)
Provides high-level functionality for a wrapped DB-API connection.

Provides execution support for string-based SQL statements as well as ClauseElement, Compiled and Default-Generator objects. Provides a begin method to return Transaction objects.

The Connection object is **not** thread-safe.

`__init__`(*engine, connection=None, close_with_result=False, _branch=False*)
Construct a new Connection.

Connection objects are typically constructed by an `Engine`, see the `connect()` and `contextual_connect()` methods of Engine.

`begin`()
Begin a transaction and return a Transaction handle.

Repeated calls to `begin` on the same Connection will create a lightweight, emulated nested transaction. Only the outermost transaction may `commit`. Calls to `commit` on inner transactions are ignored. Any transaction in the hierarchy may `rollback`, however.

`begin_nested`()
Begin a nested transaction and return a Transaction handle.

Nested transactions require SAVEPOINT support in the underlying database. Any transaction in the hierarchy may `commit` and `rollback`, however the outermost transaction still controls the overall `commit` or `rollback` of the transaction of a whole.

`begin_twophase`(*xid=None*)
Begin a two-phase or XA transaction and return a Transaction handle.

xid the two phase transaction id. If not supplied, a random id will be generated.

`close`()
Close this Connection.

`closed`
return True if this connection is closed.

`connect`()
Returns self.

This `Connectable` interface method returns self, allowing Connections to be used interchangably with Engines in most situations that require a bind.

`connection`
The underlying DB-API connection managed by this Connection.

`contextual_connect`(***kwargs*)
Returns self.

This `Connectable` interface method returns self, allowing Connections to be used interchangably with Engines in most situations that require a bind.

`create`(*entity, **kwargs*)
Create a Table or Index given an appropriate Schema object.

`detach`()
Detach the underlying DB-API connection from its connection pool.

This Connection instance will remain useable. When closed, the DB-API connection will be literally closed and not returned to its pool. The pool will typically lazily create a new connection to replace the detached connection.

This method can be used to insulate the rest of an application from a modified state on a connection (such as a transaction isolation level or similar). Also see `PoolListener` for a mechanism to modify connection state when connections leave and return to their connection pool.

`dialect`
Dialect used by this Connection.

`drop`(*entity, **kwargs*)
Drop a Table or Index given an appropriate Schema object.

`execute`(*object, *multiparams, **params*)
Executes and returns a ResultProxy.

`in_transaction`()
Return True if a transaction is in progress.

`info`
A collection of per-DB-API connection instance properties.

`invalidate`(*exception=None*)
Invalidate the underlying DBAPI connection associated with this Connection.

The underlying DB-API connection is literally closed (if possible), and is discarded. Its source connection pool will typically lazily create a new connection to replace it.

Upon the next usage, this Connection will attempt to reconnect to the pool with a new connection.

Transactions in progress remain in an "opened" state (even though the actual transaction is gone); these must be explicitly rolled back before a reconnect on this Connection can proceed. This is to prevent applications from accidentally continuing their transactional operations in a non-transactional state.

`invalidated`
return True if this connection was invalidated.

`reflecttable`(*table, include_columns=None*)
Reflect the columns in the given string table name from the database.

`scalar`(*object, *multiparams, **params*)
Executes and returns the first column of the first row.

The underlying result/cursor is closed after execution.

`should_close_with_result`
Indicates if this Connection should be closed when a corresponding ResultProxy is closed; this is essentially an auto-release mode.

class `Connectable`()
Interface for an object which supports execution of SQL constructs.

The two implementations of `Connectable` are `Connection` and `Engine`.

`contextual_connect`()
Return a Connection object which may be part of an ongoing context.

`create`(*entity, **kwargs*)
Create a table or index given an appropriate schema object.

`drop`(*entity, **kwargs*)
Drop a table or index given an appropriate schema object.

`execute`(*object, *multiparams, **params*)

Result Objects

class **`ResultProxy`**(*context*)

Wraps a DB-API cursor object to provide easier access to row columns.

Individual columns may be accessed by their integer position, case-insensitive column name, or by `schema.Column` object. e.g.:

```
row = fetchone()

col1 = row[0]    # access via integer position

col2 = row['col2']   # access via name

col3 = row[mytable.c.mycol] # access via Column object.
```

ResultProxy also contains a map of TypeEngine objects and will invoke the appropriate `result_processor()` method before returning columns, as well as the ExecutionContext corresponding to the statement execution. It provides several methods for which to obtain information from the underlying ExecutionContext.

`__init__`(*context*)

ResultProxy objects are constructed via the execute() method on SQLEngine.

`close`()

Close this ResultProxy.

Closes the underlying DBAPI cursor corresponding to the execution.

If this ResultProxy was generated from an implicit execution, the underlying Connection will also be closed (returns the underlying DBAPI connection to the connection pool.)

This method is called automatically when:

- all result rows are exhausted using the fetchXXX() methods.
- cursor.description is None.

`fetchall`()

Fetch all rows, just like DB-API `cursor.fetchall()`.

`fetchmany`(*size=None*)

Fetch many rows, just like DB-API `cursor.fetchmany(size=cursor.arraysize)`.

`fetchone`()

Fetch one row, just like DB-API `cursor.fetchone()`.

`last_inserted_ids`()

Return `last_inserted_ids()` from the underlying ExecutionContext.

See ExecutionContext for details.

`last_inserted_params`()

Return `last_inserted_params()` from the underlying ExecutionContext.

See ExecutionContext for details.

`last_updated_params`()

Return `last_updated_params()` from the underlying ExecutionContext.

See ExecutionContext for details.

`lastrow_has_defaults`()

Return `lastrow_has_defaults()` from the underlying ExecutionContext.

See ExecutionContext for details.

`postfetch_cols`()

Return `postfetch_cols()` from the underlying ExecutionContext.

See ExecutionContext for details.

scalar()
Fetch the first column of the first row, and close the result set.

supports_sane_multi_rowcount()
Return `supports_sane_multi_rowcount` from the dialect.

supports_sane_rowcount()
Return `supports_sane_rowcount` from the dialect.

class RowProxy(*parent, row*)
Proxy a single cursor row for a parent ResultProxy.

Mostly follows "ordered dictionary" behavior, mapping result values to the string-based column name, the integer position of the result in the row, as well as Column instances which can be mapped to the original Columns that produced this result set (for results that correspond to constructed SQL expressions).

__init__(*parent, row*)
RowProxy objects are constructed by ResultProxy objects.

close()
Close the parent ResultProxy.

has_key(*key*)
Return True if this RowProxy contains the given key.

items()
Return a list of tuples, each tuple containing a key/value pair.

keys()
Return the list of keys as strings represented by this RowProxy.

values()
Return the values represented by this RowProxy as a list.

Transactions

class Transaction(*connection, parent*)
Represent a Transaction in progress.

The Transaction object is **not** threadsafe.

__init__(*connection, parent*)

close()
Close this transaction.

If this transaction is the base transaction in a begin/commit nesting, the transaction will rollback(). Otherwise, the method returns.

This is used to cancel a Transaction without affecting the scope of an enclosing transaction.

commit()

rollback()

Internals

connection_memoize(*key*)
Decorator, memoize a function in a connection.info stash.

Only applicable to functions which take no arguments other than a connection. The memo will be stored in `connection.info[key]`.

class Dialect()
Define the behavior of a specific database and DB-API combination.

Any aspect of metadata definition, SQL query generation, execution, result-set handling, or anything else which varies between databases is defined under the general category of the Dialect. The Dialect acts as a factory for other database-specific object implementations including ExecutionContext, Compiled, DefaultGenerator, and TypeEngine.

All Dialects implement the following attributes:

name identifying name for the dialect (i.e. 'sqlite')

positional True if the paramstyle for this Dialect is positional.

paramstyle the paramstyle to be used (some DB-APIs support multiple paramstyles).

convert_unicode True if Unicode conversion should be applied to all `str` types.

encoding type of encoding to use for unicode, usually defaults to 'utf-8'.

schemagenerator a `SchemaVisitor` class which generates schemas.

schemadropper a `SchemaVisitor` class which drops schemas.

defaultrunner a `SchemaVisitor` class which executes defaults.

statement_compiler a `Compiled` class used to compile SQL statements

preparer a `IdentifierPreparer` class used to quote identifiers.

supports_alter `True` if the database supports `ALTER TABLE`.

max_identifier_length The maximum length of identifier names.

supports_unicode_statements Indicate whether the DB-API can receive SQL statements as Python unicode strings

supports_sane_rowcount Indicate whether the dialect properly implements rowcount for `UPDATE` and `DELETE` statements.

supports_sane_multi_rowcount Indicate whether the dialect properly implements rowcount for `UPDATE` and `DELETE` statements when executed via executemany.

preexecute_pk_sequences Indicate if the dialect should pre-execute sequences on primary key columns during an INSERT, if it's desired that the new row's primary key be available after execution.

supports_pk_autoincrement Indicates if the dialect should allow the database to passively assign a primary key column value.

dbapi_type_map A mapping of DB-API type objects present in this Dialect's DB-API implmentation mapped to TypeEngine implementations used by the dialect.

This is used to apply types to result sets based on the DB-API types present in cursor.description; it only takes effect for result sets against textual statements where no explicit typemap was present.

supports_default_values Indicates if the construct `INSERT INTO tablename DEFAULT VALUES` is supported

description_encoding type of encoding to use for unicode when working with metadata descriptions. If set to `None` no encoding will be done. This usually defaults to 'utf-8'.

`create_connect_args`(*url*)

Build DB-API compatible connection arguments.

Given a `URL` object, returns a tuple consisting of a **args/**kwargs* suitable to send directly to the dbapi's connect function.

`create_xid`()

Create a two-phase transaction ID.

This id will be passed to do_begin_twophase(), do_rollback_twophase(), do_commit_twophase(). Its format is unspecified.

`do_begin`(*connection*)

Provide an implementation of *connection.begin()*, given a DB-API connection.

do_begin_twophase(*connection, xid*)
Begin a two phase transaction on the given connection.

do_commit(*connection*)
Provide an implementation of *connection.commit()*, given a DB-API connection.

do_commit_twophase(*connection, xid, is_prepared=True, recover=False*)
Commit a two phase transaction on the given connection.

do_execute(*cursor, statement, parameters, context=None*)
Provide an implementation of *cursor.execute(statement, parameters)*.

do_executemany(*cursor, statement, parameters, context=None*)
Provide an implementation of *cursor.executemany(statement, parameters)*.

do_prepare_twophase(*connection, xid*)
Prepare a two phase transaction on the given connection.

do_recover_twophase(*connection*)
Recover list of uncommited prepared two phase transaction identifiers on the given connection.

do_release_savepoint(*connection, name*)
Release the named savepoint on a SQL Alchemy connection.

do_rollback(*connection*)
Provide an implementation of *connection.rollback()*, given a DB-API connection.

do_rollback_to_savepoint(*connection, name*)
Rollback a SQL Alchemy connection to the named savepoint.

do_rollback_twophase(*connection, xid, is_prepared=True, recover=False*)
Rollback a two phase transaction on the given connection.

do_savepoint(*connection, name*)
Create a savepoint with the given name on a SQLAlchemy connection.

get_default_schema_name(*connection*)
Return the string name of the currently selected schema given a `Connection`.

has_sequence(*connection, sequence_name, schema=None*)
Check the existence of a particular sequence in the database.

Given a `Connection` object and a string *sequence_name*, return True if the given sequence exists in the database, False otherwise.

has_table(*connection, table_name, schema=None*)
Check the existence of a particular table in the database.

Given a `Connection` object and a string *table_name*, return True if the given table (possibly within the specified *schema*) exists in the database, False otherwise.

is_disconnect(*e*)
Return True if the given DB-API error indicates an invalid connection

reflecttable(*connection, table, include_columns=None*)
Load table description from the database.

Given a `Connection` and a `Table` object, reflect its columns and properties from the database. If include_columns (a list or set) is specified, limit the autoload to the given column names.

server_version_info(*connection*)
Return a tuple of the database's version number.

type_descriptor(*typeobj*)
Transform a generic type to a database-specific type.

Transforms the given `TypeEngine` instance from generic to database-specific.

Subclasses will usually use the `adapt_type()` method in the types module to make this job easy.

class DefaultDialect(*convert_unicode=False, assert_unicode=False, encoding='utf-8', paramstyle=None, dbapi=None, label_length=None, **kwargs*)

Bases: `sqlalchemy.engine.base.Dialect`

Default implementation of Dialect

`__init__`(*convert_unicode=False, assert_unicode=False, encoding='utf-8', paramstyle=None, dbapi=None, label_length=None, **kwargs*)

`create_xid`()

Create a random two-phase transaction ID.

This id will be passed to do_begin_twophase(), do_rollback_twophase(), do_commit_twophase(). Its format is unspecified.

`defaultrunner`

alias of `DefaultRunner`

`do_begin`(*connection*)

Implementations might want to put logic here for turning autocommit on/off, etc.

`do_commit`(*connection*)

Implementations might want to put logic here for turning autocommit on/off, etc.

`do_rollback`(*connection*)

Implementations might want to put logic here for turning autocommit on/off, etc.

`preparer`

alias of `IdentifierPreparer`

`statement_compiler`

alias of `DefaultCompiler`

`type_descriptor`(*typeobj*)

Provide a database-specific `TypeEngine` object, given the generic object which comes from the types module.

Subclasses will usually use the `adapt_type()` method in the types module to make this job easy.

class DefaultExecutionContext(*dialect, connection, compiled=None, statement=None, parameters=None*)

Bases: `sqlalchemy.engine.base.ExecutionContext`

`__init__`(*dialect, connection, compiled=None, statement=None, parameters=None*)

`set_input_sizes`()

Given a cursor and ClauseParameters, call the appropriate style of `setinputsizes()` on the cursor, using DB-API types from the bind parameter's `TypeEngine` objects.

class DefaultRunner(*context*)

Bases: `sqlalchemy.schema.SchemaVisitor`

A visitor which accepts ColumnDefault objects, produces the dialect-specific SQL corresponding to their execution, and executes the SQL, returning the result value.

DefaultRunners are used internally by Engines and Dialects. Specific database modules should provide their own subclasses of DefaultRunner to allow database-specific behavior.

`__init__`(*context*)

`execute_string`(*stmt, params=None*)

execute a string statement, using the raw cursor, and return a scalar result.

class ExecutionContext()

A messenger object for a Dialect that corresponds to a single execution.

ExecutionContext should have these datamembers:

connection Connection object which can be freely used by default value generators to execute SQL. This Connection should reference the same underlying connection/transactional resources of root_connection.

root_connection Connection object which is the source of this ExecutionContext. This Connection may have close_with_result=True set, in which case it can only be used once.

dialect dialect which created this ExecutionContext.

cursor DB-API cursor procured from the connection,

compiled if passed to constructor, sqlalchemy.engine.base.Compiled object being executed,

statement string version of the statement to be executed. Is either passed to the constructor, or must be created from the sql.Compiled object by the time pre_exec() has completed.

parameters bind parameters passed to the execute() method. For compiled statements, this is a dictionary or list of dictionaries. For textual statements, it should be in a format suitable for the dialect's paramstyle (i.e. dict or list of dicts for non positional, list or list of lists/tuples for positional).

isinsert True if the statement is an INSERT.

isupdate True if the statement is an UPDATE.

should_autocommit True if the statement is a "committable" statement

postfetch_cols a list of Column objects for which a server-side default or inline SQL expression value was fired off. applies to inserts and updates.

`create_cursor`()

Return a new cursor generated from this ExecutionContext's connection.

Some dialects may wish to change the behavior of connection.cursor(), such as postgres which may return a PG "server side" cursor.

`handle_dbapi_exception`(*e*)

Receive a DBAPI exception which occured upon execute, result fetch, etc.

`last_inserted_ids`()

Return the list of the primary key values for the last insert statement executed.

This does not apply to straight textual clauses; only to `sql.Insert` objects compiled against a `schema.Table` object. The order of items in the list is the same as that of the Table's 'primary_key' attribute.

`last_inserted_params`()

Return a dictionary of the full parameter dictionary for the last compiled INSERT statement.

Includes any ColumnDefaults or Sequences that were pre-executed.

`last_updated_params`()

Return a dictionary of the full parameter dictionary for the last compiled UPDATE statement.

Includes any ColumnDefaults that were pre-executed.

`lastrow_has_defaults`()

Return True if the last INSERT or UPDATE row contained inlined or database-side defaults.

`post_exec`()

Called after the execution of a compiled statement.

If a compiled statement was passed to this ExecutionContext, the *last_insert_ids*, *last_inserted_params*, etc. datamembers should be available after this method completes.

`pre_exec`()

Called before an execution of a compiled statement.

If a compiled statement was passed to this ExecutionContext, the *statement* and *parameters* datamembers must be initialized after this statement is complete.

`result`()

Return a result object corresponding to this ExecutionContext.

Returns a ResultProxy.

`should_autocommit_text`(*statement*)

Parse the given textual statement and return True if it refers to a "committable" statement

class SchemaIterator(*connection*)
Bases: `sqlalchemy.schema.SchemaVisitor`

A visitor that can gather text into a buffer and execute the contents of the buffer.

__init__(*connection*)
Construct a new SchemaIterator.

append(*s*)
Append content to the SchemaIterator's query buffer.

execute()
Execute the contents of the SchemaIterator's buffer.

8.1.2 Connection Pooling

SQLAlchemy ships with a connection pooling framework that integrates with the Engine system and can also be used on its own to manage plain DB-API connections.

At the base of any database helper library is a system for efficiently acquiring connections to the database. Since the establishment of a database connection is typically a somewhat expensive operation, an application needs a way to get at database connections repeatedly without incurring the full overhead each time. Particularly for server-side web applications, a connection pool is the standard way to maintain a group or "pool" of active database connections which are reused from request to request in a single server process.

Connection Pool Configuration

The `Engine` returned by the `create_engine()` function in most cases has a `QueuePool` integrated, pre-configured with reasonable pooling defaults. If you're reading this section to simply enable pooling- congratulations! You're already done.

The most common `QueuePool` tuning parameters can be passed directly to `create_engine()` as keyword arguments: `pool_size`, `max_overflow`, `pool_recycle` and `pool_timeout`. For example:

```
engine = create_engine('postgres://me@localhost/mydb',
                       pool_size=20, max_overflow=0)
```

In the case of SQLite, a `SingletonThreadPool` is provided instead, to provide compatibility with SQLite's restricted threading model.

Custom Pool Construction

`Pool` instances may be created directly for your own use or to supply to `sqlalchemy.create_engine()` via the `pool=` keyword argument.

Constructing your own pool requires supplying a callable function the Pool can use to create new connections. The function will be called with no arguments.

Through this method, custom connection schemes can be made, such as a using connections from another library's pool, or making a new connection that automatically executes some initialization commands:

```
import sqlalchemy.pool as pool
import psycopg2

def getconn():
    c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')
    # execute an initialization function on the connection before returning
    c.cursor.execute("setup_encodings()")
    return c

p = pool.QueuePool(getconn, max_overflow=10, pool_size=5)
```

Or with SingletonThreadPool:

```
import sqlalchemy.pool as pool
import sqlite

p = pool.SingletonThreadPool(lambda: sqlite.connect(filename='myfile.db'))
```

Builtin Pool Implementations

class **`AssertionPool`**(*creator, **params*)

Bases: `sqlalchemy.pool.Pool`

A Pool that allows at most one checked out connection at any given time.

This will raise an exception if more than one connection is checked out at a time. Useful for debugging code that is using more connections than desired.

`__init__`(*creator, **params*)

Construct an AssertionPool.

Parameters
- *creator* – a callable function that returns a DB-API connection object. The function will be called with parameters.
- *recycle* – If set to non -1, number of seconds between connection recycling, which means upon checkout, if this timeout is surpassed the connection will be closed and replaced with a newly opened connection. Defaults to -1.
- *echo* – If True, connections being pulled and retrieved from the pool will be logged to the standard output, as well as pool sizing information. Echoing can also be achieved by enabling logging for the "sqlalchemy.pool" namespace. Defaults to False.
- *use_threadlocal* – If set to True, repeated calls to `connect()` within the same application thread will be guaranteed to return the same connection object, if one has already been retrieved from the pool and has not been returned yet. Offers a slight performance advantage at the cost of individual transactions by default. The `unique_connection()` method is provided to bypass the threadlocal behavior installed into `connect()`.
- *reset_on_return* – If true, reset the database state of connections returned to the pool. This is typically a ROLLBACK to release locks and transaction resources. Disable at your own peril. Defaults to True.
- *listeners* – A list of `PoolListener`-like objects or dictionaries of callables that receive events when DB-API connections are created, checked out and checked in to the pool.

class **`NullPool`**(*creator, recycle=-1, echo=None, use_threadlocal=False, reset_on_return=True, listeners=None*)

Bases: `sqlalchemy.pool.Pool`

A Pool which does not pool connections.

Instead it literally opens and closes the underlying DB-API connection per each connection open/close.

Reconnect-related functions such as `recycle` and connection invalidation are not supported by this Pool implementation, since no connections are held persistently.

class **`Pool`**(*creator, recycle=-1, echo=None, use_threadlocal=False, reset_on_return=True, listeners=None*)

Bases: `object`

Abstract base class for connection pools.

`__init__`(*creator, recycle=-1, echo=None, use_threadlocal=False, reset_on_return=True, listeners=None*)
Construct a Pool.

Parameters
- *creator* – a callable function that returns a DB-API connection object. The function will be called with parameters.
- *recycle* – If set to non -1, number of seconds between connection recycling, which means upon checkout, if this timeout is surpassed the connection will be closed and replaced with a newly opened connection. Defaults to -1.
- *echo* – If True, connections being pulled and retrieved from the pool will be logged to the standard output, as well as pool sizing information. Echoing can also be achieved by enabling logging for the "sqlalchemy.pool" namespace. Defaults to False.
- *use_threadlocal* – If set to True, repeated calls to `connect()` within the same application thread will be guaranteed to return the same connection object, if one has already been retrieved from the pool and has not been returned yet. Offers a slight performance advantage at the cost of individual transactions by default. The `unique_connection()` method is provided to bypass the threadlocal behavior installed into `connect()`.
- *reset_on_return* – If true, reset the database state of connections returned to the pool. This is typically a ROLLBACK to release locks and transaction resources. Disable at your own peril. Defaults to True.
- *listeners* – A list of `PoolListener`-like objects or dictionaries of callables that receive events when DB-API connections are created, checked out and checked in to the pool.

`add_listener`(*listener*)
Add a `PoolListener`-like object to this pool.

`listener` may be an object that implements some or all of PoolListener, or a dictionary of callables containing implementations of some or all of the named methods in PoolListener.

`connect`()

`create_connection`()

`dispose`()
Dispose of this pool.

This method leaves the possibility of checked-out connections remaining open, It is advised to not reuse the pool once dispose() is called, and to instead use a new pool constructed by the recreate() method.

`do_get`()

`do_return_conn`(*conn*)

`get`()

`log`(*msg*)

`recreate`()
Return a new instance with identical creation arguments.

`return_conn`(*record*)

`status`()

`unique_connection`()

class `QueuePool`(*creator, pool_size=5, max_overflow=10, timeout=30, **params*)
Bases: `sqlalchemy.pool.Pool`

A Pool that imposes a limit on the number of open connections.

`__init__`(*creator, pool_size=5, max_overflow=10, timeout=30, **params*)
Construct a QueuePool.

Parameters
- *creator* – a callable function that returns a DB-API connection object. The function will be called with parameters.

- *pool_size* – The size of the pool to be maintained. This is the largest number of connections that will be kept persistently in the pool. Note that the pool begins with no connections; once this number of connections is requested, that number of connections will remain. Defaults to 5.
- *max_overflow* – The maximum overflow size of the pool. When the number of checked-out connections reaches the size set in pool_size, additional connections will be returned up to this limit. When those additional connections are returned to the pool, they are disconnected and discarded. It follows then that the total number of simultaneous connections the pool will allow is pool_size + *max_overflow*, and the total number of "sleeping" connections the pool will allow is pool_size. *max_overflow* can be set to -1 to indicate no overflow limit; no limit will be placed on the total number of concurrent connections. Defaults to 10.
- *timeout* – The number of seconds to wait before giving up on returning a connection. Defaults to 30.
- *recycle* – If set to non -1, number of seconds between connection recycling, which means upon checkout, if this timeout is surpassed the connection will be closed and replaced with a newly opened connection. Defaults to -1.
- *echo* – If True, connections being pulled and retrieved from the pool will be logged to the standard output, as well as pool sizing information. Echoing can also be achieved by enabling logging for the "sqlalchemy.pool" namespace. Defaults to False.
- *use_threadlocal* – If set to True, repeated calls to `connect()` within the same application thread will be guaranteed to return the same connection object, if one has already been retrieved from the pool and has not been returned yet. Offers a slight performance advantage at the cost of individual transactions by default. The `unique_connection()` method is provided to bypass the threadlocal behavior installed into `connect()`.
- *reset_on_return* – If true, reset the database state of connections returned to the pool. This is typically a ROLLBACK to release locks and transaction resources. Disable at your own peril. Defaults to True.
- *listeners* – A list of `PoolListener`-like objects or dictionaries of callables that receive events when DB-API connections are created, checked out and checked in to the pool.

class **`SingletonThreadPool`**(*creator, pool_size=5, **params*)

Bases: `sqlalchemy.pool.Pool`

A Pool that maintains one connection per thread.

Maintains one connection per each thread, never moving a connection to a thread other than the one which it was created in.

This is used for SQLite, which both does not handle multithreading by default, and also requires a singleton connection if a :memory: database is being used.

Options are the same as those of `Pool`, as well as:

Parameter *pool_size* – The number of threads in which to maintain connections at once. Defaults to five.

`__init__`(*creator, pool_size=5, **params*)

`dispose`()

Dispose of this pool.

class **`StaticPool`**(*creator, **params*)

Bases: `sqlalchemy.pool.Pool`

A Pool of exactly one connection, used for all requests.

Reconnect-related functions such as `recycle` and connection invalidation (which is also used to support auto-reconnect) are not currently supported by this Pool implementation but may be implemented in a future release.

`__init__`(*creator*, ***params*)
Construct a StaticPool.

Parameters
- *creator* – a callable function that returns a DB-API connection object. The function will be called with parameters.
- *echo* – If True, connections being pulled and retrieved from the pool will be logged to the standard output, as well as pool sizing information. Echoing can also be achieved by enabling logging for the "sqlalchemy.pool" namespace. Defaults to False.
- *reset_on_return* – If true, reset the database state of connections returned to the pool. This is typically a ROLLBACK to release locks and transaction resources. Disable at your own peril. Defaults to True.
- *listeners* – A list of `PoolListener`-like objects or dictionaries of callables that receive events when DB-API connections are created, checked out and checked in to the pool.

Pooling Plain DB-API Connections

Any **PEP 249** DB-API module can be "proxied" through the connection pool transparently. Usage of the DB-API is exactly as before, except the `connect()` method will consult the pool. Below we illustrate this with `psycopg2`:

```
import sqlalchemy.pool as pool
import psycopg2 as psycopg

psycopg = pool.manage(psycopg)

# then connect normally
connection = psycopg.connect(database='test', username='scott',
                             password='tiger')
```

This produces a `_DBProxy` object which supports the same `connect()` function as the original DB-API module. Upon connection, a connection proxy object is returned, which delegates its calls to a real DB-API connection object. This connection object is stored persistently within a connection pool (an instance of `Pool`) that corresponds to the exact connection arguments sent to the `connect()` function.

The connection proxy supports all of the methods on the original connection object, most of which are proxied via `__getattr__()`. The `close()` method will return the connection to the pool, and the `cursor()` method will return a proxied cursor object. Both the connection proxy and the cursor proxy will also return the underlying connection to the pool after they have both been garbage collected, which is detected via the `__del__()` method.

Additionally, when connections are returned to the pool, a `rollback()` is issued on the connection unconditionally. This is to release any locks still held by the connection that may have resulted from normal activity.

By default, the `connect()` method will return the same connection that is already checked out in the current thread. This allows a particular connection to be used in a given thread without needing to pass it around between functions. To disable this behavior, specify `use_threadlocal=False` to the `manage()` function.

`manage`(*module*, ***params*)
Return a proxy for a DB-API module that automatically pools connections.

Given a DB-API 2.0 module and pool management parameters, returns a proxy for the module that will automatically pool connections, creating new connection pools for each distinct set of connection arguments sent to the decorated module's connect() function.

Parameters
- *module* – a DB-API 2.0 database module
- *poolclass* – the class used by the pool module to provide pooling. Defaults to `QueuePool`.
- ***params* – will be passed through to *poolclass*

clear_managers()
Remove all current DB-API 2.0 managers.

All pools and connections are disposed.

8.1.3 SQL Statements and Expressions

Functions

The expression package uses functions to construct SQL expressions. The return value of each function is an object instance which is a subclass of `ClauseElement`.

alias(*selectable, alias=None*)
Return an `Alias` object.

An `Alias` represents any `FromClause` with an alternate name assigned within SQL, typically using the `AS` clause when generated, e.g. `SELECT * FROM table AS aliasname`.

Similar functionality is available via the `alias()` method available on all `FromClause` subclasses.

selectable any `FromClause` subclass, such as a table, select statement, etc..

alias string name to be assigned as the alias. If `None`, a random name will be generated.

and_(**clauses*)
Join a list of clauses together using the `AND` operator.

The `&` operator is also overloaded on all `_CompareMixin` subclasses to produce the same result.

asc(*column*)
Return an ascending `ORDER BY` clause element.

e.g.:

```
order_by = [asc(table1.mycol)]
```

between(*ctest, cleft, cright*)
Return a `BETWEEN` predicate clause.

Equivalent of SQL `clausetest BETWEEN clauseleft AND clauseright`.

The `between()` method on all `_CompareMixin` subclasses provides similar functionality.

bindparam(*key, value=None, shortname=None, type_=None, unique=False*)
Create a bind parameter clause with the given key.

value a default value for this bind parameter. a bindparam with a value is called a `value-based bindparam`.

type_ a sqlalchemy.types.TypeEngine object indicating the type of this bind param, will invoke type-specific bind parameter processing

shortname deprecated.

unique if True, bind params sharing the same name will have their underlying `key` modified to a uniquely generated name. mostly useful with value-based bind params.

case(*whens, value=None, else_=None*)
Produce a `CASE` statement.

whens A sequence of pairs, or alternatively a dict, to be translated into “WHEN / THEN” clauses.

value Optional for simple case statements, produces a column expression as in “CASE <expr> WHEN ...”

else_ Optional as well, for case defaults produces the “ELSE” portion of the “CASE” statement.

The expressions used for THEN and ELSE, when specified as strings, will be interpreted as bound values. To specify textual SQL expressions for these, use the text(<string>) construct.

The expressions used for the WHEN criterion may only be literal strings when "value" is present, i.e. CASE table.somecol WHEN "x" THEN "y". Otherwise, literal strings are not accepted in this position, and either the text(<string>) or literal(<string>) constructs must be used to interpret raw string values.

Usage examples:

```
case([(orderline.c.qty > 100, item.c.specialprice),
      (orderline.c.qty > 10, item.c.bulkprice)
    ], else_=item.c.regularprice)
case(value=emp.c.type, whens={
        'engineer': emp.c.salary * 1.1,
        'manager':  emp.c.salary * 3,
    })
```

cast(*clause, totype, **kwargs*)
Return a `CAST` function.

Equivalent of SQL `CAST(clause AS totype)`.

Use with a `TypeEngine` subclass, i.e:

```
cast(table.c.unit_price * table.c.qty, Numeric(10,4))
```

or:

```
cast(table.c.timestamp, DATE)
```

column(*text, type_=None*)
Return a textual column clause, as would be in the columns clause of a `SELECT` statement.

The object returned is an instance of `ColumnClause`, which represents the "syntactical" portion of the schema-level `Column` object.

text the name of the column. Quoting rules will be applied to the clause like any other column name. For textual column constructs that are not to be quoted, use the `literal_column()` function.

type_ an optional `TypeEngine` object which will provide result-set translation for this column.

collate(*expression, collation*)
Return the clause `expression COLLATE collation`.

delete(*table, whereclause=None, **kwargs*)
Return a `Delete` clause element.

Similar functionality is available via the `delete()` method on `Table`.

Parameters
- *table* – The table to be updated.
- *whereclause* – A `ClauseElement` describing the `WHERE` condition of the `UPDATE` statement. Note that the `where()` generative method may be used instead.

desc(*column*)
Return a descending `ORDER BY` clause element.

e.g.:

```
order_by = [desc(table1.mycol)]
```

distinct (*expr*)
Return a `DISTINCT` clause.

except_ (**selects, **kwargs*)
Return an `EXCEPT` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

except_all (**selects, **kwargs*)
Return an `EXCEPT ALL` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

exists (**args, **kwargs*)
Return an `EXISTS` clause as applied to a `Select` object.

Calling styles are of the following forms:

```
# use on an existing select()
s = select([table.c.col1]).where(table.c.col2==5)
s = exists(s)

# construct a select() at once
exists(['*'], **select_arguments).where(criterion)

# columns argument is optional, generates "EXISTS (SELECT *)"
# by default.
exists().where(table.c.col2==5)
```

extract (*field, expr*)
Return the clause `extract(field FROM expr)`.

func
Generate SQL function expressions.

`func` is a special object instance which generates SQL functions based on name-based attributes, e.g.:

```
>>> print func.count(1)
count(:param_1)
```

Any name can be given to *func*. If the function name is unknown to SQLAlchemy, it will be rendered exactly as is. For common SQL functions which SQLAlchemy is aware of, the name may be interpreted as a *generic function* which will be compiled appropriately to the target database:

```
>>> print func.current_timestamp()
CURRENT_TIMESTAMP
```

To call functions which are present in dot-separated packages, specify them in the same manner:

```
>>> print func.stats.yield_curve(5, 10)
stats.yield_curve(:yield_curve_1, :yield_curve_2)
```

SQLAlchemy can be made aware of the return type of functions to enable type-specific lexical and result-based behavior. For example, to ensure that a string-based function returns a Unicode value and is similarly treated as a string in expressions, specify `Unicode` as the type:

```
>>> print func.my_string(u'hi', type_=Unicode) + ' ' + \
... func.my_string(u'there', type_=Unicode)
my_string(:my_string_1) || :my_string_2 || my_string(:my_string_3)
```

Functions which are interpreted as "generic" functions know how to calculate their return type automatically. For a listing of known generic functions, see *Generic Functions*.

insert (*table, values=None, inline=False, **kwargs*)
Return an `Insert` clause element.

Similar functionality is available via the `insert()` method on `Table`.

Parameters
- *table* – The table to be inserted into.
- *values* – A dictionary which specifies the column specifications of the `INSERT`, and is optional. If left as None, the column specifications are determined from the bind parameters used during the compile phase of the `INSERT` statement. If the bind parameters also are None during the compile phase, then the column specifications will be generated from the full list of table columns. Note that the `values()` generative method may also be used for this.
- *prefixes* – A list of modifier keywords to be inserted between INSERT and INTO. Alternatively, the `prefix_with()` generative method may be used.
- *inline* – if True, SQL defaults will be compiled 'inline' into the statement and not pre-executed.

If both *values* and compile-time bind parameters are present, the compile-time bind parameters override the information specified within *values* on a per-key basis.

The keys within *values* can be either `Column` objects or their string identifiers. Each key may reference one of:

- a literal data value (i.e. string, number, etc.);
- a Column object;
- a SELECT statement.

If a `SELECT` statement is specified which references this `INSERT` statement's table, the statement will be correlated against the `INSERT` statement.

intersect (**selects, **kwargs*)
Return an `INTERSECT` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

intersect_all (**selects, **kwargs*)
Return an `INTERSECT ALL` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

`join`(*left, right, onclause=None, isouter=False*)
Return a `JOIN` clause element (regular inner join).

The returned object is an instance of `Join`.

Similar functionality is also available via the `join()` method on any `FromClause`.

left The left side of the join.

right The right side of the join.

onclause Optional criterion for the `ON` clause, is derived from foreign key relationships established between left and right otherwise.

To chain joins together, use the `join()` or `outerjoin()` methods on the resulting `Join` object.

`label`(*name, obj*)
Return a `_Label` object for the given `ColumnElement`.

A label changes the name of an element in the columns clause of a `SELECT` statement, typically via the `AS` SQL keyword.

This functionality is more conveniently available via the `label()` method on `ColumnElement`.

name label name

obj a `ColumnElement`.

`literal`(*value, type_=None*)
Return a literal clause, bound to a bind parameter.

Literal clauses are created automatically when non- `ClauseElement` objects (such as strings, ints, dates, etc.) are used in a comparison operation with a `_CompareMixin` subclass, such as a `Column` object. Use this function to force the generation of a literal clause, which will be created as a `_BindParamClause` with a bound value.

value the value to be bound. Can be any Python object supported by the underlying DB-API, or is translatable via the given type argument.

type_ an optional `TypeEngine` which will provide bind-parameter translation for this literal.

`literal_column`(*text, type_=None*)
Return a textual column expression, as would be in the columns clause of a `SELECT` statement.

The object returned supports further expressions in the same way as any other column object, including comparison, math and string operations. The type_ parameter is important to determine proper expression behavior (such as, '+' means string concatenation or numerical addition based on the type).

text the text of the expression; can be any SQL expression. Quoting rules will not be applied. To specify a column-name expression which should be subject to quoting rules, use the `column()` function.

type_ an optional `TypeEngine` object which will provide result-set translation and additional expression semantics for this column. If left as None the type will be NullType.

`not_`(*clause*)
Return a negation of the given clause, i.e. `NOT(clause)`.

The ~ operator is also overloaded on all `_CompareMixin` subclasses to produce the same result.

`null`()
Return a `_Null` object, which compiles to `NULL` in a sql statement.

`or_`(**clauses*)
Join a list of clauses together using the `OR` operator.

The `|` operator is also overloaded on all `_CompareMixin` subclasses to produce the same result.

outparam (*key, type_=None*)
Create an 'OUT' parameter for usage in functions (stored procedures), for databases which support them.

The `outparam` can be used like a regular function parameter. The "output" value will be available from the `ResultProxy` object via its `out_parameters` attribute, which returns a dictionary containing the values.

outerjoin (*left, right, onclause=None*)
Return an `OUTER JOIN` clause element.

The returned object is an instance of `Join`.

Similar functionality is also available via the `outerjoin()` method on any `FromClause`.

left The left side of the join.

right The right side of the join.

onclause Optional criterion for the `ON` clause, is derived from foreign key relationships established between left and right otherwise.

To chain joins together, use the `join()` or `outerjoin()` methods on the resulting `Join` object.

select (*columns=None, whereclause=None, from_obj=, [], **kwargs*)
Returns a `SELECT` clause element.

Similar functionality is also available via the `select()` method on any `FromClause`.

The returned object is an instance of `Select`.

All arguments which accept `ClauseElement` arguments also accept string arguments, which will be converted as appropriate into either `text()` or `literal_column()` constructs.

columns A list of `ClauseElement` objects, typically `ColumnElement` objects or subclasses, which will form the columns clause of the resulting statement. For all members which are instances of `Selectable`, the individual `ColumnElement` members of the `Selectable` will be added individually to the columns clause. For example, specifying a `Table` instance will result in all the contained `Column` objects within to be added to the columns clause.

This argument is not present on the form of `select()` available on `Table`.

whereclause A `ClauseElement` expression which will be used to form the `WHERE` clause.

from_obj A list of `ClauseElement` objects which will be added to the `FROM` clause of the resulting statement. Note that "from" objects are automatically located within the columns and whereclause ClauseElements. Use this parameter to explicitly specify "from" objects which are not automatically locatable. This could include `Table` objects that aren't otherwise present, or `Join` objects whose presence will supercede that of the `Table` objects already located in the other clauses.

****kwargs** Additional parameters include:

autocommit indicates this SELECT statement modifies the database, and should be subject to autocommit behavior if no transaction has been started.

prefixes a list of strings or `ClauseElement` objects to include directly after the SELECT keyword in the generated statement, for dialect-specific query features.

distinct=False when `True`, applies a `DISTINCT` qualifier to the columns clause of the resulting statement.

use_labels=False when `True`, the statement will be generated using labels for each column in the columns clause, which qualify each column with its parent table's (or aliases) name so that name conflicts between columns in different tables don't occur. The format of the label is <tablename>_<column>. The "c" collection of the resulting `Select` object will use these names as well for targeting column members.

for_update=False when `True`, applies `FOR UPDATE` to the end of the resulting statement. Certain database dialects also support alternate values for this parameter, for example mysql supports "read" which translates to `LOCK IN SHARE MODE`, and oracle supports "nowait" which translates to `FOR UPDATE NOWAIT`.

correlate=True indicates that this `Select` object should have its contained `FromClause` elements "correlated" to an enclosing `Select` object. This means that any `ClauseElement` instance within the "froms" collection of this `Select` which is also present in the "froms" collection of an enclosing select will not be rendered in the `FROM` clause of this select statement.

group_by a list of `ClauseElement` objects which will comprise the `GROUP BY` clause of the resulting select.

having a `ClauseElement` that will comprise the `HAVING` clause of the resulting select when `GROUP BY` is used.

order_by a scalar or list of `ClauseElement` objects which will comprise the `ORDER BY` clause of the resulting select.

limit=None a numerical value which usually compiles to a `LIMIT` expression in the resulting select. Databases that don't support `LIMIT` will attempt to provide similar functionality.

offset=None a numeric value which usually compiles to an `OFFSET` expression in the resulting select. Databases that don't support `OFFSET` will attempt to provide similar functionality.

bind=None an `Engine` or `Connection` instance to which the resulting `` Select ` object will be bound.  The ``Select`` object will otherwise automatically bind to whatever `Connectable` instances can be located within its contained `ClauseElement` members.

scalar=False deprecated. Use select(...).as_scalar() to create a "scalar column" proxy for an existing Select object.

subquery(*alias, *args, **kwargs*)

Return an `Alias` object derived from a `Select`.

name alias name

*args, **kwargs

all other arguments are delivered to the `select()` function.

table(*name, *columns*)

Return a `TableClause` object.

This is a primitive version of the `Table` object, which is a subclass of this object.

text(*text, bind=None, *args, **kwargs*)

Create literal text to be inserted into a query.

When constructing a query from a `select()`, `update()`, `insert()` or `delete()`, using plain strings for argument values will usually result in text objects being created automatically. Use this function when creating textual clauses outside of other `ClauseElement` objects, or optionally wherever plain text is to be used.

text the text of the SQL statement to be created. use `:<param>` to specify bind parameters; they will be compiled to their engine-specific format.

bind an optional connection or engine to be used for this text query.

autocommit=True indicates this SELECT statement modifies the database, and should be subject to autocommit behavior if no transaction has been started.

bindparams a list of `bindparam()` instances which can be used to define the types and/or initial values for the bind parameters within the textual statement; the keynames of the bindparams must match those within the text of the statement. The types will be used for pre-processing on bind values.

typemap a dictionary mapping the names of columns represented in the `SELECT` clause of the textual statement to type objects, which will be used to perform post-processing on columns within the result set (for textual statements that produce result sets).

union(**selects, **kwargs*)

Return a `UNION` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

A similar `union()` method is available on all `FromClause` subclasses.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

`union_all`(**selects, **kwargs*)

Return a `UNION ALL` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

A similar `union_all()` method is available on all `FromClause` subclasses.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

`update`(*table, whereclause=None, values=None, inline=False, **kwargs*)

Return an `Update` clause element.

Similar functionality is available via the `update()` method on `Table`.

Parameters
- *table* – The table to be updated.
- *whereclause* – A `ClauseElement` describing the `WHERE` condition of the `UPDATE` statement. Note that the `where()` generative method may also be used for this.
- *values* – A dictionary which specifies the `SET` conditions of the `UPDATE`, and is optional. If left as None, the `SET` conditions are determined from the bind parameters used during the compile phase of the `UPDATE` statement. If the bind parameters also are None during the compile phase, then the `SET` conditions will be generated from the full list of table columns. Note that the `values()` generative method may also be used for this.
- *inline* – if True, SQL defaults will be compiled 'inline' into the statement and not pre-executed.

If both *values* and compile-time bind parameters are present, the compile-time bind parameters override the information specified within *values* on a per-key basis.

The keys within *values* can be either `Column` objects or their string identifiers. Each key may reference one of:

- a literal data value (i.e. string, number, etc.);
- a Column object;
- a SELECT statement.

If a `SELECT` statement is specified which references this `UPDATE` statement's table, the statement will be correlated against the `UPDATE` statement.

Classes

class `Alias`(*selectable, alias=None*)

Bases: `sqlalchemy.sql.expression.FromClause`

Represents an table or selectable alias (AS).

Represents an alias, as typically applied to any table or sub-select within a SQL statement using the `AS` keyword (or without the keyword on certain databases such as Oracle).

This object is constructed from the `alias()` module level function as well as the `alias()` method available on all `FromClause` subclasses.

`__init__`(*selectable, alias=None*)

class `ClauseElement`()

Bases: `sqlalchemy.sql.visitors.Visitable`

Base class for elements of a programmatically constructed SQL expression.

bind
Returns the Engine or Connection to which this ClauseElement is bound, or None if none found.

compare (*other*)
Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

compile (*bind=None, column_keys=None, compiler=None, dialect=None, inline=False*)
Compile this SQL expression.

The return value is a `Compiled` object. Calling *str()* or *unicode()* on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the *params* accessor.

Parameters
- *bind* – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- *column_keys* – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- *compiler* – A `Compiled` instance which will be used to compile this expression. This argument takes precedence over the *bind* and *dialect* arguments as well as this `ClauseElement`'s bound engine, if any.
- *dialect* – A `Dialect` instance frmo which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- *inline* – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.

execute (**multiparams, **params*)
Compile and execute this `ClauseElement`.

get_children (***kwargs*)
Return immediate child elements of this `ClauseElement`.

This is used for visit traversal.

**kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

params (**optionaldict, **kwargs*)
Return a copy with `bindparam()` elments replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

scalar (**multiparams, **params*)
Compile and execute this `ClauseElement`, returning the result's scalar representation.

unique_params (**optionaldict, **kwargs*)
Return a copy with `bindparam()` elments replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class **ColumnClause**(*text, selectable=None, type_=None, is_literal=False*)

Bases: `sqlalchemy.sql.expression._Immutable`, `sqlalchemy.sql.expression.ColumnElement`

Represents a generic column expression from any textual string.

This includes columns associated with tables, aliases and select statements, but also any arbitrary text. May or may not be bound to an underlying `Selectable`. `ColumnClause` is usually created publically via the `column()` function or the `literal_column()` function.

text the text of the element.

selectable parent selectable.

type `TypeEngine` object which can associate this `ColumnClause` with a type.

is_literal if True, the `ColumnClause` is assumed to be an exact expression that will be delivered to the output with no quoting rules applied regardless of case sensitive settings. the `literal_column()` function is usually used to create such a `ColumnClause`.

__init__(*text, selectable=None, type_=None, is_literal=False*)

class **ColumnCollection**(**cols*)

Bases: `sqlalchemy.util.OrderedProperties`

An ordered dictionary that stores a list of ColumnElement instances.

Overrides the `__eq__()` method to produce SQL clauses between sets of correlated columns.

__init__(**cols*)

add(*column*)

Add a column to this collection.

The key attribute of the column will be used as the hash key for this dictionary.

replace(*column*)

add the given column to this collection, removing unaliased versions of this column as well as existing columns with the same key.

e.g.:

```
t = Table('sometable', metadata, Column('col1', Integer))
t.columns.replace(Column('col1', Integer, key='columnone'))
```

will remove the original 'col1' from the collection, and add the new column under the name 'columnname'.

Used by schema.Column to override columns during table reflection.

class **ColumnElement**()

Bases: `sqlalchemy.sql.expression.ClauseElement`, `sqlalchemy.sql.expression._CompareMixin`

Represent an element that is usable within the "column clause" portion of a `SELECT` statement.

This includes columns associated with tables, aliases, and subqueries, expressions, function calls, SQL keywords such as `NULL`, literals, etc. `ColumnElement` is the ultimate base class for all such elements.

`ColumnElement` supports the ability to be a *proxy* element, which indicates that the `ColumnElement` may be associated with a `Selectable` which was derived from another `Selectable`. An example of a "derived" `Selectable` is an `Alias` of a `Table`.

A `ColumnElement`, by subclassing the `_CompareMixin` mixin class, provides the ability to generate new `ClauseElement` objects using Python expressions. See the `_CompareMixin` docstring for more details.

shares_lineage(*othercolumn*)

Return True if the given `ColumnElement` has a common ancestor to this `ColumnElement`.

class **_CompareMixin**()

Bases: `sqlalchemy.sql.expression.ColumnOperators`

Defines comparison and math operations for `ClauseElement` instances.

asc()
Produce a ASC clause, i.e. `<columnname> ASC`

between(*cleft, cright*)
Produce a BETWEEN clause, i.e. `<column> BETWEEN <cleft> AND <cright>`

collate(*collation*)
Produce a COLLATE clause, i.e. `<column> COLLATE utf8_bin`

contains(*other, escape=None*)
Produce the clause `LIKE '%<other>%'`

desc()
Produce a DESC clause, i.e. `<columnname> DESC`

distinct()
Produce a DISTINCT clause, i.e. `DISTINCT <columnname>`

endswith(*other, escape=None*)
Produce the clause `LIKE '%<other>'`

in_(*other*)

label(*name*)
Produce a column label, i.e. `<columnname> AS <name>`.

if 'name' is None, an anonymous label name will be generated.

match(*other*)
Produce a MATCH clause, i.e. `MATCH '<other>'`

The allowed contents of `other` are database backend specific.

op(*operator*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

operator a string which will be output as the infix operator between this `ClauseElement` and the expression passed to the generated function.

operate(*op, *other, **kwargs*)

reverse_operate(*op, other, **kwargs*)

startswith(*other, escape=None*)
Produce the clause `LIKE '<other>%'`

class **ColumnOperators**()
Defines comparison and math operations.

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

asc()

between(*cleft, cright*)

collate(*collation*)

concat(*other*)

contains(*other, **kwargs*)

desc()

distinct()

endswith (*other, **kwargs*)

ilike (*other, escape=None*)

in_ (*other*)

like (*other, escape=None*)

match (*other, **kwargs*)

op (*opstring*)

operate (*op, *other, **kwargs*)

reverse_operate (*op, other, **kwargs*)

startswith (*other, **kwargs*)

timetuple
Hack, allows datetime objects to be compared on the LHS.

class **CompoundSelect** (*keyword, *selects, **kwargs*)
Bases: `sqlalchemy.sql.expression._SelectBaseMixin`, `sqlalchemy.sql.expression.FromClause`

Forms the basis of `UNION`, `UNION ALL`, and other SELECT-based set operations.

__init__ (*keyword, *selects, **kwargs*)

class **Delete** (*table, whereclause, bind=None, **kwargs*)
Bases: `sqlalchemy.sql.expression._UpdateBase`

Represent a DELETE construct.

The `Delete` object is created using the `delete()` function.

where (*whereclause*)
Add the given WHERE clause to a newly returned delete construct.

class **FromClause** ()
Bases: `sqlalchemy.sql.expression.Selectable`

Represent an element that can be used within the `FROM` clause of a `SELECT` statement.

alias (*name=None*)
return an alias of this `FromClause`.

For table objects, this has the effect of the table being rendered as `tablename AS aliasname` in a SELECT statement. For select objects, the effect is that of creating a named subquery, i.e. `(select ...) AS aliasname`. The `alias()` method is the general way to create a "subquery" out of an existing SELECT.

The `name` parameter is optional, and if left blank an "anonymous" name will be generated at compile time, guaranteed to be unique against other anonymous constructs used in the same statement.

c
Return the collection of Column objects contained by this FromClause.

columns
Return the collection of Column objects contained by this FromClause.

correspond_on_equivalents (*column, equivalents*)
Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column (*column, require_embedded=False*)
Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common anscestor column.

Parameters
- *column* – the target `ColumnElement` to be matched
- *require_embedded* – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common anscestor with one of the exported columns of this `FromClause`.

`count` (*whereclause=None, **params*)
return a SELECT COUNT generated against this `FromClause`.

`description`
a brief description of this FromClause.

Used primarily for error message formatting.

`foreign_keys`
Return the collection of ForeignKey objects which this FromClause references.

`is_derived_from` (*fromclause*)
Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

`join` (*right, onclause=None, isouter=False*)
return a join of this `FromClause` against another `FromClause`.

`outerjoin` (*right, onclause=None*)
return an outer join of this `FromClause` against another `FromClause`.

`primary_key`
Return the collection of Column objects which comprise the primary key of this FromClause.

`replace_selectable` (*old, alias*)
replace all occurences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

`select` (*whereclause=None, **params*)
return a SELECT of this `FromClause`.

class `Insert` (*table, values=None, inline=False, bind=None, prefixes=None, **kwargs*)
Bases: `sqlalchemy.sql.expression._ValuesBase`

Represent an INSERT construct.

The `Insert` object is created using the `insert()` function.

`prefix_with` (*clause*)
Add a word or expression between INSERT and INTO. Generative.

If multiple prefixes are supplied, they will be separated with spaces.

`values` (**args, **kwargs*)
specify the VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

****kwargs** key=<somevalue> arguments

***args** A single dictionary can be sent as the first positional argument. This allows non-string based keys, such as Column objects, to be used.

class `Join` (*left, right, onclause=None, isouter=False*)
Bases: `sqlalchemy.sql.expression.FromClause`

represent a `JOIN` construct between two `FromClause` elements.

The public constructor function for `Join` is the module-level `join()` function, as well as the `join()` method available off all `FromClause` subclasses.

`__init__` (*left, right, onclause=None, isouter=False*)

`alias` (*name=None*)
Create a `Select` out of this `Join` clause and return an `Alias` of it.

The `Select` is not correlating.

`select` (*whereclause=None, fold_equivalents=False, **kwargs*)
Create a `Select` from this `Join`.

Parameters • *whereclause* – the WHERE criterion that will be sent to the `select()` function

- *fold_equivalents* – based on the join criterion of this `Join`, do not include repeat column names in the column list of the resulting select, for columns that are calculated to be "equivalent" based on the join criterion of this `Join`. This will recursively apply to any joins directly nested by this one as well. This flag is specific to a particular use case by the ORM and will be deprecated in 0.6.
- ***kwargs* – all other kwargs are sent to the underlying `select()` function.

class `Select`(*columns, whereclause=None, from_obj=None, distinct=False, having=None, correlate=True, prefixes=None, **kwargs*)

Bases: `sqlalchemy.sql.expression._SelectBaseMixin`, `sqlalchemy.sql.expression.FromClause`

Represents a `SELECT` statement.

Select statements support appendable clauses, as well as the ability to execute themselves and return a result set.

`__init__`(*columns, whereclause=None, from_obj=None, distinct=False, having=None, correlate=True, prefixes=None, **kwargs*)

Construct a Select object.

The public constructor for Select is the `select()` function; see that function for argument descriptions.

Additional generative and mutator methods are available on the `_SelectBaseMixin` superclass.

`append_column`(*column*)

append the given column expression to the columns clause of this select() construct.

`append_correlation`(*fromclause*)

append the given correlation expression to this select() construct.

`append_from`(*fromclause*)

append the given FromClause expression to this select() construct's FROM clause.

`append_having`(*having*)

append the given expression to this select() construct's HAVING criterion.

The expression will be joined to existing HAVING criterion via AND.

`append_prefix`(*clause*)

append the given columns clause prefix expression to this select() construct.

`append_whereclause`(*whereclause*)

append the given expression to this select() construct's WHERE criterion.

The expression will be joined to existing WHERE criterion via AND.

`column`(*column*)

return a new select() construct with the given column expression added to its columns clause.

`correlate`(**fromclauses*)

return a new select() construct which will correlate the given FROM clauses to that of an enclosing select(), if a match is found.

By "match", the given fromclause must be present in this select's list of FROM objects and also present in an enclosing select's list of FROM objects.

Calling this method turns off the select's default behavior of "auto-correlation". Normally, select() auto-correlates all of its FROM clauses to those of an embedded select when compiled.

If the fromclause is None, correlation is disabled for the returned select().

`distinct`()

return a new select() construct which will apply DISTINCT to its columns clause.

`except_`(*other, **kwargs*)

return a SQL EXCEPT of this select() construct against the given selectable.

`except_all`(*other, **kwargs*)

return a SQL EXCEPT ALL of this select() construct against the given selectable.

froms
Return the displayed list of FromClause elements.

get_children(*column_collections=True, **kwargs*)
return child elements as per the ClauseElement specification.

having(*having*)
return a new select() construct with the given expression added to its HAVING clause, joined to the existing clause via AND, if any.

inner_columns
an iterator of all ColumnElement expressions which would be rendered into the columns clause of the resulting SELECT statement.

intersect(*other, **kwargs*)
return a SQL INTERSECT of this select() construct against the given selectable.

intersect_all(*other, **kwargs*)
return a SQL INTERSECT ALL of this select() construct against the given selectable.

prefix_with(*clause*)
return a new select() construct which will apply the given expression to the start of its columns clause, not using any commas.

select_from(*fromclause*)
return a new select() construct with the given FROM expression applied to its list of FROM objects.

self_group(*against=None*)
return a 'grouping' construct as per the ClauseElement specification.

This produces an element that can be embedded in an expression. Note that this method is called automatically as needed when constructing expressions.

union(*other, **kwargs*)
return a SQL UNION of this select() construct against the given selectable.

union_all(*other, **kwargs*)
return a SQL UNION ALL of this select() construct against the given selectable.

where(*whereclause*)
return a new select() construct with the given expression added to its WHERE clause, joined to the existing clause via AND, if any.

with_only_columns(*columns*)
return a new select() construct with its columns clause replaced with the given columns.

class Selectable()
Bases: `sqlalchemy.sql.expression.ClauseElement`

mark a class as being selectable

class TableClause(*name, *columns*)
Bases: `sqlalchemy.sql.expression._Immutable`, `sqlalchemy.sql.expression.FromClause`

Represents a "table" construct.

Note that this represents tables only as another syntactical construct within SQL expressions; it does not provide schema-level functionality.

__init__(*name, *columns*)

delete(*whereclause=None, **kwargs*)
Generate a `delete()` construct.

insert(*values=None, inline=False, **kwargs*)
Generate an `insert()` construct.

update(*whereclause=None, values=None, inline=False, **kwargs*)
Generate an `update()` construct.

class **Update** (*table, whereclause, values=None, inline=False, bind=None, **kwargs*)
Bases: `sqlalchemy.sql.expression._ValuesBase`

Represent an Update construct.

The `Update` object is created using the `update()` function.

where (*whereclause*)
return a new update() construct with the given expression added to its WHERE clause, joined to the existing clause via AND, if any.

values (*args, **kwargs*)
specify the VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

****kwargs** key=<somevalue> arguments

***args** A single dictionary can be sent as the first positional argument. This allows non-string based keys, such as Column objects, to be used.

Generic Functions

SQL functions which are known to SQLAlchemy with regards to database-specific rendering, return types and argument behavior. Generic functions are invoked like all SQL functions, using the `func` attribute:

```
select([func.count()]).select_from(sometable)
```

class **AnsiFunction** (***kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

__init__ (***kwargs*)

class **GenericFunction** (*type_=None, args=(), **kwargs*)
Bases: `sqlalchemy.sql.expression.Function`

__init__ (*type_=None, args=(), **kwargs*)

class **ReturnTypeFromArgs** (**args, **kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

Define a function whose return type is the same as its arguments.

__init__ (**args, **kwargs*)

class **char_length** (*arg, **kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

__init__ (*arg, **kwargs*)

class **coalesce** (**args, **kwargs*)
Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

class **concat** (**args, **kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

__init__ (**args, **kwargs*)

class **count** (*expression=None, **kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

The ANSI COUNT aggregate function. With no arguments, emits COUNT *.

__init__ (*expression=None, **kwargs*)

class **current_date** (***kwargs*)
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`current_time`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`current_timestamp`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`current_user`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`localtime`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`localtimestamp`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`max`** (**args, **kwargs*)
    Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

class **`min`** (**args, **kwargs*)
    Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

class **`now`** (*type_=None, args=(), **kwargs*)
    Bases: `sqlalchemy.sql.functions.GenericFunction`

class **`random`** (**args, **kwargs*)
    Bases: `sqlalchemy.sql.functions.GenericFunction`

    `__init__` (**args, **kwargs*)

class **`session_user`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`sum`** (**args, **kwargs*)
    Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

class **`sysdate`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

class **`user`** (***kwargs*)
    Bases: `sqlalchemy.sql.functions.AnsiFunction`

8.1.4 Database Schema

SQLAlchemy schema definition language. For more usage examples, see *Database Meta Data*.

Tables and Columns

class **`Column`** (**args, **kwargs*)
    Bases: `sqlalchemy.schema.SchemaItem`, `sqlalchemy.sql.expression.ColumnClause`

    Represents a column in a database table.

    `__init__` (**args, **kwargs*)
        Construct a new `Column` object.

        Parameters
        - *name* – The name of this column as represented in the database. This argument may be the first positional argument, or specified via keyword.
          Names which contain no upper case characters will be treated as case insensitive names, and will not be quoted unless they are a reserved word. Names with any number of upper case characters will be quoted and sent exactly. Note that this behavior applies even for databases which standardize upper case names as case insensitive such as Oracle.

The name field may be omitted at construction time and applied later, at any time before the Column is associated with a `Table`. This is to support convenient usage within the `declarative` extension.

- *type_* – The column's type, indicated using an instance which subclasses `AbstractType`. If no arguments are required for the type, the class of the type can be sent as well, e.g.:

  ```
  # use a type with arguments
  Column('data', String(50))

  # use no arguments
  Column('level', Integer)
  ```

  The `type` argument may be the second positional argument or specified by keyword.
  If this column also contains a `ForeignKey`, the type argument may be left as `None` in which case the type assigned will be that of the referenced column.
- **args* – Additional positional arguments include various `SchemaItem` derived constructs which will be applied as options to the column. These include instances of `Constraint`, `ForeignKey`, `ColumnDefault`, and `Sequence`. In some cases an equivalent keyword argument is available such as `server_default`, `default` and `unique`.
- *autoincrement* – This flag may be set to `False` to disable SQLAlchemy indicating at the DDL level that an integer primary key column should have autoincrementing behavior. This is an oft misunderstood flag and has no effect whatsoever unless all of the following conditions are met:
  - The column is of the `Integer` datatype.
  - The column has the `primary_key` flag set, or is otherwise a member of a `PrimaryKeyConstraint` on this table.
  - a CREATE TABLE statement is being issued via `create()` or `create_all()`. The flag has no relevance at any other time.
  - The database supports autoincrementing behavior, such as Postgres or MySQL, and this behavior can be disabled (which does not include SQLite).
- *default* – A scalar, Python callable, or `ClauseElement` representing the *default value* for this column, which will be invoked upon insert if this column is otherwise not specified in the VALUES clause of the insert. This is a shortcut to using `ColumnDefault` as a positional argument.
  Contrast this argument to `server_default` which creates a default generator on the database side.
- *key* – An optional string identifier which will identify this `Column` object on the `Table`. When a key is provided, this is the only identifier referencing the `Column` within the application, including ORM attribute mapping; the `name` field is used only when rendering SQL.
- *index* – When `True`, indicates that the column is indexed. This is a shortcut for using a `Index` construct on the table. To specify indexes with explicit names or indexes that contain multiple columns, use the `Index` construct instead.
- *info* – A dictionary which defaults to `{}`. A space to store application specific data. This must be a dictionary.
- *nullable* – If set to the default of `True`, indicates the column will be rendered as allowing NULL, else it's rendered as NOT NULL. This parameter is only used when issuing CREATE TABLE statements.
- *onupdate* – A scalar, Python callable, or `ClauseElement` representing a default value to be applied to the column within UPDATE statements, which wil be invoked upon update if this column is not present in the SET clause of the update. This is a shortcut to using `ColumnDefault` as a positional argument with `for_update=True`.

- *primary_key* – If `True`, marks this column as a primary key column. Multiple columns can have this flag set to specify composite primary keys. As an alternative, the primary key of a `Table` can be specified via an explicit `PrimaryKeyConstraint` object.
- *server_default* – A `FetchedValue` instance, str, Unicode or `text()` construct representing the DDL DEFAULT value for the column.
  String types will be emitted as-is, surrounded by single quotes:
  ```
  Column('x', Text, server_default="val")

  x TEXT DEFAULT 'val'
  ```
  A `text()` expression will be rendered as-is, without quotes:
  ```
  Column('y', DateTime, server_default=text('NOW()'))0

  y DATETIME DEFAULT NOW()
  ```
  Strings and text() will be converted into a `DefaultClause` object upon initialization.
  Use `FetchedValue` to indicate that an already-existing column will generate a default value on the database side which will be available to SQLAlchemy for post-fetch after inserts. This construct does not specify any DDL and the implementation is left to the database, such as via a trigger.
- *server_onupdate* – A `FetchedValue` instance representing a database-side default generation function. This indicates to SQLAlchemy that a newly generated value will be available after updates. This construct does not specify any DDL and the implementation is left to the database, such as via a trigger.
- *quote* – Force quoting of this column's name on or off, corresponding to `True` or `False`. When left at its default of `None`, the column identifier will be quoted according to whether the name is case sensitive (identifiers with at least one upper case character are treated as case sensitive), or if it's a reserved word. This flag is only needed to force quoting of a reserved word which is not known by the SQLAlchemy dialect.
- *unique* – When `True`, indicates that this column contains a unique constraint, or if `index` is `True` as well, indicates that the `Index` should be created with the unique flag. To specify multiple columns in the constraint/index or to specify an explicit name, use the `UniqueConstraint` or `Index` constructs explicitly.

append_foreign_key(*fk*)

asc()
Produce a ASC clause, i.e. `<columnname> ASC`

between(*cleft, cright*)
Produce a BETWEEN clause, i.e. `<column> BETWEEN <cleft> AND <cright>`

bind

collate(*collation*)
Produce a COLLATE clause, i.e. `<column> COLLATE utf8_bin`

compare(*other*)
Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

compile(*bind=None, column_keys=None, compiler=None, dialect=None, inline=False*)
Compile this SQL expression.

The return value is a `Compiled` object. Calling *str()* or *unicode()* on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the *params* accessor.

Parameters
- *bind* – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.

- *column_keys* – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- *compiler* – A `Compiled` instance which will be used to compile this expression. This argument takes precedence over the *bind* and *dialect* arguments as well as this `ClauseElement`'s bound engine, if any.
- *dialect* – A `Dialect` instance frmo which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- *inline* – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.

concat(*other*)

contains(*other, escape=None*)
Produce the clause `LIKE '%<other>%'`

copy(***kw*)
Create a copy of this `Column`, unitialized.

This is used in `Table.tometadata`.

desc()
Produce a DESC clause, i.e. `<columnname> DESC`

distinct()
Produce a DISTINCT clause, i.e. `DISTINCT <columnname>`

endswith(*other, escape=None*)
Produce the clause `LIKE '%<other>'`

execute(**multiparams, **params*)
Compile and execute this `ClauseElement`.

get_children(*schema_visitor=False, **kwargs*)

ilike(*other, escape=None*)

in_(*other*)

info

label(*name*)

like(*other, escape=None*)

match(*other*)
Produce a MATCH clause, i.e. `MATCH '<other>'`

The allowed contents of `other` are database backend specific.

op(*operator*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

operator a string which will be output as the infix operator between this `ClauseElement` and the expression passed to the generated function.

operate(*op, *other, **kwargs*)

params (**optionaldict, **kwargs*)
Return a copy with `bindparam()` elments replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

references (*column*)
Return True if this Column references the given column via foreign key.

reverse_operate (*op, other, **kwargs*)

scalar (**multiparams, **params*)
Compile and execute this `ClauseElement`, returning the result's scalar representation.

self_group (*against=None*)

shares_lineage (*othercolumn*)
Return True if the given `ColumnElement` has a common ancestor to this `ColumnElement`.

startswith (*other, escape=None*)
Produce the clause `LIKE '<other>%'`

unique_params (**optionaldict, **kwargs*)
Return a copy with `bindparam()` elments replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class **MetaData** (*bind=None, reflect=False*)
Bases: `sqlalchemy.schema.SchemaItem`

A collection of Tables and their associated schema constructs.

Holds a collection of Tables and an optional binding to an `Engine` or `Connection`. If bound, the `Table` objects in the collection and their columns may participate in implicit SQL execution.

The *Table* objects themselves are stored in the *metadata.tables* dictionary.

The `bind` property may be assigned to dynamically. A common pattern is to start unbound and then bind later when an engine is available:

```
metadata = MetaData()
# define tables
Table('mytable', metadata, ...)
# connect to an engine later, perhaps after loading a URL from a
# configuration file
metadata.bind = an_engine
```

MetaData is a thread-safe object after tables have been explicitly defined or loaded via reflection.

__init__ (*bind=None, reflect=False*)
Create a new MetaData object.

bind An Engine or Connection to bind to. May also be a string or URL instance, these are passed to create_engine() and this MetaData will be bound to the resulting engine.

reflect Optional, automatically load all tables from the bound database. Defaults to False. `bind` is required when this option is set. For finer control over loaded tables, use the `reflect` method of `MetaData`.

append_ddl_listener (*event, listener*)

Append a DDL event listener to this `MetaData`.

The `listener` callable will be triggered when this `MetaData` is involved in DDL creates or drops, and will be invoked either before all Table-related actions or after.

Arguments are:

event One of `MetaData.ddl_events`; 'before-create', 'after-create', 'before-drop' or 'after-drop'.

listener A callable, invoked with three positional arguments:

- **event** The event currently being handled
- **schema_item** The `MetaData` object being operated upon
- **bind** The `Connection` bueing used for DDL execution.

Listeners are added to the MetaData's `ddl_listeners` attribute.

Note: MetaData listeners are invoked even when `Tables` are created in isolation. This may change in a future release. I.e.:

```
# triggers all MetaData and Table listeners:
metadata.create_all()

# triggers MetaData listeners too:
some.table.create()
```

bind

An Engine or Connection to which this MetaData is bound.

This property may be assigned an `Engine` or `Connection`, or assigned a string or URL to automatically create a basic `Engine` for this bind with `create_engine()`.

clear ()

Clear all Table objects from this MetaData.

connect (*bind, **kwargs*)

Bind this MetaData to an Engine.

Deprecated. Use `metadata.bind = <engine>` or `metadata.bind = <url>`.

bind A string, `URL`, `Engine` or `Connection` instance. If a string or `URL`, will be passed to `create_engine()` along with `\**kwargs` to produce the engine which to connect to. Otherwise connects directly to the given `Engine`.

create_all (*bind=None, tables=None, checkfirst=True*)

Create all tables stored in this metadata.

Conditional by default, will not attempt to recreate tables already present in the target database.

bind A `Connectable` used to access the database; if None, uses the existing bind on this `MetaData`, if any.

tables Optional list of `Table` objects, which is a subset of the total tables in the `MetaData` (others are ignored).

checkfirst Defaults to True, don't issue CREATEs for tables already present in the target database.

drop_all (*bind=None, tables=None, checkfirst=True*)

Drop all tables stored in this metadata.

Conditional by default, will not attempt to drop tables not present in the target database.

bind A `Connectable` used to access the database; if None, uses the existing bind on this `MetaData`, if any.

tables Optional list of `Table` objects, which is a subset of the total tables in the `MetaData` (others are ignored).

checkfirst Defaults to True, only issue DROPs for tables confirmed to be present in the target database.

is_bound ()

True if this MetaData is bound to an Engine or Connection.

reflect(*bind=None, schema=None, only=None*)
Load all available table definitions from the database.

Automatically creates `Table` entries in this `MetaData` for any table available in the database but not yet present in the `MetaData`. May be called multiple times to pick up tables recently added to the database, however no special action is taken if a table in this `MetaData` no longer exists in the database.

bind A `Connectable` used to access the database; if None, uses the existing bind on this `MetaData`, if any.

schema Optional, query and reflect tables from an alterate schema.

only Optional. Load only a sub-set of available named tables. May be specified as a sequence of names or a callable.
If a sequence of names is provided, only those tables will be reflected. An error is raised if a table is requested but not available. Named tables already present in this `MetaData` are ignored.
If a callable is provided, it will be used as a boolean predicate to filter the list of potential table names. The callable is called with a table name and this `MetaData` instance as positional arguments and should return a true value for any table to reflect.

remove(*table*)
Remove the given Table object from this MetaData.

sorted_tables
Returns a list of `Table` objects sorted in order of dependency.

table_iterator(*reverse=True, tables=None*)
Deprecated - use metadata.sorted_tables().

Deprecated. Use `metadata.sorted_tables`

class Table(*name, metadata, *args, **kwargs*)
Bases: `sqlalchemy.schema.SchemaItem`, `sqlalchemy.sql.expression.TableClause`

Represent a table in a database.

__init__(*name, metadata, *args, **kwargs*)
Construct a Table.

Parameters
- *name* – The name of this table as represented in the database.
  This property, along with the *schema*, indicates the *singleton identity* of this table in relation to its parent `MetaData`. Additional calls to `Table` with the same name, metadata, and schema name will return the same `Table` object.
  Names which contain no upper case characters will be treated as case insensitive names, and will not be quoted unless they are a reserved word. Names with any number of upper case characters will be quoted and sent exactly. Note that this behavior applies even for databases which standardize upper case names as case insensitive such as Oracle.
- *metadata* – a `MetaData` object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular `Connectable`.
- **args* – Additional positional arguments are used primarily to add the list of `Column` objects contained within this table. Similar to the style of a CREATE TABLE statement, other `SchemaItem` constructs may be added here, including `PrimaryKeyConstraint`, and `ForeignKeyConstraint`.
- *autoload* – Defaults to False: the Columns for this table should be reflected from the database. Usually there will be no Column objects in the constructor if this property is set.
- *autoload_with* – If autoload==True, this is an optional Engine or Connection instance to be used for the table reflection. If `None`, the underlying MetaData's bound connectable will be used.
- *include_columns* – A list of strings indicating a subset of columns to be loaded via the `autoload` operation; table columns who aren't present in this list will not be represented on the resulting `Table` object. Defaults to `None` which indicates all columns should be reflected.

- *info* – A dictionary which defaults to `{}`. A space to store application specific data. This must be a dictionary.
- *mustexist* – When `True`, indicates that this Table must already be present in the given `MetaData`' collection.
- *prefixes* – A list of strings to insert after CREATE in the CREATE TABLE statement. They will be separated by spaces.
- *quote* – Force quoting of this table's name on or off, corresponding to `True` or `False`. When left at its default of `None`, the column identifier will be quoted according to whether the name is case sensitive (identifiers with at least one upper case character are treated as case sensitive), or if it's a reserved word. This flag is only needed to force quoting of a reserved word which is not known by the SQLAlchemy dialect.
- *quote_schema* – same as 'quote' but applies to the schema identifier.
- *schema* – The *schema name* for this table, which is required if the table resides in a schema other than the default selected schema for the engine's database connection. Defaults to `None`.
- *useexisting* – When `True`, indicates that if this Table is already present in the given `MetaData`, apply further arguments within the constructor to the existing `Table`. If this flag is not set, an error is raised when the parameters of an existing `Table` are overwritten.

alias(*name=None*)
return an alias of this `FromClause`.

For table objects, this has the effect of the table being rendered as `tablename AS aliasname` in a SELECT statement. For select objects, the effect is that of creating a named subquery, i.e. `(select ...) AS aliasname`. The `alias()` method is the general way to create a "subquery" out of an existing SELECT.

The `name` parameter is optional, and if left blank an "anonymous" name will be generated at compile time, guaranteed to be unique against other anonymous constructs used in the same statement.

append_column(*column*)
Append a `Column` to this `Table`.

append_constraint(*constraint*)
Append a `Constraint` to this `Table`.

append_ddl_listener(*event, listener*)
Append a DDL event listener to this `Table`.

The `listener` callable will be triggered when this `Table` is created or dropped, either directly before or after the DDL is issued to the database. The listener may modify the Table, but may not abort the event itself.

Arguments are:

event One of `Table.ddl_events`; e.g. 'before-create', 'after-create', 'before-drop' or 'after-drop'.

listener A callable, invoked with three positional arguments:

event The event currently being handled

schema_item The `Table` object being created or dropped

bind The `Connection` bueing used for DDL execution.

Listeners are added to the Table's `ddl_listeners` attribute.

bind
Return the connectable associated with this SchemaItem.

c
Return the collection of Column objects contained by this FromClause.

columns
Return the collection of Column objects contained by this FromClause.

compare (*other*)
Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

compile (*bind=None, column_keys=None, compiler=None, dialect=None, inline=False*)
Compile this SQL expression.

The return value is a `Compiled` object. Calling *str()* or *unicode()* on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the *params* accessor.

Parameters
- *bind* – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- *column_keys* – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- *compiler* – A `Compiled` instance which will be used to compile this expression. This argument takes precedence over the *bind* and *dialect* arguments as well as this `ClauseElement`'s bound engine, if any.
- *dialect* – A `Dialect` instance frmo which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- *inline* – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.

correspond_on_equivalents (*column, equivalents*)
Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column (*column, require_embedded=False*)
Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common anscestor column.

Parameters
- *column* – the target `ColumnElement` to be matched
- *require_embedded* – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common anscestor with one of the exported columns of this `FromClause`.

count (*whereclause=None, **params*)

create (*bind=None, checkfirst=False*)
Issue a `CREATE` statement for this table.

See also `metadata.create_all()`.

delete (*whereclause=None, **kwargs*)
Generate a `delete()` construct.

drop (*bind=None, checkfirst=False*)
Issue a `DROP` statement for this table.

See also `metadata.drop_all()`.

execute (**multiparams, **params*)
Compile and execute this `ClauseElement`.

exists (*bind=None*)
Return True if this table exists.

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

get_children (*column_collections=True, schema_visitor=False, **kwargs*)

info

insert (*values=None, inline=False, **kwargs*)
Generate an `insert()` construct.

is_derived_from (*fromclause*)
Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

join (*right, onclause=None, isouter=False*)
return a join of this `FromClause` against another `FromClause`.

key

outerjoin (*right, onclause=None*)
return an outer join of this `FromClause` against another `FromClause`.

params (**optionaldict, **kwargs*)
Return a copy with `bindparam()` elments replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key

replace_selectable (*old, alias*)
replace all occurences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

scalar (**multiparams, **params*)
Compile and execute this `ClauseElement`, returning the result's scalar representation.

select (*whereclause=None, **params*)
return a SELECT of this `FromClause`.

self_group (*against=None*)

tometadata (*metadata, schema=None*)
Return a copy of this `Table` associated with a different `MetaData`.

unique_params (**optionaldict, **kwargs*)
Return a copy with `bindparam()` elments replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

update (*whereclause=None, values=None, inline=False, **kwargs*)
Generate an `update()` construct.

class **ThreadLocalMetaData** ()
Bases: `sqlalchemy.schema.MetaData`

A MetaData variant that presents a different `bind` in every thread.

Makes the `bind` property of the MetaData a thread-local value, allowing this collection of tables to be bound to different `Engine` implementations or connections in each thread.

The ThreadLocalMetaData starts off bound to None in each thread. Binds must be made explicitly by assigning to the `bind` property or using `connect()`. You can also re-bind dynamically multiple times per thread, just like a regular `MetaData`.

`__init__`()
Construct a ThreadLocalMetaData.

`bind`
The bound Engine or Connection for this thread.

This property may be assigned an Engine or Connection, or assigned a string or URL to automatically create a basic Engine for this bind with `create_engine()`.

`connect`(*bind, **kwargs*)
Bind to an Engine in the caller's thread.

Deprecated. Use `metadata.bind = <engine>` or `metadata.bind = <url>`.

bind A string, `URL`, `Engine` or `Connection` instance. If a string or `URL`, will be passed to `create_engine()` along with `\**kwargs` to produce the engine which to connect to. Otherwise connects directly to the given `Engine`.

`dispose`()
Dispose all bound engines, in all thread contexts.

`is_bound`()
True if there is a bind for this thread.

Constraints

class `CheckConstraint`(*sqltext, name=None, deferrable=None, initially=None*)
Bases: `sqlalchemy.schema.Constraint`

A table- or column-level CHECK constraint.

Can be included in the definition of a Table or Column.

`__init__`(*sqltext, name=None, deferrable=None, initially=None*)
Construct a CHECK constraint.

sqltext A string containing the constraint definition. Will be used verbatim.

name Optional, the in-database name of the constraint.

deferrable Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.

initially Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.

`copy`(***kw*)

class `Constraint`(*name=None, deferrable=None, initially=None*)
Bases: `sqlalchemy.schema.SchemaItem`

A table-level SQL constraint, such as a KEY.

Implements a hybrid of dict/setlike behavior with regards to the list of underying columns.

`__init__`(*name=None, deferrable=None, initially=None*)
Create a SQL constraint.

name Optional, the in-database name of this `Constraint`.

deferrable Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.

initially Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.

`contains_column`(*col*)

`copy`(***kw*)

`keys`()

class ForeignKey (*column, constraint=None, use_alter=False, name=None, onupdate=None, ondelete=None, deferrable=None, initially=None, link_to_name=False*)

Bases: `sqlalchemy.schema.SchemaItem`

Defines a column-level FOREIGN KEY constraint between two columns.

`ForeignKey` is specified as an argument to a `Column` object, e.g.:

```
t = Table("remote_table", metadata,
    Column("remote_id", ForeignKey("main_table.id"))
)
```

For a composite (multiple column) FOREIGN KEY, use a `ForeignKeyConstraint` object specified at the level of the `Table`.

Further examples of foreign key configuration are in *Defining Foreign Keys*.

__init__ (*column, constraint=None, use_alter=False, name=None, onupdate=None, ondelete=None, deferrable=None, initially=None, link_to_name=False*)

Construct a column-level FOREIGN KEY.

Parameters
- *column* – A single target column for the key relationship. A `Column` object or a column name as a string: `tablename.columnkey` or `schema.tablename.columnkey`. `columnkey` is the `key` which has been assigned to the column (defaults to the column name itself), unless `link_to_name` is `True` in which case the rendered name of the column is used.
- *constraint* – Optional. A parent `ForeignKeyConstraint` object. If not supplied, a `ForeignKeyConstraint` will be automatically created and added to the parent table.
- *name* – Optional string. An in-database name for the key if *constraint* is not provided.
- *onupdate* – Optional string. If set, emit ON UPDATE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- *ondelete* – Optional string. If set, emit ON DELETE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- *deferrable* – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- *initially* – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- *link_to_name* – if True, the string name given in `column` is the rendered name of the referenced column, not its locally assigned `key`.
- *use_alter* – If True, do not emit this key as part of the CREATE TABLE definition. Instead, use ALTER TABLE after table creation to add the key. Useful for circular dependencies.

copy (*schema=None*)

Produce a copy of this ForeignKey object.

get_referent (*table*)

Return the column in the given table referenced by this ForeignKey.

Returns None if this `ForeignKey` does not reference the given table.

references (*table*)

Return True if the given table is referenced by this ForeignKey.

target_fullname

class ForeignKeyConstraint (*columns, refcolumns, name=None, onupdate=None, ondelete=None, use_alter=False, deferrable=None, initially=None, link_to_name=False*)

Bases: `sqlalchemy.schema.Constraint`

A table-level FOREIGN KEY constraint.

Defines a single column or composite FOREIGN KEY ... REFERENCES constraint. For a no-frills, single column foreign key, adding a `ForeignKey` to the definition of a `Column` is a shorthand equivalent for an unnamed, single column `ForeignKeyConstraint`.

Examples of foreign key configuration are in *Defining Foreign Keys*.

`__init__`(*columns, refcolumns, name=None, onupdate=None, ondelete=None, use_alter=False, deferrable=None, initially=None, link_to_name=False*)

Construct a composite-capable FOREIGN KEY.

Parameters
- *columns* – A sequence of local column names. The named columns must be defined and present in the parent Table. The names should match the `key` given to each column (defaults to the name) unless `link_to_name` is True.
- *refcolumns* – A sequence of foreign column names or Column objects. The columns must all be located within the same Table.
- *name* – Optional, the in-database name of the key.
- *onupdate* – Optional string. If set, emit ON UPDATE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- *ondelete* – Optional string. If set, emit ON DELETE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- *deferrable* – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- *initially* – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- *link_to_name* – if True, the string name given in `column` is the rendered name of the referenced column, not its locally assigned `key`.
- *use_alter* – If True, do not emit this key as part of the CREATE TABLE definition. Instead, use ALTER TABLE after table creation to add the key. Useful for circular dependencies.

`append_element`(*col, refcol*)

`copy`(***kw*)

class **`Index`**(*name, *columns, **kwargs*)

Bases: `sqlalchemy.schema.SchemaItem`

A table-level INDEX.

Defines a composite (one or more column) INDEX. For a no-frills, single column index, adding `index=True` to the `Column` definition is a shorthand equivalent for an unnamed, single column Index.

`__init__`(*name, *columns, **kwargs*)

Construct an index object.

Arguments are:

name The name of the index

***columns** Columns to include in the index. All columns must belong to the same table, and no column may appear more than once.

****kwargs** Keyword arguments include:

unique Defaults to False: create a unique index.

postgres_where Defaults to None: create a partial index when using PostgreSQL

`append_column`(*column*)

`create`(*bind=None*)

`drop`(*bind=None*)

class **`PrimaryKeyConstraint`**(**columns, **kwargs*)

Bases: `sqlalchemy.schema.Constraint`

A table-level PRIMARY KEY constraint.

Defines a single column or composite PRIMARY KEY constraint. For a no-frills primary key, adding `primary_key=True` to one or more `Column` definitions is a shorthand equivalent for an unnamed single- or multiple-column PrimaryKeyConstraint.

`__init__`(**columns, **kwargs*)
Construct a composite-capable PRIMARY KEY.

***columns** A sequence of column names. All columns named must be defined and present within the parent Table.

name Optional, the in-database name of the key.

deferrable Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.

initially Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.

`add`(*col*)

`append_column`(*col*)

`copy`(***kw*)

`remove`(*col*)

`replace`(*col*)

class **`UniqueConstraint`**(**columns, **kwargs*)
Bases: `sqlalchemy.schema.Constraint`

A table-level UNIQUE constraint.

Defines a single column or composite UNIQUE constraint. For a no-frills, single column constraint, adding `unique=True` to the `Column` definition is a shorthand equivalent for an unnamed, single column UniqueConstraint.

`__init__`(**columns, **kwargs*)
Construct a UNIQUE constraint.

***columns** A sequence of column names. All columns named must be defined and present within the parent Table.

name Optional, the in-database name of the key.

deferrable Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.

initially Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.

`append_column`(*col*)

`copy`(***kw*)

Default Generators and Markers

class **`ColumnDefault`**(*arg, **kwargs*)
Bases: `sqlalchemy.schema.DefaultGenerator`

A plain default value on a column.

This could correspond to a constant, a callable function, or a SQL clause.

`__init__`(*arg, **kwargs*)

class **`DefaultClause`**(*arg, for_update=False*)
Bases: `sqlalchemy.schema.FetchedValue`

A DDL-specified DEFAULT column value.

class **`DefaultGenerator`**(*for_update=False, metadata=None*)
Bases: `sqlalchemy.schema.SchemaItem`

Base class for column *default* values.

__init__(*for_update=False, metadata=None*)

execute(*bind=None, **kwargs*)

class **FetchedValue**(*for_update=False*)
Bases: `object`

A default that takes effect on the database side.

__init__(*for_update=False*)

PassiveDefault
alias of `DefaultClause`

class **Sequence**(*name, start=None, increment=None, schema=None, optional=False, quote=None, **kwargs*)
Bases: `sqlalchemy.schema.DefaultGenerator`

Represents a named database sequence.

__init__(*name, start=None, increment=None, schema=None, optional=False, quote=None, **kwargs*)

create(*bind=None, checkfirst=True*)
Creates this sequence in the database.

drop(*bind=None, checkfirst=True*)
Drops this sequence from the database.

DDL

class **DDL**(*statement, on=None, context=None, bind=None*)
Bases: `object`

A literal DDL statement.

Specifies literal SQL DDL to be executed by the database. DDL objects can be attached to `Tables` or `MetaData` instances, conditionally executing SQL as part of the DDL lifecycle of those schema items. Basic templating support allows a single DDL instance to handle repetitive tasks for multiple tables.

Examples:

```
tbl = Table('users', metadata, Column('uid', Integer)) # ...
DDL('DROP TRIGGER users_trigger').execute_at('before-create', tbl)

spow = DDL('ALTER TABLE %(table)s SET secretpowers TRUE', on='somedb')
spow.execute_at('after-create', tbl)

drop_spow = DDL('ALTER TABLE users SET secretpowers FALSE')
connection.execute(drop_spow)
```

__init__(*statement, on=None, context=None, bind=None*)
Create a DDL statement.

statement A string or unicode string to be executed. Statements will be processed with Python's string formatting operator. See the `context` argument and the `execute_at` method.
A literal '%' in a statement must be escaped as '%%'.
SQL bind parameters are not available in DDL statements.

on Optional filtering criteria. May be a string or a callable predicate. If a string, it will be compared to the name of the executing database dialect:

```
DDL('something', on='postgres')
```

If a callable, it will be invoked with three positional arguments:

event The name of the event that has triggered this DDL, such as 'after-create' Will be None if the DDL is executed explicitly.

schema_item A SchemaItem instance, such as `Table` or `MetaData`. May be None if the DDL is executed explicitly.

connection The `Connection` being used for DDL execution

If the callable returns a true value, the DDL statement will be executed.

context Optional dictionary, defaults to None. These values will be available for use in string substitutions on the DDL statement.

bind Optional. A `Connectable`, used by default when `execute()` is invoked without a bind argument.

bind

An Engine or Connection to which this DDL is bound.

This property may be assigned an `Engine` or `Connection`, or assigned a string or URL to automatically create a basic `Engine` for this bind with `create_engine()`.

execute(*bind=None, schema_item=None*)

Execute this DDL immediately.

Executes the DDL statement in isolation using the supplied `Connectable` or `Connectable` assigned to the `.bind` property, if not supplied. If the DDL has a conditional `on` criteria, it will be invoked with None as the event.

bind Optional, an `Engine` or `Connection`. If not supplied, a valid `Connectable` must be present in the `.bind` property.

schema_item Optional, defaults to None. Will be passed to the `on` callable criteria, if any, and may provide string expansion data for the statement. See `execute_at` for more information.

execute_at(*event, schema_item*)

Link execution of this DDL to the DDL lifecycle of a SchemaItem.

Links this `DDL` to a `Table` or `MetaData` instance, executing it when that schema item is created or dropped. The DDL statement will be executed using the same Connection and transactional context as the Table create/drop itself. The `.bind` property of this statement is ignored.

event One of the events defined in the schema item's `.ddl_events`; e.g. 'before-create', 'after-create', 'before-drop' or 'after-drop'

schema_item A Table or MetaData instance

When operating on Table events, the following additional `statement` string substitions are available:

```
%(table)s  - the Table name, with any required quoting applied
%(schema)s - the schema name, with any required quoting applied
%(fullname)s - the Table name including schema, quoted if needed
```

The DDL's `context`, if any, will be combined with the standard substutions noted above. Keys present in the context will override the standard substitutions.

A DDL instance can be linked to any number of schema items. The statement subsitution support allows for DDL instances to be used in a template fashion.

`execute_at` builds on the `append_ddl_listener` interface of MetaDta and Table objects.

Caveat: Creating or dropping a Table in isolation will also trigger any DDL set to `execute_at` that Table's MetaData. This may change in a future release.

Internals

class **SchemaItem**()

Bases: `sqlalchemy.sql.visitors.Visitable`

Base class for items that define a database schema.

bind

Return the connectable associated with this SchemaItem.

get_children(***kwargs*)
used to allow SchemaVisitor access

info

class **SchemaVisitor**()
Bases: `sqlalchemy.sql.visitors.ClauseVisitor`

Define the visiting for `SchemaItem` objects.

8.1.5 Column and Data Types

SQLAlchemy provides abstractions for most common database data types, and a mechanism for specifying your own custom data types.

The methods and attributes of type objects are rarely used directly. Type objects are supplied to `Table` definitions and can be supplied as type hints to *functions* for occasions where the database driver returns an incorrect type.

```
>>> users = Table('users', metadata,
...               Column('id', Integer, primary_key=True)
...               Column('login', String(32))
...              )
```

SQLAlchemy will use the `Integer` and `String(32)` type information when issuing a `CREATE TABLE` statement and will use it again when reading back rows `SELECTed` from the database. Functions that accept a type (such as `Column()`) will typically accept a type class or instance; `Integer` is equivalent to `Integer()` with no construction arguments in this case.

Generic Types

Generic types specify a column that can read, write and store a particular type of Python data. SQLAlchemy will choose the best database column type available on the target database when issuing a `CREATE TABLE` statement. For complete control over which column type is emitted in `CREATE TABLE`, such as `VARCHAR` see SQL Standard Types and the other sections of this chapter.

class **String**(*length=None, convert_unicode=False, assert_unicode=None*)
Bases: `sqlalchemy.types.Concatenable`, `sqlalchemy.types.TypeEngine`

The base for all string and character types.

In SQL, corresponds to VARCHAR. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.)

The *length* field is usually required when the *String* type is used within a CREATE TABLE statement, as VARCHAR requires a length on most databases.

class **Unicode**(*length=None, **kwargs*)
Bases: `sqlalchemy.types.String`

A variable length Unicode string.

The `Unicode` type is a `String` which converts Python `unicode` objects (i.e., strings that are defined as `u'somevalue'`) into encoded bytestrings when passing the value to the database driver, and similarly decodes values from the database back into Python `unicode` objects.

When using the `Unicode` type, it is only appropriate to pass Python `unicode` objects, and not plain `str`. If a bytestring (`str`) is passed, a runtime warning is issued. If you notice your application raising these warnings but you're not sure where, the Python `warnings` filter can be used to turn these warnings into exceptions which will illustrate a stack trace:

```
import warnings
warnings.simplefilter('error')
```

Bytestrings sent to and received from the database are encoded using the dialect's `encoding`, which defaults to *utf-8*.

A synonym for String(length, convert_unicode=True, assert_unicode='warn').

class Text (*length=None, convert_unicode=False, assert_unicode=None*)

Bases: `sqlalchemy.types.String`

A variably sized string type.

In SQL, usually corresponds to CLOB or TEXT. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.)

class UnicodeText (*length=None, **kwargs*)

Bases: `sqlalchemy.types.Text`

A synonym for Text(convert_unicode=True, assert_unicode='warn').

class Integer (**args, **kwargs*)

Bases: `sqlalchemy.types.TypeEngine`

A type for `int` integers.

class SmallInteger (**args, **kwargs*)

Bases: `sqlalchemy.types.Integer`

A type for smaller `int` integers.

Typically generates a `SMALLINT` in DDL, and otherwise acts like a normal `Integer` on the Python side.

class Numeric (*precision=10, scale=2, asdecimal=True, length=None*)

Bases: `sqlalchemy.types.TypeEngine`

A type for fixed precision numbers.

Typically generates DECIMAL or NUMERIC. Returns `decimal.Decimal` objects by default.

class Float (*precision=10, asdecimal=False, **kwargs*)

Bases: `sqlalchemy.types.Numeric`

A type for `float` numbers.

class DateTime (*timezone=False*)

Bases: `sqlalchemy.types.TypeEngine`

A type for `datetime.datetime()` objects.

Date and time types return objects from the Python `datetime` module. Most DBAPIs have built in support for the datetime module, with the noted exception of SQLite. In the case of SQLite, date and time types are stored as strings which are then converted back to datetime objects when rows are returned.

class Date (**args, **kwargs*)

Bases: `sqlalchemy.types.TypeEngine`

A type for `datetime.date()` objects.

class Time (*timezone=False*)

Bases: `sqlalchemy.types.TypeEngine`

A type for `datetime.time()` objects.

class Interval ()

Bases: `sqlalchemy.types.TypeDecorator`

A type for `datetime.timedelta()` objects.

The Interval type deals with `datetime.timedelta` objects. In PostgreSQL, the native `INTERVAL` type is used; for others, the value is stored as a date which is relative to the "epoch" (Jan. 1, 1970).

class `Boolean` (**args, **kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

A bool datatype.

Boolean typically uses BOOLEAN or SMALLINT on the DDL side, and on the Python side deals in `True` or `False`.

class `Binary` (*length=None*)
Bases: `sqlalchemy.types.TypeEngine`

A type for binary byte data.

The Binary type generates BLOB or BYTEA when tables are created, and also converts incoming values using the `Binary` callable provided by each DB-API.

class `PickleType` (*protocol=2, pickler=None, mutable=True, comparator=None*)
Bases: `sqlalchemy.types.MutableType`, `sqlalchemy.types.TypeDecorator`

Holds Python objects.

PickleType builds upon the Binary type to apply Python's `pickle.dumps()` to incoming objects, and `pickle.loads()` on the way out, allowing any pickleable Python object to be stored as a serialized binary field.

SQL Standard Types

The SQL standard types always create database column types of the same name when `CREATE TABLE` is issued. Some types may not be supported on all databases.

class `INT` (**args, **kwargs*)
Bases: `sqlalchemy.types.Integer`

The SQL INT or INTEGER type.

`INTEGER`
alias of `INT`

class `CHAR` (*length=None, convert_unicode=False, assert_unicode=None*)
Bases: `sqlalchemy.types.String`

The SQL CHAR type.

class `VARCHAR` (*length=None, convert_unicode=False, assert_unicode=None*)
Bases: `sqlalchemy.types.String`

The SQL VARCHAR type.

class `NCHAR` (*length=None, **kwargs*)
Bases: `sqlalchemy.types.Unicode`

The SQL NCHAR type.

`TEXT`
alias of `Text`

class `FLOAT` (*precision=10, asdecimal=False, **kwargs*)
Bases: `sqlalchemy.types.Float`

The SQL FLOAT type.

class `NUMERIC` (*precision=10, scale=2, asdecimal=True, length=None*)
Bases: `sqlalchemy.types.Numeric`

The SQL NUMERIC type.

class DECIMAL (*precision=10, scale=2, asdecimal=True, length=None*)
Bases: `sqlalchemy.types.Numeric`

The SQL DECIMAL type.

class TIMESTAMP (*timezone=False*)
Bases: `sqlalchemy.types.DateTime`

The SQL TIMESTAMP type.

class DATETIME (*timezone=False*)
Bases: `sqlalchemy.types.DateTime`

The SQL DATETIME type.

class CLOB (*length=None, convert_unicode=False, assert_unicode=None*)
Bases: `sqlalchemy.types.Text`

The SQL CLOB type.

class BLOB (*length=None*)
Bases: `sqlalchemy.types.Binary`

The SQL BLOB type.

class BOOLEAN (**args, **kwargs*)
Bases: `sqlalchemy.types.Boolean`

The SQL BOOLEAN type.

class SMALLINT (**args, **kwargs*)
Bases: `sqlalchemy.types.SmallInteger`

The SQL SMALLINT type.

class DATE (**args, **kwargs*)
Bases: `sqlalchemy.types.Date`

The SQL DATE type.

class TIME (*timezone=False*)
Bases: `sqlalchemy.types.Time`

The SQL TIME type.

Vendor-Specific Types

Database-specific types are also available for import from each database's dialect module. See the *sqlalchemy.databases* reference for the database you're interested in.

For example, MySQL has a `BIGINTEGER` type and PostgreSQL has an `INET` type. To use these, import them from the module explicitly:

```
from sqlalchemy.databases.mysql import MSBigInteger, MSEnum

table = Table('foo', meta,
    Column('id', MSBigInteger),
    Column('enumerates', MSEnum('a', 'b', 'c'))
)
```

Or some PostgreSQL types:

```
from sqlalchemy.databases.postgres import PGInet, PGArray
```

```
table = Table('foo', meta,
    Column('ipaddress', PGInet),
    Column('elements', PGArray(str))
    )
```

Custom Types

User-defined types may be created to match special capabilities of a particular database or simply for implementing custom processing logic in Python.

The simplest method is implementing a `TypeDecorator`, a helper class that makes it easy to augment the bind parameter and result processing capabilities of one of the built in types.

To build a type object from scratch, subclass *:class:TypeEngine*.

class **TypeDecorator** (**args, **kwargs*)

Bases: `sqlalchemy.types.AbstractType`

Allows the creation of types which add additional functionality to an existing type.

Typical usage:

```
import sqlalchemy.types as types

class MyType(types.TypeDecorator):
    # Prefixes Unicode values with "PREFIX:" on the way in and
    # strips it off on the way out.

    impl = types.Unicode

    def process_bind_param(self, value, dialect):
        return "PREFIX:" + value

    def process_result_value(self, value, dialect):
        return value[7:]

    def copy(self):
        return MyType(self.impl.length)
```

The class-level "impl" variable is required, and can reference any TypeEngine class. Alternatively, the load_dialect_impl() method can be used to provide different type classes based on the dialect given; in this case, the "impl" variable can reference `TypeEngine` as a placeholder.

The reason that type behavior is modified using class decoration instead of subclassing is due to the way dialect specific types are used. Such as with the example above, when using the mysql dialect, the actual type in use will be a `sqlalchemy.databases.mysql.MSString` instance. `TypeDecorator` handles the mechanics of passing the values between user-defined `process_` methods and the current dialect-specific type in use.

__init__ (**args, **kwargs*)

adapt_operator (*op*)

Given an operator from the sqlalchemy.sql.operators package, translate it to a new operator based on the semantics of this type.

By default, returns the operator unchanged.

bind_processor (*dialect*)

compare_values (*x, y*)

copy ()

copy_value(*value*)

dialect_impl(*dialect, **kwargs*)

get_col_spec()

get_dbapi_type(*dbapi*)

is_mutable()

load_dialect_impl(*dialect*)
Loads the dialect-specific implementation of this type.

by default calls dialect.type_descriptor(self.impl), but can be overridden to provide different behavior.

process_bind_param(*value, dialect*)

process_result_value(*value, dialect*)

result_processor(*dialect*)

class **TypeEngine**(**args, **kwargs*)
Bases: `sqlalchemy.types.AbstractType`

Base for built-in types.

May be sub-classed to create entirely new types. Example:

```
import sqlalchemy.types as types

class MyType(types.TypeEngine):
    def __init__(self, precision = 8):
        self.precision = precision

    def get_col_spec(self):
        return "MYTYPE(%s)" % self.precision

    def bind_processor(self, dialect):
        def process(value):
            return value
        return process

    def result_processor(self, dialect):
        def process(value):
            return value
        return process
```

Once the type is made, it's immediately usable:

```
table = Table('foo', meta,
    Column('id', Integer, primary_key=True),
    Column('data', MyType(16))
    )
```

__init__(**args, **kwargs*)

adapt(*cls*)

adapt_operator(*op*)
Given an operator from the sqlalchemy.sql.operators package, translate it to a new operator based on the semantics of this type.

By default, returns the operator unchanged.

bind_processor(*dialect*)
Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

compare_values(*x, y*)
Compare two values for equality.

copy_value(*value*)

dialect_impl(*dialect, **kwargs*)

get_col_spec()
Return the DDL representation for this type.

get_dbapi_type(*dbapi*)
Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

get_search_list()
return a list of classes to test for a match when adapting this type to a dialect-specific type.

is_mutable()
Return True if the target Python type is 'mutable'.

This allows systems like the ORM to know if a column value can be considered 'not changed' by comparing the identity of objects alone.

Use the `MutableType` mixin or override this method to return True in custom types that hold mutable values such as `dict`, `list` and custom objects.

result_processor(*dialect*)
Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

class **AbstractType**(**args, **kwargs*)
Bases: `object`

__init__(**args, **kwargs*)

adapt_operator(*op*)
Given an operator from the sqlalchemy.sql.operators package, translate it to a new operator based on the semantics of this type.

By default, returns the operator unchanged.

bind_processor(*dialect*)
Defines a bind parameter processing function.

compare_values(*x, y*)
Compare two values for equality.

copy_value(*value*)

get_dbapi_type(*dbapi*)
Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

is_mutable()
Return True if the target Python type is 'mutable'.

This allows systems like the ORM to know if a column value can be considered 'not changed' by comparing the identity of objects alone.

Use the `MutableType` mixin or override this method to return True in custom types that hold mutable values such as `dict`, `list` and custom objects.

result_processor (*dialect*)
Defines a result-column processing function.

class **MutableType** ()
Bases: `object`

A mixin that marks a Type as holding a mutable object.

`copy_value()` and `compare_values()` should be customized as needed to match the needs of the object.

__init__ ()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

compare_values (*x, y*)
Compare *x* == *y*.

copy_value (*value*)
Unimplemented.

is_mutable ()
Return True, mutable.

class **Concatenable** ()
Bases: `object`

A mixin that marks a type as supporting 'concatenation', typically strings.

__init__ ()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

adapt_operator (*op*)
Converts an add operator to concat.

class **NullType** (**args, **kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

An unknown type.

NullTypes will stand in if `Table` reflection encounters a column data type unknown to SQLAlchemy. The resulting columns are nearly fully usable: the DB-API adapter will handle all translation to and from the database data type.

NullType does not have sufficient information to particpate in a `CREATE TABLE` statement and will raise an exception if encountered during a `create()` operation.

8.1.6 Interfaces

Interfaces and abstract types.

class **ConnectionProxy** ()
Allows interception of statement execution by Connections.

Either or both of the `execute()` and `cursor_execute()` may be implemented to intercept compiled statement and cursor level executions, e.g.:

```
class MyProxy(ConnectionProxy):
    def execute(self, conn, execute, clauseelement, *multiparams, **params):
        print "compiled statement:", clauseelement
        return execute(clauseelement, *multiparams, **params)

    def cursor_execute(self, execute, cursor, statement, parameters, context, executem
        print "raw statement:", statement
        return execute(cursor, statement, parameters, context)
```

The `execute` argument is a function that will fulfill the default execution behavior for the operation. The signature illustrated in the example should be used.

The proxy is installed into an `Engine` via the `proxy` argument:

```
e = create_engine('someurl://', proxy=MyProxy())
```

`cursor_execute`(*execute, cursor, statement, parameters, context, executemany*)
Intercept low-level cursor execute() events.

`execute`(*conn, execute, clauseelement, *multiparams, **params*)
Intercept high level execute() events.

class **`PoolListener`**()
Hooks into the lifecycle of connections in a `Pool`.

Usage:

```
class MyListener(PoolListener):
    def connect(self, dbapi_con, con_record):
        '''perform connect operations'''
    # etc.

# create a new pool with a listener
p = QueuePool(..., listeners=[MyListener()])

# add a listener after the fact
p.add_listener(MyListener())

# usage with create_engine()
e = create_engine("url://", listeners=[MyListener()])
```

All of the standard connection `Pool` types can accept event listeners for key connection lifecycle events: creation, pool check-out and check-in. There are no events fired when a connection closes.

For any given DB-API connection, there will be one `connect` event, *n* number of `checkout` events, and either *n* or *n - 1* `checkin` events. (If a `Connection` is detached from its pool via the `detach()` method, it won't be checked back in.)

These are low-level events for low-level objects: raw Python DB-API connections, without the conveniences of the SQLAlchemy `Connection` wrapper, `Dialect` services or `ClauseElement` execution. If you execute SQL through the connection, explicitly closing all cursors and other resources is recommended.

Events also receive a `_ConnectionRecord`, a long-lived internal `Pool` object that basically represents a "slot" in the connection pool. `_ConnectionRecord` objects have one public attribute of note: `info`, a dictionary whose contents are scoped to the lifetime of the DB-API connection managed by the record. You can use this shared storage area however you like.

There is no need to subclass `PoolListener` to handle events. Any class that implements one or more of these methods can be used as a pool listener. The `Pool` will inspect the methods provided by a listener object and add the listener to one or more internal event queues based on its capabilities. In terms of efficiency and function call overhead, you're much better off only providing implementations for the hooks you'll be using.

`checkin`(*dbapi_con, con_record*)
Called when a connection returns to the pool.

Note that the connection may be closed, and may be None if the connection has been invalidated. `checkin` will not be called for detached connections. (They do not return to the pool.)

dbapi_con A raw DB-API connection

con_record The `_ConnectionRecord` that persistently manages the connection

checkout (*dbapi_con, con_record, con_proxy*)
Called when a connection is retrieved from the Pool.

dbapi_con A raw DB-API connection

con_record The `_ConnectionRecord` that persistently manages the connection

con_proxy The `_ConnectionFairy` which manages the connection for the span of the current checkout.

If you raise an `exc.DisconnectionError`, the current connection will be disposed and a fresh connection retrieved. Processing of all checkout listeners will abort and restart using the new connection.

connect (*dbapi_con, con_record*)
Called once for each new DB-API connection or Pool's `creator()`.

dbapi_con A newly connected raw DB-API connection (not a SQLAlchemy `Connection` wrapper).

con_record The `_ConnectionRecord` that persistently manages the connection

8.2 sqlalchemy.orm

8.2.1 Class Mapping

Defining Mappings

Python classes are mapped to the database using the `mapper()` function.

mapper (*class_, local_table=None, *args, **params*)
Return a new `Mapper` object.

class_ The class to be mapped.

local_table The table to which the class is mapped, or None if this mapper inherits from another mapper using concrete table inheritance.

always_refresh If True, all query operations for this mapped class will overwrite all data within object instances that already exist within the session, erasing any in-memory changes with whatever information was loaded from the database. Usage of this flag is highly discouraged; as an alternative, see the method *populate_existing()* on `Query`.

allow_null_pks Indicates that composite primary keys where one or more (but not all) columns contain NULL is a valid primary key. Primary keys which contain NULL values usually indicate that a result row does not contain an entity and should be skipped.

batch Indicates that save operations of multiple entities can be batched together for efficiency. setting to False indicates that an instance will be fully saved before saving the next instance, which includes inserting/updating all table rows corresponding to the entity as well as calling all `MapperExtension` methods corresponding to the save operation.

column_prefix A string which will be prepended to the *key* name of all Columns when creating column-based properties from the given Table. Does not affect explicitly specified column-based properties

concrete If True, indicates this mapper should use concrete table inheritance with its parent mapper.

extension A `MapperExtension` instance or list of `MapperExtension` instances which will be applied to all operations by this `Mapper`.

inherits Another `Mapper` for which this `Mapper` will have an inheritance relationship with.

inherit_condition For joined table inheritance, a SQL expression (constructed `ClauseElement`) which will define how the two tables are joined; defaults to a natural join between the two tables.

inherit_foreign_keys when inherit_condition is used and the condition contains no ForeignKey columns, specify the "foreign" columns of the join condition in this list. else leave as None.

order_by A single `Column` or list of `Columns` for which selection operations should use as the default ordering for entities. Defaults to the OID/ROWID of the table if any, or the first primary key column of the table.

non_primary Construct a `Mapper` that will define only the selection of instances, not their persistence. Any number of non_primary mappers may be created for a particular class.

polymorphic_on Used with mappers in an inheritance relationship, a `Column` which will identify the class/mapper combination to be used with a particular row. Requires the `polymorphic_identity` value to be set for all mappers in the inheritance hierarchy. The column specified by `polymorphic_on` is usually a column that resides directly within the base mapper's mapped table; alternatively, it may be a column that is only present within the <selectable> portion of the `with_polymorphic` argument.

_polymorphic_map Used internally to propagate the full map of polymorphic identifiers to surrogate mappers.

polymorphic_identity A value which will be stored in the Column denoted by polymorphic_on, corresponding to the *class identity* of this mapper.

polymorphic_fetch Deprecated. Unloaded columns load as deferred in all cases; loading can be controlled using the "with_polymorphic" option.

properties A dictionary mapping the string names of object attributes to `MapperProperty` instances, which define the persistence behavior of that attribute. Note that the columns in the mapped table are automatically converted into `ColumnProperty` instances based on the *key* property of each `Column` (although they can be overridden using this dictionary).

include_properties An inclusive list of properties to map. Columns present in the mapped table but not present in this list will not be automatically converted into properties.

exclude_properties A list of properties not to map. Columns present in the mapped table and present in this list will not be automatically converted into properties. Note that neither this option nor include_properties will allow an end-run around Python inheritance. If mapped class `B` inherits from mapped class `A`, no combination of includes or excludes will allow `B` to have fewer properties than its superclass, `A`.

primary_key A list of `Column` objects which define the *primary key* to be used against this mapper's selectable unit. This is normally simply the primary key of the *local_table*, but can be overridden here.

with_polymorphic A tuple in the form `(<classes>, <selectable>)` indicating the default style of "polymorphic" loading, that is, which tables are queried at once. <classes> is any single or list of mappers and/or classes indicating the inherited classes that should be loaded at once. The special value `'*'` may be used to indicate all descending classes should be loaded immediately. The second tuple argument <selectable> indicates a selectable that will be used to query for multiple classes. Normally, it is left as None, in which case this mapper will form an outer join from the base mapper's table to that of all desired sub-mappers. When specified, it provides the selectable to be used for polymorphic loading. When with_polymorphic includes mappers which load from a "concrete" inheriting table, the <selectable> argument is required, since it usually requires more complex UNION queries.

select_table Deprecated. Synonymous with `with_polymorphic=('*', <selectable>)`.

version_id_col A `Column` which must have an integer type that will be used to keep a running *version id* of mapped entities in the database. this is used during save operations to ensure that no other thread or process has updated the instance during the lifetime of the entity, else a `ConcurrentModificationError` exception is thrown.

Mapper Properties

A basic mapping of a class will simply make the columns of the database table or selectable available as attributes on the class. **Mapper properties** allow you to customize and add additional properties to your classes, for example making the results one-to-many join available as a Python list of `related` objects.

Mapper properties are most commonly included in the `mapper()` call:

```
mapper(Parent, properties={
   'children': relation(Children)
}
```

backref(*name, **kwargs*)
: Create a BackRef object with explicit arguments, which are the same arguments one can send to `relation()`.

  Used with the *backref* keyword argument to `relation()` in place of a string argument.

column_property(**args, **kwargs*)
: Provide a column-level property for use with a Mapper.

  Column-based properties can normally be applied to the mapper's `properties` dictionary using the `schema.Column` element directly. Use this function when the given column is not directly present within the mapper's selectable; examples include SQL expressions, functions, and scalar SELECT queries.

  Columns that aren't present in the mapper's selectable won't be persisted by the mapper and are effectively "read-only" attributes.

  ***cols** list of Column objects to be mapped.

  comparator_factory a class which extends `sqlalchemy.orm.properties.ColumnProperty.Comparator` which provides custom SQL clause generation for comparison operations.

  group a group name for this property when marked as deferred.

  deferred when True, the column property is "deferred", meaning that it does not load immediately, and is instead loaded when the attribute is first accessed on an instance. See also `deferred()`.

  extension an `AttributeExtension` instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class. These listeners will receive append and set events before the operation proceeds, and may be used to halt (via exception throw) or change the value used in the operation.

comparable_property(*comparator_factory, descriptor=None*)
: Provide query semantics for an unmanaged attribute.

  Allows a regular Python @property (descriptor) to be used in Queries and SQL constructs like a managed attribute. comparable_property wraps a descriptor with a proxy that directs operator overrides such as == (__eq__) to the supplied comparator but proxies everything else through to the original descriptor:

```
class MyClass(object):
    @property
    def myprop(self):
        return 'foo'

class MyComparator(sqlalchemy.orm.interfaces.PropComparator):
    def __eq__(self, other):
        ....

mapper(MyClass, mytable, properties=dict(
         'myprop': comparable_property(MyComparator)))
```

  Used with the `properties` dictionary sent to `mapper()`.

  comparator_factory A PropComparator subclass or factory that defines operator behavior for this property.

  descriptor Optional when used in a `properties={}` declaration. The Python descriptor or property to layer comparison behavior on top of.

  The like-named descriptor will be automatically retreived from the mapped class if left blank in a `properties` declaration.

composite(*class_, *cols, **kwargs*)
Return a composite column-based property for use with a Mapper.

This is very much like a column-based property except the given class is used to represent "composite" values composed of one or more columns.

The class must implement a constructor with positional arguments matching the order of columns supplied here, as well as a __composite_values__() method which returns values in the same order.

A simple example is representing separate two columns in a table as a single, first-class "Point" object:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __composite_values__(self):
        return self.x, self.y
    def __eq__(self, other):
        return other is not None and self.x == other.x and self.y == other.y

# and then in the mapping:
... composite(Point, mytable.c.x, mytable.c.y) ...
```

The composite object may have its attributes populated based on the names of the mapped columns. To override the way internal state is set, additionally implement `__set_composite_values__`:

```
class Point(object):
    def __init__(self, x, y):
        self.some_x = x
        self.some_y = y
    def __composite_values__(self):
        return self.some_x, self.some_y
    def __set_composite_values__(self, x, y):
        self.some_x = x
        self.some_y = y
    def __eq__(self, other):
        return other is not None and self.some_x == other.x and self.some_y == other.y
```

Arguments are:

class_ The "composite type" class.

***cols** List of Column objects to be mapped.

group A group name for this property when marked as deferred.

deferred When True, the column property is "deferred", meaning that it does not load immediately, and is instead loaded when the attribute is first accessed on an instance. See also `deferred()`.

comparator_factory a class which extends `sqlalchemy.orm.properties.CompositeProperty.Comparator` which provides custom SQL clause generation for comparison operations.

extension an `AttributeExtension` instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class. These listeners will receive append and set events before the operation proceeds, and may be used to halt (via exception throw) or change the value used in the operation.

deferred(**columns, **kwargs*)
Return a `DeferredColumnProperty`, which indicates this object attributes should only be loaded from its corresponding table column when first accessed.

Used with the *properties* dictionary sent to `mapper()`.

dynamic_loader(*argument, secondary=None, primaryjoin=None, secondaryjoin=None, foreign_keys=None, backref=None, post_update=False, cascade=False, remote_side=None, enable_typechecks=True, passive_deletes=False, order_by=None, comparator_factory=None, query_class=None*)

Construct a dynamically-loading mapper property.

This property is similar to `relation()`, except read operations return an active `Query` object which reads from the database when accessed. Items may be appended to the attribute via `append()`, or removed via `remove()`; changes will be persisted to the database during a `Sesion.flush()`. However, no other Python list or collection mutation operations are available.

A subset of arguments available to `relation()` are available here.

Parameters

- *argument* – a class or `Mapper` instance, representing the target of the relation.
- *secondary* – for a many-to-many relationship, specifies the intermediary table. The *secondary* keyword argument should generally only be used for a table that is not otherwise expressed in any class mapping. In particular, using the Association Object Pattern is generally mutually exclusive with the use of the *secondary* keyword argument.
- *query_class* – Optional, a custom Query subclass to be used as the basis for dynamic collection.

relation(*argument, secondary=None, **kwargs*)

Provide a relationship of a primary Mapper to a secondary Mapper.

This corresponds to a parent-child or associative table relationship. The constructed class is an instance of `RelationProperty`.

A typical `relation()`:

```
mapper(Parent, properties={
  'children': relation(Children)
})
```

Parameters

- *argument* – a class or `Mapper` instance, representing the target of the relation.
- *secondary* – for a many-to-many relationship, specifies the intermediary table. The *secondary* keyword argument should generally only be used for a table that is not otherwise expressed in any class mapping. In particular, using the Association Object Pattern is generally mutually exclusive with the use of the *secondary* keyword argument.
- *backref* – indicates the string name of a property to be placed on the related mapper's class that will handle this relationship in the other direction. The other property will be created automatically when the mappers are configured. Can also be passed as a `backref()` object to control the configuration of the new relation.
- *back_populates* – Takes a string name and has the same meaning as `backref`, except the complementing property is **not** created automatically, and instead must be configured explicitly on the other mapper. The complementing property should also indicate `back_populates` to this relation to ensure proper functioning.
- *cascade* – a comma-separated list of cascade rules which determines how Session operations should be "cascaded" from parent to child. This defaults to `False`, which means the default cascade should be used. The default value is `"save-update, merge"`.
  Available cascades are:

  `save-update` - cascade the "add()" operation (formerly known as save() and update())
  `merge` - cascade the "merge()" operation
  `expunge` - cascade the "expunge()" operation
  `delete` - cascade the "delete()" operation

`delete-orphan` - if an item of the child's type with no parent is detected, mark it for deletion. Note that this option prevents a pending item of the child's class from being persisted without a parent present.
`refresh-expire` - cascade the expire() and refresh() operations
`all` - shorthand for "save-update,merge, refresh-expire, expunge, delete"

- *collection_class* – a class or callable that returns a new list-holding object. will be used in place of a plain list for storing elements.
- *comparator_factory* – a class which extends `RelationProperty.Comparator` which provides custom SQL clause generation for comparison operations.
- *extension* – an `AttributeExtension` instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class. These listeners will receive append and set events before the operation proceeds, and may be used to halt (via exception throw) or change the value used in the operation.
- *foreign_keys* – a list of columns which are to be used as "foreign key" columns. this parameter should be used in conjunction with explicit `primaryjoin` and `secondaryjoin` (if needed) arguments, and the columns within the `foreign_keys` list should be present within those join conditions. Normally, `relation()` will inspect the columns within the join conditions to determine which columns are the "foreign key" columns, based on information in the `Table` metadata. Use this argument when no ForeignKey's are present in the join condition, or to override the table-defined foreign keys.
- *join_depth* – when non-`None`, an integer value indicating how many levels deep eagerload joins should be constructed on a self-referring or cyclical relationship. The number counts how many times the same Mapper shall be present in the loading condition along a particular join branch. When left at its default of `None`, eager loads will automatically stop chaining joins when they encounter a mapper which is already higher up in the chain.
- *lazy=(True|False|None|'dynamic')* – specifies how the related items should be loaded. Values include:

  True - items should be loaded lazily when the property is first accessed.

  False - items should be loaded "eagerly" in the same query as that of the parent, using a JOIN or LEFT OUTER JOIN.

  None - no loading should occur at any time. This is to support "write-only" attributes, or attributes which are populated in some manner specific to the application.

  'dynamic' - a `DynaLoader` will be attached, which returns a `Query` object for all read operations. The dynamic- collection supports only `append()` and `remove()` for write operations; changes to the dynamic property will not be visible until the data is flushed to the database.
- *order_by* – indicates the ordering that should be applied when loading these items.
- *passive_deletes=False* – Indicates loading behavior during delete operations.

  A value of True indicates that unloaded child items should not be loaded during a delete operation on the parent. Normally, when a parent item is deleted, all child items are loaded so that they can either be marked as deleted, or have their foreign key to the parent set to NULL. Marking this flag as True usually implies an ON DELETE <CASCADE|SET NULL> rule is in place which will handle updating/deleting child rows on the database side.

  Additionally, setting the flag to the string value 'all' will disable the "nulling out" of the child foreign keys, when there is no delete or delete-orphan cascade enabled. This is typically used when a triggering or error raise scenario is in place on the database side. Note that the foreign key attributes on in-session child objects will not be changed after a flush occurs so this is a very special use-case setting.
- *passive_updates=True* – Indicates loading and INSERT/UPDATE/DELETE behavior when the source of a foreign key value changes (i.e. an "on update" cascade), which are typically the primary key columns of the source row.

When True, it is assumed that ON UPDATE CASCADE is configured on the foreign key in the database, and that the database will handle propagation of an UPDATE from a source column to dependent rows. Note that with databases which enforce referential integrity (i.e. Postgres, MySQL with InnoDB tables), ON UPDATE CASCADE is required for this operation. The relation() will update the value of the attribute on related items which are locally present in the session during a flush.

When False, it is assumed that the database does not enforce referential integrity and will not be issuing its own CASCADE operation for an update. The relation() will issue the appropriate UPDATE statements to the database in response to the change of a referenced key, and items locally present in the session during a flush will also be refreshed.

This flag should probably be set to False if primary key changes are expected and the database in use doesn't support CASCADE (i.e. SQLite, MySQL MyISAM tables).

- *post_update* – this indicates that the relationship should be handled by a second UPDATE statement after an INSERT or before a DELETE. Currently, it also will issue an UPDATE after the instance was UPDATEd as well, although this technically should be improved. This flag is used to handle saving bi-directional dependencies between two individual rows (i.e. each row references the other), where it would otherwise be impossible to INSERT or DELETE both rows fully since one row exists before the other. Use this flag when a particular mapping arrangement will incur two rows that are dependent on each other, such as a table that has a one-to-many relationship to a set of child rows, and also has a column that references a single child row within that list (i.e. both tables contain a foreign key to each other). If a `flush()` operation returns an error that a "cyclical dependency" was detected, this is a cue that you might want to use `post_update` to "break" the cycle.
- *primaryjoin* – a ClauseElement that will be used as the primary join of this child object against the parent object, or in a many-to-many relationship the join of the primary object to the association table. By default, this value is computed based on the foreign key relationships of the parent and child tables (or association table).
- *remote_side* – used for self-referential relationships, indicates the column or list of columns that form the "remote side" of the relationship.
- *secondaryjoin* – a ClauseElement that will be used as the join of an association table to the child object. By default, this value is computed based on the foreign key relationships of the association and child tables.
- *single_parent=(True|False)* – when True, installs a validator which will prevent objects from being associated with more than one parent at a time. This is used for many-to-one or many-to-many relationships that should be treated either as one-to-one or one-to-many. Its usage is optional unless delete-orphan cascade is also set on this relation(), in which case its required (new in 0.5.2).
- *uselist=(True|False)* – a boolean that indicates if this property should be loaded as a list or a scalar. In most cases, this value is determined automatically by `relation()`, based on the type and direction of the relationship - one to many forms a list, many to one forms a scalar, many to many is a list. If a scalar is desired where normally a list would be present, such as a bi-directional one-to-one relationship, set uselist to False.
- *viewonly=False* – when set to True, the relation is used only for loading objects within the relationship, and has no effect on the unit-of-work flush process. Relationships with viewonly can specify any kind of join conditions to provide additional views of related objects onto a parent object. Note that the functionality of a viewonly relationship has its limits - complicated join conditions may not compile into eager or lazy loaders properly. If this is the case, use an alternative method.

`synonym`(*name, map_column=False, descriptor=None, comparator_factory=None, proxy=False*)

Set up *name* as a synonym to another mapped property.

Used with the `properties` dictionary sent to `mapper()`.

Any existing attributes on the class which map the key name sent to the `properties` dictionary will be used by the synonym to provide instance-attribute behavior (that is, any Python property object, provided by the `property` builtin or providing a `__get__()`, `__set__()` and `__del__()` method). If no name exists for the key, the `synonym()` creates a default getter/setter object automatically and applies it to the class.

name refers to the name of the existing mapped property, which can be any other `MapperProperty` including column-based properties and relations.

If *map_column* is `True`, an additional `ColumnProperty` is created on the mapper automatically, using the synonym's name as the keyname of the property, and the keyname of this `synonym()` as the name of the column to map. For example, if a table has a column named `status`:

```
class MyClass(object):
    def _get_status(self):
        return self._status
    def _set_status(self, value):
        self._status = value
    status = property(_get_status, _set_status)

mapper(MyClass, sometable, properties={
    "status":synonym("_status", map_column=True)
})
```

The column named `status` will be mapped to the attribute named `_status`, and the `status` attribute on `MyClass` will be used to proxy access to the column-based attribute.

The *proxy* keyword argument is deprecated and currently does nothing; synonyms now always establish an attribute getter/setter function if one is not already available.

Decorators

`reconstructor`(*fn*)

Decorate a method as the 'reconstructor' hook.

Designates a method as the "reconstructor", an `__init__`-like method that will be called by the ORM after the instance has been loaded from the database or otherwise reconstituted.

The reconstructor will be invoked with no arguments. Scalar (non-collection) database-mapped attributes of the instance will be available for use within the function. Eagerly-loaded collections are generally not yet available and will usually only contain the first element. ORM state changes made to objects at this stage will not be recorded for the next flush() operation, so the activity within a reconstructor should be conservative.

`validates`(**names*)

Decorate a method as a 'validator' for one or more named properties.

Designates a method as a validator, a method which receives the name of the attribute as well as a value to be assigned, or in the case of a collection to be added to the collection. The function can then raise validation exceptions to halt the process from continuing, or can modify or replace the value before proceeding. The function should otherwise return the given value.

Utilities

`object_mapper`(*instance*)

Given an object, return the primary Mapper associated with the object instance.

Raises UnmappedInstanceError if no mapping is configured.

class_mapper(*class_, compile=True*)
Given a class, return the primary Mapper associated with the key.

Raises UnmappedClassError if no mapping is configured.

compile_mappers()
Compile all mappers that have been defined.

This is equivalent to calling `compile()` on any individual mapper.

clear_mappers()
Remove all mappers that have been created thus far.

The mapped classes will return to their initial "unmapped" state and can be re-mapped with new mappers.

Attribute Utilities

del_attribute(*instance, key*)
Delete the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

get_attribute(*instance, key*)
Get the value of an attribute, firing any callables required.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to make usage of attribute state as understood by SQLAlchemy.

get_history(*obj, key, **kwargs*)
Return a History record for the given object and attribute key.

obj is an instrumented object instance. An InstanceState is accepted directly for backwards compatibility but this usage is deprecated.

init_collection(*obj, key*)
Initialize a collection attribute and return the collection adapter.

This function is used to provide direct access to collection internals for a previously unloaded attribute. e.g.:

```
collection_adapter = init_collection(someobject, 'elements')
for elem in values:
    collection_adapter.append_without_event(elem)
```

For an easier way to do the above, see `set_committed_value()`.

obj is an instrumented object instance. An InstanceState is accepted directly for backwards compatibility but this usage is deprecated.

instance_state()
Return the `InstanceState` for a given object.

is_instrumented(*instance, key*)
Return True if the given attribute on the given instance is instrumented by the attributes package.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required.

manager_of_class()
Return the `ClassManager` for a given class.

set_attribute(*instance, key, value*)
Set the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

set_committed_value(*instance, key, value*)
Set the value of an attribute with no history events.

Cancels any previous history present. The value should be a scalar value for scalar-holding attributes, or an iterable for any collection-holding attribute.

This is the same underlying method used when a lazy loader fires off and loads additional data from the database. In particular, this method can be used by application code which has loaded additional attributes or collections through separate queries, which can then be attached to an instance as though it were part of its original loaded state.

Internals

class **Mapper**(*class_, local_table, properties=None, primary_key=None, non_primary=False, inherits=None, inherit_condition=None, inherit_foreign_keys=None, extension=None, order_by=False, always_refresh=False, version_id_col=None, polymorphic_on=None, _polymorphic_map=None, polymorphic_identity=None, polymorphic_fetch=None, concrete=False, select_table=None, with_polymorphic=None, allow_null_pks=False, batch=True, column_prefix=None, include_properties=None, exclude_properties=None, eager_defaults=False*)
Define the correlation of class attributes to database table columns.

Instances of this class should be constructed via the `mapper()` function.

__init__(*class_, local_table, properties=None, primary_key=None, non_primary=False, inherits=None, inherit_condition=None, inherit_foreign_keys=None, extension=None, order_by=False, always_refresh=False, version_id_col=None, polymorphic_on=None, _polymorphic_map=None, polymorphic_identity=None, polymorphic_fetch=None, concrete=False, select_table=None, with_polymorphic=None, allow_null_pks=False, batch=True, column_prefix=None, include_properties=None, exclude_properties=None, eager_defaults=False*)
Construct a new mapper.

Mappers are normally constructed via the `mapper()` function. See for details.

add_properties(*dict_of_properties*)
Add the given dictionary of properties to this mapper, using *add_property*.

add_property(*key, prop*)
Add an individual MapperProperty to this mapper.

If the mapper has not been compiled yet, just adds the property to the initial properties dictionary sent to the constructor. If this Mapper has already been compiled, then the given MapperProperty is compiled immediately.

cascade_iterator(*type_, state, halt_on=None*)
Iterate each element and its mapper in an object graph, for all relations that meet the given cascade rule.

type_: The name of the cascade rule (i.e. save-update, delete, etc.)

state: The lead InstanceState. child items will be processed per the relations defined for this object's mapper.

the return value are object instances; this provides a strong reference so that they don't fall out of scope immediately.

common_parent(*other*)
Return true if the given mapper shares a common inherited parent as this mapper.

compile()
: Compile this mapper and all other non-compiled mappers.

  This method checks the local compiled status as well as for any new mappers that have been defined, and is safe to call repeatedly.

get_property(*key, resolve_synonyms=False, raiseerr=True*)
: return a MapperProperty associated with the given key.

identity_key_from_instance(*instance*)
: Return the identity key for the given instance, based on its primary key attributes.

  This value is typically also found on the instance state under the attribute name *key*.

identity_key_from_primary_key(*primary_key*)
: Return an identity-map key for use in storing/retrieving an item from an identity map.

  primary_key A list of values indicating the identifier.

identity_key_from_row(*row, adapter=None*)
: Return an identity-map key for use in storing/retrieving an item from the identity map.

  row A `sqlalchemy.engine.base.RowProxy` instance or a dictionary corresponding result-set `ColumnElement` instances to their values within a row.

isa(*other*)
: Return True if the this mapper inherits from the given mapper.

iterate_properties
: return an iterator of all MapperProperty objects.

polymorphic_iterator()
: Iterate through the collection including this mapper and all descendant mappers.

  This includes not just the immediately inheriting mappers but all their inheriting mappers as well.

  To iterate through an entire hierarchy, use `mapper.base_mapper.polymorphic_iterator()`.

primary_key_from_instance(*instance*)
: Return the list of primary key values for the given instance.

primary_mapper()
: Return the primary mapper corresponding to this mapper's class key (class).

8.2.2 Collection Mapping

This is an in-depth discussion of collection mechanics. For simple examples, see *Alternate Collection Implementations*. Support for collections of mapped entities.

The collections package supplies the machinery used to inform the ORM of collection membership changes. An instrumentation via decoration approach is used, allowing arbitrary types (including built-ins) to be used as entity collections without requiring inheritance from a base class.

Instrumentation decoration relays membership change events to the `InstrumentedCollectionAttribute` that is currently managing the collection. The decorators observe function call arguments and return values, tracking entities entering or leaving the collection. Two decorator approaches are provided. One is a bundle of generic decorators that map function arguments and return values to events:

```
from sqlalchemy.orm.collections import collection
class MyClass(object):
    # ...

    @collection.adds(1)
    def store(self, item):
        self.data.append(item)
    @collection.removes_return()
    def pop(self):
        return self.data.pop()
```

The second approach is a bundle of targeted decorators that wrap appropriate append and remove notifiers around the mutation methods present in the standard Python `list`, `set` and `dict` interfaces. These could be specified in terms of generic decorator recipes, but are instead hand-tooled for increased efficiency. The targeted decorators occasionally implement adapter-like behavior, such as mapping bulk-set methods (`extend`, `update`, `__setslice__`, etc.) into the series of atomic mutation events that the ORM requires.

The targeted decorators are used internally for automatic instrumentation of entity collection classes. Every collection class goes through a transformation process roughly like so:

1. If the class is a built-in, substitute a trivial sub-class
2. Is this class already instrumented?
3. Add in generic decorators
4. Sniff out the collection interface through duck-typing
5. Add targeted decoration to any undecorated interface method

This process modifies the class at runtime, decorating methods and adding some bookkeeping properties. This isn't possible (or desirable) for built-in classes like `list`, so trivial sub-classes are substituted to hold decoration:

```
class InstrumentedList(list):
    pass
```

Collection classes can be specified in `relation(collection_class=)` as types or a function that returns an instance. Collection classes are inspected and instrumented during the mapper compilation phase. The collection_class callable will be executed once to produce a specimen instance, and the type of that specimen will be instrumented. Functions that return built-in types like `lists` will be adapted to produce instrumented instances.

When extending a known type like `list`, additional decorations are not generally not needed. Odds are, the extension method will delegate to a method that's already instrumented. For example:

```
class QueueIsh(list):
   def push(self, item):
       self.append(item)
   def shift(self):
       return self.pop(0)
```

There's no need to decorate these methods. `append` and `pop` are already instrumented as part of the `list` interface. Decorating them would fire duplicate events, which should be avoided.

The targeted decoration tries not to rely on other methods in the underlying collection class, but some are unavoidable. Many depend on 'read' methods being present to properly instrument a 'write', for example, `__setitem__` needs `__getitem__`. "Bulk" methods like `update` and `extend` may also reimplemented in terms of atomic appends and removes, so the `extend` decoration will actually perform many `append` operations and not call the underlying method at all.

Tight control over bulk operation and the firing of events is also possible by implementing the instrumentation internally in your methods. The basic instrumentation package works under the general assumption that collection mutation will not raise unusual exceptions. If you want to closely orchestrate append and remove events with exception management, internal instrumentation may be the answer. Within your method, `collection_adapter(self)` will retrieve an object that you can use for explicit control over triggering append and remove events.

The owning object and InstrumentedCollectionAttribute are also reachable through the adapter, allowing for some very sophisticated behavior.

`attribute_mapped_collection`(*attr_name*)

A dictionary-based collection type with attribute-based keying.

Returns a MappedCollection factory with a keying based on the 'attr_name' attribute of entities in the collection.

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

class **`collection`**()

Decorators for entity collection classes.

The decorators fall into two groups: annotations and interception recipes.

The annotating decorators (appender, remover, iterator, internally_instrumented, on_link) indicate the method's purpose and take no arguments. They are not written with parens:

```
@collection.appender
def append(self, append): ...
```

The recipe decorators all require parens, even those that take no arguments:

```
@collection.adds('entity'):
def insert(self, position, entity): ...

@collection.removes_return()
def popitem(self): ...
```

Decorators can be specified in long-hand for Python 2.3, or with the class-level dict attribute '__instrumentation__'- see the source for details.

`collection_adapter`(*collection*)

Fetch the CollectionAdapter for a collection.

`column_mapped_collection`(*mapping_spec*)

A dictionary-based collection type with column-based keying.

Returns a MappedCollection factory with a keying function generated from mapping_spec, which may be a Column or a sequence of Columns.

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

`mapped_collection`(*keyfunc*)

A dictionary-based collection type with arbitrary keying.

Returns a MappedCollection factory with a keying function generated from keyfunc, a callable that takes an entity and returns a key value.

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

8.2.3 Querying

The Query Object

`Query` is produced in terms of a given `Session`, using the `query()` function:

```
q = session.query(SomeMappedClass)
```

Following is the full interface for the `Query` object.

class `Query` (*entities, session=None*)
Encapsulates the object-fetching operations provided by Mappers.

`__init__` (*entities, session=None*)

`add_column` (*column*)
Add a SQL ColumnElement to the list of result columns to be returned.

`add_entity` (*entity, alias=None*)
add a mapped entity to the list of result columns to be returned.

`all` ()
Return the results represented by this `Query` as a list.

This results in an execution of the underlying query.

`autoflush` (*setting*)
Return a Query with a specific 'autoflush' setting.

Note that a Session with autoflush=False will not autoflush, even if this flag is set to True at the Query level. Therefore this flag is usually used only to disable autoflush for a specific Query.

`correlate` (**args*)

`count` ()
Apply this query's criterion to a SELECT COUNT statement.

If column expressions or LIMIT/OFFSET/DISTINCT are present, the query "SELECT count(1) FROM (SELECT ...)" is issued, so that the result matches the total number of rows this query would return. For mapped entities, the primary key columns of each is written to the columns clause of the nested SELECT statement.

For a Query which is only against mapped entities, a simpler "SELECT count(1) FROM table1, table2, ... WHERE criterion" is issued.

`delete` (*synchronize_session='fetch'*)
Perform a bulk delete query.

Deletes rows matched by this query from the database.

Parameter *synchronize_session* – chooses the strategy for the removal of matched objects from the session. Valid values are:

False don't synchronize the session. This option is the most efficient and is reliable once the session is expired, which typically occurs after a commit(). Before the expiration, objects may still remain in the session which were in fact deleted which can lead to confusing results if they are accessed via get() or already loaded collections.

'fetch' performs a select query before the delete to find objects that are matched by the delete query and need to be removed from the session. Matched objects are removed from the session. 'fetch' is the default strategy.

'evaluate' experimental feature. Tries to evaluate the querys criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, the 'fetch' strategy will be used as a fallback.
The expression evaluator currently doesn't account for differing string collations between the database and Python.

Returns the number of rows deleted, excluding any cascades.

The method does *not* offer in-Python cascading of relations - it is assumed that ON DELETE CASCADE is configured for any foreign key references which require it. The Session needs to be expired (occurs automatically after commit(), or call expire_all()) in order for the state of dependent objects subject to delete or delete-orphan cascade to be correctly represented.

Also, the `before_delete()` and `after_delete()` `MapperExtension` methods are not called from this method. For a delete hook here, use the `after_bulk_delete()` `MapperExtension` method.

`distinct``()`
Apply a `DISTINCT` to the query and return the newly resulting `Query`.

`enable_eagerloads``(`*value*`)`
Control whether or not eager joins are rendered.

When set to False, the returned Query will not render eager joins regardless of eagerload() options or mapper-level lazy=False configurations.

This is used primarily when nesting the Query's statement into a subquery or other selectable.

`except_``(`**q*`)`
Produce an EXCEPT of this Query against one or more queries.

Works the same way as `union()`. See that method for usage examples.

`except_all``(`**q*`)`
Produce an EXCEPT ALL of this Query against one or more queries.

Works the same way as `union()`. See that method for usage examples.

`filter``(`*criterion*`)`
apply the given filtering criterion to the query and return the newly resulting `Query`

the criterion is any sql.ClauseElement applicable to the WHERE clause of a select.

`filter_by``(`***kwargs*`)`
apply the given filtering criterion to the query and return the newly resulting `Query`.

`first``()`
Return the first result of this `Query` or None if the result doesn't contain any row.

This results in an execution of the underlying query.

`from_self``(`**entities*`)`
return a Query that selects from this Query's SELECT statement.

*entities - optional list of entities which will replace those being selected.

`from_statement``(`*statement*`)`
Execute the given SELECT statement and return results.

This method bypasses all internal statement compilation, and the statement is executed without modification.

The statement argument is either a string, a `select()` construct, or a `text()` construct, and should return the set of columns appropriate to the entity class represented by this `Query`.

Also see the `instances()` method.

`get``(`*ident*`)`
Return an instance of the object based on the given identifier, or None if not found.

The *ident* argument is a scalar or tuple of primary key column values in the order of the table def's primary key columns.

`group_by``(`**criterion*`)`
apply one or more GROUP BY criterion to the query and return the newly resulting `Query`

`having``(`*criterion*`)`
apply a HAVING criterion to the query and return the newly resulting `Query`.

instances (*cursor, _Query__context=None*)
Given a ResultProxy cursor as returned by connection.execute(), return an ORM result as an iterator.

e.g.:

```
result = engine.execute("select * from users")
for u in session.query(User).instances(result):
    print u
```

intersect (**q*)
Produce an INTERSECT of this Query against one or more queries.

Works the same way as `union()`. See that method for usage examples.

intersect_all (**q*)
Produce an INTERSECT ALL of this Query against one or more queries.

Works the same way as `union()`. See that method for usage examples.

iterate_instances (*cursor, _Query__context=None*)
Given a ResultProxy cursor as returned by connection.execute(), return an ORM result as an iterator.

Deprecated.

e.g.:

```
result = engine.execute("select * from users")
for u in session.query(User).instances(result):
    print u
```

join (**props, **kwargs*)
Create a join against this `Query` object's criterion and apply generatively, returning the newly resulting `Query`.

Each element in *props may be:

- a string property name, i.e. "rooms". This will join along the relation of the same name from this Query's "primary" mapper, if one is present.
- a class-mapped attribute, i.e. Houses.rooms. This will create a join from "Houses" table to that of the "rooms" relation.
- a 2-tuple containing a target class or selectable, and an "ON" clause. The ON clause can be the property name/ attribute like above, or a SQL expression.

e.g.:

```
# join along string attribute names
session.query(Company).join('employees')
session.query(Company).join('employees', 'tasks')

# join the Person entity to an alias of itself,
# along the "friends" relation
PAlias = aliased(Person)
session.query(Person).join((Palias, Person.friends))

# join from Houses to the "rooms" attribute on the
# "Colonials" subclass of Houses, then join to the
# "closets" relation on Room
session.query(Houses).join(Colonials.rooms, Room.closets)

# join from Company entities to the "employees" collection,
# using "people JOIN engineers" as the target.  Then join
# to the "computers" collection on the Engineer entity.
session.query(Company).join((people.join(engineers), 'employees'), Engineer.compute

# join from Articles to Keywords, using the "keywords" attribute.
# assume this is a many-to-many relation.
session.query(Article).join(Article.keywords)

# same thing, but spelled out entirely explicitly
# including the association table.
session.query(Article).join(
    (article_keywords, Articles.id==article_keywords.c.article_id),
    (Keyword, Keyword.id==article_keywords.c.keyword_id)
    )
```

**kwargs include:

aliased - when joining, create anonymous aliases of each table. This is used for self-referential joins or multiple joins to the same table. Consider usage of the aliased(SomeClass) construct as a more explicit approach to this.

from_joinpoint - when joins are specified using string property names, locate the property from the mapper found in the most recent previous join() call, instead of from the root entity.

`limit`(*limit*)

Apply a `LIMIT` to the query and return the newly resulting

`Query`.

`offset`(*offset*)

Apply an `OFFSET` to the query and return the newly resulting `Query`.

`one`()

Return exactly one result or raise an exception.

Raises `sqlalchemy.orm.exc.NoResultFound` if the query selects no rows. Raises `sqlalchemy.orm.exc.MultipleResultsFound` if multiple rows are selected.

This results in an execution of the underlying query.

`options`(**args*)

Return a new Query object, applying the given list of MapperOptions.

`order_by`(**criterion*)

apply one or more ORDER BY criterion to the query and return the newly resulting `Query`

`outerjoin`(**props, **kwargs*)

Create a left outer join against this `Query` object's criterion and apply generatively, retunring the newly resulting `Query`.

Usage is the same as the `join()` method.

`params`(**args, **kwargs*)

add values for bind parameters which may have been specified in filter().

parameters may be specified using **kwargs, or optionally a single dictionary as the first positional argument. The reason for both is that **kwargs is convenient, however some parameter dictionaries contain unicode keys in which case **kwargs cannot be used.

`populate_existing`()

Return a Query that will refresh all instances loaded.

This includes all entities accessed from the database, including secondary entities, eagerly-loaded collection items.

All changes present on entities which are already present in the session will be reset and the entities will all be marked "clean".

An alternative to populate_existing() is to expire the Session fully using session.expire_all().

class **`query_from_parent`**(*instance, property, **kwargs*)

Return a new Query with criterion corresponding to a parent instance.

Deprecated. Use sqlalchemy.orm.with_parent in conjunction with filter().

Return a newly constructed Query object, with criterion corresponding to a relationship to the given parent instance.

instance a persistent or detached instance which is related to class represented by this query.

property string name of the property which relates this query's class to the instance.

****kwargs** all extra keyword arguments are propagated to the constructor of Query.

reset_joinpoint()

return a new Query reset the 'joinpoint' of this Query reset back to the starting mapper. Subsequent generative calls will be constructed from the new joinpoint.

Note that each call to join() or outerjoin() also starts from the root.

scalar()

Return the first element of the first result or None.

```
>>> session.query(Item).scalar()
<Item>
>>> session.query(Item.id).scalar()
1
>>> session.query(Item.id).filter(Item.id < 0).scalar()
None
>>> session.query(Item.id, Item.name).scalar()
1
>>> session.query(func.count(Parent.id)).scalar()
20
```

This results in an execution of the underlying query.

select_from(*from_obj*)

Set the *from_obj* parameter of the query and return the newly resulting `Query`. This replaces the table which this Query selects from with the given table.

from_obj is a single table or selectable.

slice(*start, stop*)

apply LIMIT/OFFSET to the `Query` based on a range and return the newly resulting `Query`.

statement

The full SELECT statement represented by this Query.

subquery()

return the full SELECT statement represented by this Query, embedded within an Alias.

Eager JOIN generation within the query is disabled.

union(**q*)

Produce a UNION of this Query against one or more queries.

e.g.:

```
q1 = sess.query(SomeClass).filter(SomeClass.foo=='bar')
q2 = sess.query(SomeClass).filter(SomeClass.bar=='foo')

q3 = q1.union(q2)
```

The method accepts multiple Query objects so as to control the level of nesting. A series of `union()` calls such as:

```
x.union(y).union(z).all()
```

will nest on each `union()`, and produces:

```
SELECT * FROM (SELECT * FROM (SELECT * FROM X UNION SELECT * FROM y) UNION SELECT *
```

Whereas:

```
x.union(y, z).all()
```

produces:

```
SELECT * FROM (SELECT * FROM X UNION SELECT * FROM y UNION SELECT * FROM Z)
```

union_all(**q*)
Produce a UNION ALL of this Query against one or more queries.

Works the same way as `union()`. See that method for usage examples.

update(*values, synchronize_session='expire'*)
Perform a bulk update query.

Updates rows matched by this query in the database.

Parameters
- *values* – a dictionary with attributes names as keys and literal values or sql expressions as values.
- *synchronize_session* – chooses the strategy to update the attributes on objects in the session. Valid values are:

  False don't synchronize the session. Use this when you don't need to use the session after the update or you can be sure that none of the matched objects are in the session.

  'expire' performs a select query before the update to find objects that are matched by the update query. The updated attributes are expired on matched objects.

  'evaluate' experimental feature. Tries to evaluate the querys criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, the 'expire' strategy will be used as a fallback.
  The expression evaluator currently doesn't account for differing string collations between the database and Python.

Returns the number of rows matched by the update.

The method does *not* offer in-Python cascading of relations - it is assumed that ON UPDATE CASCADE is configured for any foreign key references which require it. The Session needs to be expired (occurs automatically after commit(), or call expire_all()) in order for the state of dependent objects subject foreign key cascade to be correctly represented.

Also, the `before_update()` and `after_update()` `MapperExtension` methods are not called from this method. For an update hook here, use the `after_bulk_update()` `SessionExtension` method.

value(*column*)
Return a scalar result corresponding to the given column expression.

values(**columns*)
Return an iterator yielding result tuples corresponding to the given list of columns

whereclause
The WHERE criterion for this Query.

with_labels()
Apply column labels to the return value of Query.statement.

Indicates that this Query's *statement* accessor should return a SELECT statement that applies labels to all columns in the form <tablename>_<columnname>; this is commonly used to disambiguate columns from multiple tables which have the same name.

When the *Query* actually issues SQL to load rows, it always uses column labeling.

with_lockmode(*mode*)
Return a new Query object with the specified locking mode.

with_parent(*instance, property=None*)
Add a join criterion corresponding to a relationship to the given parent instance.

instance a persistent or detached instance which is related to class represented by this query.

property string name of the property which relates this query's class to the instance. if None, the method will attempt to find a suitable property.

Currently, this method only works with immediate parent relationships, but in the future may be enhanced to work across a chain of parent mappers.

with_polymorphic (*cls_or_mappers, selectable=None, discriminator=None*)
Load columns for descendant mappers of this Query's mapper.

Using this method will ensure that each descendant mapper's tables are included in the FROM clause, and will allow filter() criterion to be used against those tables. The resulting instances will also have those columns already loaded so that no "post fetch" of those columns will be required.

Parameters
- *cls_or_mappers* – a single class or mapper, or list of class/mappers, which inherit from this Query's mapper. Alternatively, it may also be the string `'*'`, in which case all descending mappers will be added to the FROM clause.
- *selectable* – a table or select() statement that will be used in place of the generated FROM clause. This argument is required if any of the desired mappers use concrete table inheritance, since SQLAlchemy currently cannot generate UNIONs among tables automatically. If used, the `selectable` argument must represent the full set of tables and columns mapped by every desired mapper. Otherwise, the unaccounted mapped columns will result in their table being appended directly to the FROM clause which will usually lead to incorrect results.
- *discriminator* – a column to be used as the "discriminator" column for the given selectable. If not given, the polymorphic_on attribute of the mapper will be used, if any. This is useful for mappers that don't have polymorphic loading behavior by default, such as concrete table mappers.

yield_per (*count*)
Yield only `count` rows at a time.

WARNING: use this method with caution; if the same instance is present in more than one batch of rows, end-user changes to attributes will be overwritten.

In particular, it's usually impossible to use this setting with eagerly loaded collections (i.e. any lazy=False) since those collections will be cleared for a new load when encountered in a subsequent result batch.

ORM-Specific Query Constructs

aliased
alias of `AliasedClass`

join (*left, right, onclause=None, isouter=False, join_to_left=True*)
Produce an inner join between left and right clauses.

In addition to the interface provided by `join()`, left and right may be mapped classes or AliasedClass instances. The onclause may be a string name of a relation(), or a class-bound descriptor representing a relation.

join_to_left indicates to attempt aliasing the ON clause, in whatever form it is passed, to the selectable passed as the left side. If False, the onclause is used as is.

outerjoin (*left, right, onclause=None, join_to_left=True*)
Produce a left outer join between left and right clauses.

In addition to the interface provided by `outerjoin()`, left and right may be mapped classes or AliasedClass instances. The onclause may be a string name of a relation(), or a class-bound descriptor representing a relation.

Query Options

Options which are passed to `query.options()`, to affect the behavior of loading.

contains_eager (**keys, **kwargs*)
Return a `MapperOption` that will indicate to the query that the given attribute will be eagerly loaded.

Used when feeding SQL result sets directly into `query.instances()`. Also bundles an `EagerLazyOption` to turn on eager loading in case it isn't already.

alias is the string name of an alias, **or** an `sql.Alias` object, which represents the aliased columns in the query. This argument is optional.

`defer`(**keys*)
Return a `MapperOption` that will convert the column property of the given name into a deferred load.

Used with `query.options()`

`eagerload`(**keys*)
Return a `MapperOption` that will convert the property of the given name into an eager load.

Used with `query.options()`.

`eagerload_all`(**keys*)
Return a `MapperOption` that will convert all properties along the given dot-separated path into an eager load.

For example, this:

```
query.options(eagerload_all('orders.items.keywords'))...
```

will set all of 'orders', 'orders.items', and 'orders.items.keywords' to load in one eager load.

Used with `query.options()`.

`extension`(*ext*)
Return a `MapperOption` that will insert the given `MapperExtension` to the beginning of the list of extensions that will be called in the context of the `Query`.

Used with `query.options()`.

`lazyload`(**keys*)
Return a `MapperOption` that will convert the property of the given name into a lazy load.

Used with `query.options()`.

`undefer`(**keys*)
Return a `MapperOption` that will convert the column property of the given name into a non-deferred (regular column) load.

Used with `query.options()`.

8.2.4 Sessions

`create_session`(*bind=None, **kwargs*)
Create a new `Session`.

Parameters
- *bind* – optional, a single Connectable to use for all database access in the created `Session`.
- ***kwargs* – optional, passed through to the `Session` constructor.

Returns an `Session` instance

The defaults of create_session() are the opposite of that of `sessionmaker()`; `autoflush` and `expire_on_commit` are False, `autocommit` is True. In this sense the session acts more like the "classic" SQLAlchemy 0.3 session with these.

Usage:

```
>>> from sqlalchemy.orm import create_session
>>> session = create_session()
```

It is recommended to use `sessionmaker()` instead of create_session().

scoped_session (*session_factory, scopefunc=None*)
Provides thread-local management of Sessions.

This is a front-end function to ScopedSession.

Parameters
- *session_factory* – a callable function that produces Session instances, such as sessionmaker() or create_session().
- *scopefunc* – optional, TODO

Returns an ScopedSession instance

Usage:

```
Session = scoped_session(sessionmaker(autoflush=True))
```

To instantiate a Session object which is part of the scoped context, instantiate normally:

```
session = Session()
```

Most session methods are available as classmethods from the scoped session:

```
Session.commit()
Session.close()
```

To map classes so that new instances are saved in the current Session automatically, as well as to provide session-aware class attributes such as "query", use the *mapper* classmethod from the scoped session:

```
mapper = Session.mapper
mapper(Class, table, ...)
```

sessionmaker (*bind=None, class_=None, autoflush=True, autocommit=False, expire_on_commit=True, **kwargs*)
Generate a custom-configured Session class.

The returned object is a subclass of Session, which, when instantiated with no arguments, uses the keyword arguments configured here as its constructor arguments.

It is intended that the *sessionmaker()* function be called within the global scope of an application, and the returned class be made available to the rest of the application as the single class used to instantiate sessions.

e.g.:

```
# global scope
Session = sessionmaker(autoflush=False)

# later, in a local scope, create and use a session:
sess = Session()
```

Any keyword arguments sent to the constructor itself will override the "configured" keywords:

```
Session = sessionmaker()

# bind an individual session to a connection
sess = Session(bind=connection)
```

The class also includes a special classmethod configure(), which allows additional configurational options to take place after the custom Session class has been generated. This is useful particularly for defining the specific Engine (or engines) to which new instances of Session should be bound:

```
Session = sessionmaker()
Session.configure(bind=create_engine('sqlite:///foo.db'))

sess = Session()
```

Options:

autocommit Defaults to `False`. When `True`, the `Session` does not keep a persistent transaction running, and will acquire connections from the engine on an as-needed basis, returning them immediately after their use. Flushes will begin and commit (or possibly rollback) their own transaction if no transaction is present. When using this mode, the *session.begin()* method may be used to begin a transaction explicitly.

Leaving it on its default value of `False` means that the `Session` will acquire a connection and begin a transaction the first time it is used, which it will maintain persistently until `rollback()`, `commit()`, or `close()` is called. When the transaction is released by any of these methods, the `Session` is ready for the next usage, which will again acquire and maintain a new connection/transaction.

autoflush When `True`, all query operations will issue a `flush()` call to this `Session` before proceeding. This is a convenience feature so that `flush()` need not be called repeatedly in order for database queries to retrieve results. It's typical that `autoflush` is used in conjunction with `autocommit=False`. In this scenario, explicit calls to `flush()` are rarely needed; you usually only need to call `commit()` (which flushes) to finalize changes.

bind An optional `Engine` or `Connection` to which this `Session` should be bound. When specified, all SQL operations performed by this session will execute via this connectable.

binds An optional dictionary, which contains more granular "bind" information than the `bind` parameter provides. This dictionary can map individual `Table` instances as well as `Mapper` instances to individual `Engine` or `Connection` objects. Operations which proceed relative to a particular `Mapper` will consult this dictionary for the direct `Mapper` instance as well as the mapper's `mapped_table` attribute in order to locate an connectable to use. The full resolution is described in the `get_bind()` method of `Session`. Usage looks like:

```
sess = Session(binds={
    SomeMappedClass: create_engine('postgres://engine1'),
    somemapper: create_engine('postgres://engine2'),
    some_table: create_engine('postgres://engine3'),
    })
```

Also see the `bind_mapper()` and `bind_table()` methods.

class_ Specify an alternate class other than `sqlalchemy.orm.session.Session` which should be used by the returned class. This is the only argument that is local to the `sessionmaker()` function, and is not sent directly to the constructor for `Session`.

echo_uow Deprecated. Use `logging.getLogger('sqlalchemy.orm.unitofwork').setLevel(logging.DEB`

_enable_transaction_accounting Defaults to `True`. A legacy-only flag which when `False` disables *all* 0.5-style object accounting on transaction boundaries, including auto-expiry of instances on rollback and commit, maintenance of the "new" and "deleted" lists upon rollback, and autoflush of pending changes upon begin(), all of which are interdependent.

expire_on_commit Defaults to `True`. When `True`, all instances will be fully expired after each `commit()`, so that all attribute/object access subsequent to a completed transaction will load from the most recent database state.

extension An optional `SessionExtension` instance, or a list of such instances, which will receive pre- and post- commit and flush events, as well as a post-rollback event. User- defined code may be placed within these hooks using a user-defined subclass of `SessionExtension`.

query_cls Class which should be used to create new Query objects, as returned by the `query()` method. Defaults to `Query`.

twophase When `True`, all transactions will be started using :mod:~sqlalchemy.engine_TwoPhaseTransaction. During a `commit()`, after `flush()` has been issued for all attached databases, the `prepare()` method on each database's `TwoPhaseTransaction` will be called. This allows each database to roll back the entire transaction, before each transaction is committed.

weak_identity_map When set to the default value of `True`, a weak-referencing map is used; instances which are not externally referenced will be garbage collected immediately. For dereferenced instances which have pending changes present, the attribute management system will create a temporary strong-reference to the object which lasts until the changes are flushed to the database, at which point it's again dereferenced. Alternatively, when using the value `False`, the identity map uses a regular Python dictionary to store instances. The session will maintain all instances present until they are removed using expunge(), clear(), or purge().

class Session(*bind=None, autoflush=True, expire_on_commit=True, _enable_transaction_accounting=True, autocommit=False, twophase=False, echo_uow=None, weak_identity_map=True, binds=None, extension=None, query_cls=<class 'sqlalchemy.orm.query.Query'>*)

Manages persistence operations for ORM-mapped objects.

The Session is the front end to SQLAlchemy's **Unit of Work** implementation. The concept behind Unit of Work is to track modifications to a field of objects, and then be able to flush those changes to the database in a single operation.

SQLAlchemy's unit of work includes these functions:

- The ability to track in-memory changes on scalar- and collection-based object attributes, such that database persistence operations can be assembled based on those changes.
- The ability to organize individual SQL queries and population of newly generated primary and foreign key-holding attributes during a persist operation such that referential integrity is maintained at all times.
- The ability to maintain insert ordering against the order in which new instances were added to the session.
- An Identity Map, which is a dictionary keying instances to their unique primary key identity. This ensures that only one copy of a particular entity is ever present within the session, even if repeated load operations for the same entity occur. This allows many parts of an application to get a handle to a particular object without any chance of modifications going to two different places.

When dealing with instances of mapped classes, an instance may be *attached* to a particular Session, else it is *unattached* . An instance also may or may not correspond to an actual row in the database. These conditions break up into four distinct states:

- *Transient* - an instance that's not in a session, and is not saved to the database; i.e. it has no database identity. The only relationship such an object has to the ORM is that its class has a `mapper()` associated with it.
- *Pending* - when you `add()` a transient instance, it becomes pending. It still wasn't actually flushed to the database yet, but it will be when the next flush occurs.
- *Persistent* - An instance which is present in the session and has a record in the database. You get persistent instances by either flushing so that the pending instances become persistent, or by querying the database for existing instances (or moving persistent instances from other sessions into your local session).
- *Detached* - an instance which has a record in the database, but is not in any session. Theres nothing wrong with this, and you can use objects normally when they're detached, **except** they will not be able to issue any SQL in order to load collections or attributes which are not yet loaded, or were marked as "expired".

The session methods which control instance state include `add()`, `delete()`, `merge()`, and `expunge()`.

The Session object is generally **not** threadsafe. A session which is set to `autocommit` and is only read from may be used by concurrent threads if it's acceptable that some object instances may be loaded twice.

The typical pattern to managing Sessions in a multi-threaded environment is either to use mutexes to limit concurrent access to one thread at a time, or more commonly to establish a unique session for every thread, using a threadlocal variable. SQLAlchemy provides a thread-managed Session adapter, provided by the `scoped_session()` function.

`__init__`(*bind=None, autoflush=True, expire_on_commit=True, _enable_transaction_accounting=True, autocommit=False, twophase=False, echo_uow=None, weak_identity_map=True, binds=None, extension=None, query_cls=<class 'sqlalchemy.orm.query.Query'>*)

Construct a new Session.

Arguments to `Session` are described using the `sessionmaker()` function.

`add`(*instance*)

Place an object in the `Session`.

Its state will be persisted to the database on the next flush operation.

Repeated calls to `add()` will be ignored. The opposite of `add()` is `expunge()`.

`add_all`(*instances*)

Add the given collection of instances to this `Session`.

`begin`(*subtransactions=False, nested=False*)

Begin a transaction on this Session.

If this Session is already within a transaction, either a plain transaction or nested transaction, an error is raised, unless `subtransactions=True` or `nested=True` is specified.

The `subtransactions=True` flag indicates that this `begin()` can create a subtransaction if a transaction is already in progress. A subtransaction is a non-transactional, delimiting construct that allows matching begin()/commit() pairs to be nested together, with only the outermost begin/commit pair actually affecting transactional state. When a rollback is issued, the subtransaction will directly roll back the innermost real transaction, however each subtransaction still must be explicitly rolled back to maintain proper stacking of subtransactions.

If no transaction is in progress, then a real transaction is begun.

The `nested` flag begins a SAVEPOINT transaction and is equivalent to calling `begin_nested()`.

`begin_nested`()

Begin a *nested* transaction on this Session.

The target database(s) must support SQL SAVEPOINTs or a SQLAlchemy-supported vendor implementation of the idea.

The nested transaction is a real transation, unlike a "subtransaction" which corresponds to multiple `begin()` calls. The next `rollback()` or `commit()` call will operate upon this nested transaction.

`bind_mapper`(*mapper, bind*)

Bind operations for a mapper to a Connectable.

mapper A mapper instance or mapped class

bind Any Connectable: a `Engine` or `Connection`.

All subsequent operations involving this mapper will use the given *bind*.

`bind_table`(*table, bind*)

Bind operations on a Table to a Connectable.

table A `Table` instance

bind Any Connectable: a `Engine` or `Connection`.

All subsequent operations involving this `Table` will use the given *bind*.

`clear`()

Remove all object instances from this `Session`.

Use session.expunge_all()

This is equivalent to calling `expunge(obj)` on all objects in this `Session`.

`close`()

Close this Session.

This clears all items and ends any transaction in progress.

If this session were created with `autocommit=False`, a new transaction is immediately begun. Note that this new transaction does not use any connection resources until they are first needed.

class **close_all**()
Close *all* sessions in memory.

commit()
Flush pending changes and commit the current transaction.

If no transaction is in progress, this method raises an InvalidRequestError.

If a subtransaction is in effect (which occurs when begin() is called multiple times), the subtransaction will be closed, and the next call to `commit()` will operate on the enclosing transaction.

For a session configured with autocommit=False, a new transaction will be begun immediately after the commit, but note that the newly begun transaction does *not* use any connection resources until the first SQL is actually emitted.

connection(*mapper=None, clause=None*)
Return the active Connection.

Retrieves the `Connection` managing the current transaction. Any operations executed on the Connection will take place in the same transactional context as `Session` operations.

For `autocommit` Sessions with no active manual transaction, `connection()` is a passthrough to `contextual_connect()` on the underlying engine.

Ambiguity in multi-bind or unbound Sessions can be resolved through any of the optional keyword arguments. See `get_bind()` for more information.

mapper Optional, a `mapper` or mapped class

clause Optional, any `ClauseElement`

delete(*instance*)
Mark an instance as deleted.

The database delete operation occurs upon `flush()`.

deleted
The set of all instances marked as 'deleted' within this `Session`

dirty
The set of all persistent instances considered dirty.

Instances are considered dirty when they were modified but not deleted.

Note that this 'dirty' calculation is 'optimistic'; most attribute-setting or collection modification operations will mark an instance as 'dirty' and place it in this set, even if there is no net change to the attribute's value. At flush time, the value of each attribute is compared to its previously saved value, and if there's no net change, no SQL operation will occur (this is a more expensive operation so it's only done at flush time).

To check if an instance has actionable net changes to its attributes, use the is_modified() method.

execute(*clause, params=None, mapper=None, **kw*)
Execute a clause within the current transaction.

Returns a `ResultProxy` of execution results. *autocommit* Sessions will create a transaction on the fly.

Connection ambiguity in multi-bind or unbound Sessions will be resolved by inspecting the clause for binds. The 'mapper' and 'instance' keyword arguments may be used if this is insufficient, See `get_bind()` for more information.

clause A ClauseElement (i.e. select(), text(), etc.) or string SQL statement to be executed

params Optional, a dictionary of bind parameters.

mapper Optional, a `mapper` or mapped class

****kw** Additional keyword arguments are sent to `get_bind()` which locates a connectable to use for the execution. Subclasses of `Session` may override this.

expire(*instance, attribute_names=None*)
Expire the attributes on an instance.

Marks the attributes of an instance as out of date. When an expired attribute is next accessed, query will be issued to the database and the attributes will be refreshed with their current database value. `expire()` is a lazy variant of `refresh()`.

The `attribute_names` argument is an iterable collection of attribute names indicating a subset of attributes to be expired.

`expire_all`()
Expires all persistent instances within this Session.

`expunge`(*instance*)
Remove the *instance* from this `Session`.

This will free all internal references to the instance. Cascading will be applied according to the *expunge* cascade rule.

`expunge_all`()
Remove all object instances from this `Session`.

This is equivalent to calling `expunge(obj)` on all objects in this `Session`.

`flush`(*objects=None*)
Flush all the object changes to the database.

Writes out all pending object creations, deletions and modifications to the database as INSERTs, DELETEs, UPDATEs, etc. Operations are automatically ordered by the Session's unit of work dependency solver..

Database operations will be issued in the current transactional context and do not affect the state of the transaction. You may flush() as often as you like within a transaction to move changes from Python to the database's transaction buffer.

For `autocommit` Sessions with no active manual transaction, flush() will create a transaction on the fly that surrounds the entire set of operations int the flush.

objects Optional; a list or tuple collection. Restricts the flush operation to only these objects, rather than all pending changes. Deprecated - this flag prevents the session from properly maintaining accounting among inter-object relations and can cause invalid results.

`get_bind`(*mapper, clause=None*)
Return an engine corresponding to the given arguments.

All arguments are optional.

mapper Optional, a `Mapper` or mapped class

clause Optional, A ClauseElement (i.e. select(), text(), etc.)

`is_active`
True if this Session has an active transaction.

`is_modified`(*instance, include_collections=True, passive=False*)
Return True if instance has modified attributes.

This method retrieves a history instance for each instrumented attribute on the instance and performs a comparison of the current value to its previously committed value. Note that instances present in the 'dirty' collection may result in a value of `False` when tested with this method.

include_collections indicates if multivalued collections should be included in the operation. Setting this to False is a way to detect only local-column based properties (i.e. scalar columns or many-to-one foreign keys) that would result in an UPDATE for this instance upon flush.

The *passive* flag indicates if unloaded attributes and collections should not be loaded in the course of performing this test.

`merge`(*instance, dont_load=False*)
Copy the state an instance onto the persistent instance with the same identifier.

If there is no persistent instance currently associated with the session, it will be loaded. Return the persistent instance. If the given instance is unsaved, save a copy of and return it as a newly persistent instance. The given instance does not become associated with the session.

This operation cascades to associated instances if the association is mapped with `cascade="merge"`.

`new`
The set of all instances marked as 'new' within this `Session`.

class **`object_session`**(*instance*)
Return the `Session` to which an object belongs.

`prepare`()
Prepare the current transaction in progress for two phase commit.

If no transaction is in progress, this method raises an InvalidRequestError.

Only root transactions of two phase sessions can be prepared. If the current transaction is not such, an InvalidRequestError is raised.

`prune`()
Remove unreferenced instances cached in the identity map.

Note that this method is only meaningful if "weak_identity_map" is set to False. The default weak identity map is self-pruning.

Removes any object in this Session's identity map that is not referenced in user code, modified, new or scheduled for deletion. Returns the number of objects pruned.

`query`(**entities, **kwargs*)
Return a new `Query` object corresponding to this `Session`.

`refresh`(*instance, attribute_names=None*)
Refresh the attributes on the given instance.

A query will be issued to the database and all attributes will be refreshed with their current database value.

Lazy-loaded relational attributes will remain lazily loaded, so that the instance-wide refresh operation will be followed immediately by the lazy load of that attribute.

Eagerly-loaded relational attributes will eagerly load within the single refresh operation.

The `attribute_names` argument is an iterable collection of attribute names indicating a subset of attributes to be refreshed.

`rollback`()
Rollback the current transaction in progress.

If no transaction is in progress, this method is a pass-through.

This method rolls back the current transaction or nested transaction regardless of subtransactions being in effect. All subtrasactions up to the first real transaction are closed. Subtransactions occur when begin() is called mulitple times.

`save`(*instance*)
Add a transient (unsaved) instance to this `Session`.

Use session.add()

This operation cascades the *save_or_update* method to associated instances if the relation is mapped with `cascade="save-update"`.

`save_or_update`(*instance*)
Place an object in the `Session`.

Use session.add()

Its state will be persisted to the database on the next flush operation.

Repeated calls to `add()` will be ignored. The opposite of `add()` is `expunge()`.

`scalar`(*clause, params=None, mapper=None, **kw*)
Like execute() but return a scalar result.

`update`(*instance*)
Bring a detached (saved) instance into this `Session`.

Use session.add()

If there is a persistent instance with the same instance key, but different identity already associated with this `Session`, an InvalidRequestError exception is thrown.

This operation cascades the *save_or_update* method to associated instances if the relation is mapped with `cascade="save-update"`.

class **ScopedSession** (*session_factory, scopefunc=None*)

Provides thread-local management of Sessions.

Usage:

```
Session = scoped_session(sessionmaker(autoflush=True))

To map classes so that new instances are saved in the current
Session automatically, as well as to provide session-aware
class attributes such as "query":

mapper = Session.mapper
mapper(Class, table, ...)
```

__init__ (*session_factory, scopefunc=None*)

configure (***kwargs*)

reconfigure the sessionmaker used by this ScopedSession.

mapper (**args, **kwargs*)

return a mapper() function which associates this ScopedSession with the Mapper.

query_property (*query_cls=None*)

return a class property which produces a *Query* object against the class when called.

e.g.:: Session = scoped_session(sessionmaker())

class MyClass(object): query = Session.query_property()

# after mappers are defined result = MyClass.query.filter(MyClass.name=='foo').all()

Produces instances of the session's configured query class by default. To override and use a custom implementation, provide a `query_cls` callable. The callable will be invoked with the class's mapper as a positional argument and a session keyword argument.

There is no limit to the number of query properties placed on a class.

8.2.5 Interfaces

Semi-private module containing various base classes used throughout the ORM.

Defines the extension classes `MapperExtension`, `SessionExtension`, and `AttributeExtension` as well as other user-subclassable extension objects.

class **AttributeExtension** ()

An event handler for individual attribute change events.

AttributeExtension is assembled within the descriptors associated with a mapped class.

append (*state, value, initiator*)

Receive a collection append event.

The returned value will be used as the actual value to be appended.

remove (*state, value, initiator*)

Receive a remove event.

No return value is defined.

set (*state, value, oldvalue, initiator*)

Receive a set event.

The returned value will be used as the actual value to be set.

class **InstrumentationManager** (*class_*)

User-defined class instrumentation extension.

The API for this class should be considered as semi-stable, and may change slightly with new releases.

`__init__`(*class_*)

`dict_getter`(*class_*)

`dispose`(*class_, manager*)

`get_instance_dict`(*class_, instance*)

`initialize_instance_dict`(*class_, instance*)

`install_descriptor`(*class_, key, inst*)

`install_member`(*class_, key, implementation*)

`install_state`(*class_, instance, state*)

`instrument_attribute`(*class_, key, inst*)

`instrument_collection_class`(*class_, key, collection_class*)

`manage`(*class_, manager*)

`manager_getter`(*class_*)

`post_configure_attribute`(*class_, key, inst*)

`remove_state`(*class_, instance*)

`state_getter`(*class_*)

`uninstall_descriptor`(*class_, key*)

`uninstall_member`(*class_, key*)

class `MapperExtension`()

Base implementation for customizing Mapper behavior.

For each method in MapperExtension, returning a result of EXT_CONTINUE will allow processing to continue to the next MapperExtension in line or use the default functionality if there are no other extensions.

Returning EXT_STOP will halt processing of further extensions handling that method. Some methods such as `load` have other return requirements, see the individual documentation for details. Other than these exception cases, any return value other than EXT_CONTINUE or EXT_STOP will be interpreted as equivalent to EXT_STOP.

`after_delete`(*mapper, connection, instance*)
Receive an object instance after that instance is DELETEed.

`after_insert`(*mapper, connection, instance*)
Receive an object instance after that instance is INSERTed.

`after_update`(*mapper, connection, instance*)
Receive an object instance after that instance is UPDATEed.

`append_result`(*mapper, selectcontext, row, instance, result, **flags*)
Receive an object instance before that instance is appended to a result list.

If this method returns EXT_CONTINUE, result appending will proceed normally. if this method returns any other value or None, result appending will not proceed for this instance, giving this extension an opportunity to do the appending itself, if desired.

mapper The mapper doing the operation.

selectcontext SelectionContext corresponding to the instances() call.

row The result row from the database.

instance The object instance to be appended to the result.

result List to which results are being appended.

****flags** extra information about the row, same as criterion in `create_row_processor()` method of `MapperProperty`

before_delete (*mapper, connection, instance*)
Receive an object instance before that instance is DELETEed.

Note that *no* changes to the overall flush plan can be made here; this means any collection modification, save() or delete() operations which occur within this method will not take effect until the next flush call.

before_insert (*mapper, connection, instance*)
Receive an object instance before that instance is INSERTed into its table.

This is a good place to set up primary key values and such that aren't handled otherwise.

Column-based attributes can be modified within this method which will result in the new value being inserted. However *no* changes to the overall flush plan can be made; this means any collection modification or save() operations which occur within this method will not take effect until the next flush call.

before_update (*mapper, connection, instance*)
Receive an object instance before that instance is UPDATEed.

Note that this method is called for all instances that are marked as "dirty", even those which have no net changes to their column-based attributes. An object is marked as dirty when any of its column-based attributes have a "set attribute" operation called or when any of its collections are modified. If, at update time, no column-based attributes have any net changes, no UPDATE statement will be issued. This means that an instance being sent to before_update is *not* a guarantee that an UPDATE statement will be issued (although you can affect the outcome here).

To detect if the column-based attributes on the object have net changes, and will therefore generate an UPDATE statement, use `object_session(instance).is_modified(instance, include_collections=False)`.

Column-based attributes can be modified within this method which will result in their being updated. However *no* changes to the overall flush plan can be made; this means any collection modification or save() operations which occur within this method will not take effect until the next flush call.

create_instance (*mapper, selectcontext, row, class_*)
Receive a row when a new object instance is about to be created from that row.

The method can choose to create the instance itself, or it can return EXT_CONTINUE to indicate normal object creation should take place.

mapper The mapper doing the operation

selectcontext SelectionContext corresponding to the instances() call

row The result row from the database

class_ The class we are mapping.

return value A new object instance, or EXT_CONTINUE

init_failed (*mapper, class_, oldinit, instance, args, kwargs*)

init_instance (*mapper, class_, oldinit, instance, args, kwargs*)

instrument_class (*mapper, class_*)

populate_instance (*mapper, selectcontext, row, instance, **flags*)
Receive an instance before that instance has its attributes populated.

This usually corresponds to a newly loaded instance but may also correspond to an already-loaded instance which has unloaded attributes to be populated. The method may be called many times for a single instance, as multiple result rows are used to populate eagerly loaded collections.

If this method returns EXT_CONTINUE, instance population will proceed normally. If any other value or None is returned, instance population will not proceed, giving this extension an opportunity to populate the instance itself, if desired.

As of 0.5, most usages of this hook are obsolete. For a generic "object has been newly created from a row" hook, use `reconstruct_instance()`, or the `@orm.reconstructor` decorator.

reconstruct_instance (*mapper, instance*)
Receive an object instance after it has been created via `__new__`, and after initial attribute population has occurred.

This typically occurs when the instance is created based on incoming result rows, and is only called once for that instance's lifetime.

Note that during a result-row load, this method is called upon the first row received for this instance. If eager loaders are set to further populate collections on the instance, those will *not* yet be completely loaded.

`translate_row`(*mapper, context, row*)

Perform pre-processing on the given result row and return a new row instance.

This is called when the mapper first receives a row, before the object identity or the instance itself has been derived from that row.

class **`PropComparator`**(*prop, mapper, adapter=None*)

defines comparison operations for MapperProperty objects.

PropComparator instances should also define an accessor 'property' which returns the MapperProperty associated with this PropComparator.

`__init__`(*prop, mapper, adapter=None*)

`adapted`(*adapter*)

Return a copy of this PropComparator which will use the given adaption function on the local side of generated expressions.

`any`(*criterion=None, **kwargs*)

Return true if this collection contains any member that meets the given criterion.

criterion an optional ClauseElement formulated against the member class' table or attributes.

****kwargs** key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

static **`any_op`**(*a, b, **kwargs*)

`has`(*criterion=None, **kwargs*)

Return true if this element references a member which meets the given criterion.

criterion an optional ClauseElement formulated against the member class' table or attributes.

****kwargs** key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

static **`has_op`**(*a, b, **kwargs*)

`of_type`(*class_*)

Redefine this object in terms of a polymorphic subclass.

Returns a new PropComparator from which further criterion can be evaluated.

e.g.:

```
query.join(Company.employees.of_type(Engineer)).\
   filter(Engineer.name=='foo')
```

class_ a class or mapper indicating that criterion will be against this specific subclass.

static **`of_type_op`**(*a, class_*)

class **`SessionExtension`**()

An extension hook object for Sessions. Subclasses may be installed into a Session (or sessionmaker) using the `extension` keyword argument.

`after_attach`(*session, instance*)

Execute after an instance is attached to a session.

This is called after an add, delete or merge.

`after_begin`(*session, transaction, connection*)

Execute after a transaction is begun on a connection

transaction is the SessionTransaction. This method is called after an engine level transaction is begun on a connection.

`after_bulk_delete` (*session, query, query_context, result*)
Execute after a bulk delete operation to the session.

This is called after a session.query(...).delete()

query is the query object that this delete operation was called on. *query_context* was the query context object. *result* is the result object returned from the bulk operation.

`after_bulk_update` (*session, query, query_context, result*)
Execute after a bulk update operation to the session.

This is called after a session.query(...).update()

query is the query object that this update operation was called on. *query_context* was the query context object. *result* is the result object returned from the bulk operation.

`after_commit` (*session*)
Execute after a commit has occured.

Note that this may not be per-flush if a longer running transaction is ongoing.

`after_flush` (*session, flush_context*)
Execute after flush has completed, but before commit has been called.

Note that the session's state is still in pre-flush, i.e. 'new', 'dirty', and 'deleted' lists still show pre-flush state as well as the history settings on instance attributes.

`after_flush_postexec` (*session, flush_context*)
Execute after flush has completed, and after the post-exec state occurs.

This will be when the 'new', 'dirty', and 'deleted' lists are in their final state. An actual commit() may or may not have occured, depending on whether or not the flush started its own transaction or participated in a larger transaction.

`after_rollback` (*session*)
Execute after a rollback has occured.

Note that this may not be per-flush if a longer running transaction is ongoing.

`before_commit` (*session*)
Execute right before commit is called.

Note that this may not be per-flush if a longer running transaction is ongoing.

`before_flush` (*session, flush_context, instances*)
Execute before flush process has started.

instances is an optional list of objects which were passed to the `flush()` method.

8.2.6 Utilities

`identity_key` (**args, **kwargs*)
Get an identity key.

Valid call signatures:

- `identity_key(class, ident)`

  class mapped class (must be a positional argument)

  ident primary key, if the key is composite this is a tuple

- `identity_key(instance=instance)`

  instance object instance (must be given as a keyword arg)

- `identity_key(class, row=row)`

  class mapped class (must be a positional argument)

  row result proxy row (must be given as a keyword arg)

class `Validator` (*key, validator*)
Runs a validation method on an attribute value to be set or appended.

The Validator class is used by the `validates()` decorator, and direct access is usually not needed.

`__init__` (*key, validator*)
Construct a new Validator.

key - name of the attribute to be validated; will be passed as the second argument to the validation method (the first is the object instance itself).

validator - an function or instance method which accepts three arguments; an instance (usually just 'self' for a method), the key name of the attribute, and the value. The function should return the same value given, unless it wishes to modify it.

`append` (*state, value, initiator*)

`set` (*state, value, oldvalue, initiator*)

`with_parent` (*instance, prop*)
Return criterion which selects instances with a given parent.

instance a parent instance, which should be persistent or detached.

property a class-attached descriptor, MapperProperty or string property name attached to the parent instance.

****kwargs** all extra keyword arguments are propagated to the constructor of Query.

`polymorphic_union` (*table_map, typecolname, aliasname='p_union'*)
Create a `UNION` statement used by a polymorphic mapper.

See *Concrete Table Inheritance* for an example of how this is used.

8.3 sqlalchemy.databases

8.3.1 Access

8.3.2 Firebird

Firebird backend

This module implements the Firebird backend, thru the kinterbasdb DBAPI module.

Firebird dialects

Firebird offers two distinct dialects (not to be confused with the SA `Dialect` thing):

dialect 1 This is the old syntax and behaviour, inherited from Interbase pre-6.0.

dialect 3 This is the newer and supported syntax, introduced in Interbase 6.0.

From the user point of view, the biggest change is in date/time handling: under dialect 1, there's a single kind of field, `DATE` with a synonim `DATETIME`, that holds a *timestamp* value, that is a date with hour, minute, second. Under dialect 3 there are three kinds, a `DATE` that holds a date, a `TIME` that holds a *time of the day* value and a `TIMESTAMP`, equivalent to the old `DATE`.

The problem is that the dialect of a Firebird database is a property of the database itself [1] (that is, any single database has been created with one dialect or the other: there is no way to change the after creation). SQLAlchemy has a single

[1] Well, that is not the whole story, as the client may still ask a different (lower) dialect...

instance of the class that controls all the connections to a particular kind of database, so it cannot easily differentiate between the two modes, and in particular it **cannot** simultaneously talk with two distinct Firebird databases with different dialects.

By default this module is biased toward dialect 3, but you can easily tweak it to handle dialect 1 if needed:

```
from sqlalchemy import types as sqltypes
from sqlalchemy.databases.firebird import FBDate, colspecs, ischema_names

# Adjust the mapping of the timestamp kind
ischema_names['TIMESTAMP'] = FBDate
colspecs[sqltypes.DateTime] = FBDate,
```

Other aspects may be version-specific. You can use the `server_version_info()` method on the `FBDialect` class to do whatever is needed:

```
from sqlalchemy.databases.firebird import FBCompiler

if engine.dialect.server_version_info(connection) < (2,0):
    # Change the name of the function ``length`` to use the UDF version
    # instead of ``char_length``
    FBCompiler.LENGTH_FUNCTION_NAME = 'strlen'
```

Pooling connections

The default strategy used by SQLAlchemy to pool the database connections in particular cases may raise an `OperationalError` with a message *"object XYZ is in use"*. This happens on Firebird when there are two connections to the database, one is using, or has used, a particular table and the other tries to drop or alter the same table. To garantee DDL operations success Firebird recommend doing them as the single connected user.

In case your SA application effectively needs to do DDL operations while other connections are active, the following setting may alleviate the problem:

```
from sqlalchemy import pool
from sqlalchemy.databases.firebird import dialect

# Force SA to use a single connection per thread
dialect.poolclass = pool.SingletonThreadPool
```

RETURNING support

Firebird 2.0 supports returning a result set from inserts, and 2.1 extends that to deletes and updates.

To use this pass the column/expression list to the `firebird_returning` parameter when creating the queries:

```
raises = tbl.update(empl.c.sales > 100, values=dict(salary=empl.c.salary * 1.1),
                    firebird_returning=[empl.c.id, empl.c.salary]).execute().fetchall()
```

8.3.3 Informix

8.3.4 MaxDB

Support for the MaxDB database.

TODO: More module docs! MaxDB support is currently experimental.

Overview

The `maxdb` dialect is **experimental** and has only been tested on 7.6.03.007 and 7.6.00.037. Of these, **only 7.6.03.007 will work** with SQLAlchemy's ORM. The earlier version has severe `LEFT JOIN` limitations and will return incorrect results from even very simple ORM queries.

Only the native Python DB-API is currently supported. ODBC driver support is a future enhancement.

Connecting

The username is case-sensitive. If you usually connect to the database with sqlcli and other tools in lower case, you likely need to use upper case for DB-API.

Implementation Notes

Also check the DatabaseNotes page on the wiki for detailed information.

With the 7.6.00.37 driver and Python 2.5, it seems that all DB-API generated exceptions are broken and can cause Python to crash.

For 'somecol.in_([])' to work, the IN operator's generation must be changed to cast 'NULL' to a numeric, i.e. NUM(NULL). The DB-API doesn't accept a bind parameter there, so that particular generation must inline the NULL value, which depends on [ticket:807].

The DB-API is very picky about where bind params may be used in queries.

Bind params for some functions (e.g. MOD) need type information supplied. The dialect does not yet do this automatically.

Max will occasionally throw up 'bad sql, compile again' exceptions for perfectly valid SQL. The dialect does not currently handle these, more research is needed.

MaxDB 7.5 and Sap DB <= 7.4 reportedly do not support schemas. A very slightly different version of this dialect would be required to support those versions, and can easily be added if there is demand. Some other required components such as an Max-aware 'old oracle style' join compiler (thetas with (+) outer indicators) are already done and available for integration- email the devel list if you're interested in working on this.

8.3.5 SQL Server

Support for the Microsoft SQL Server database.

Driver

The MSSQL dialect will work with three different available drivers:

- *pyodbc* - http://pyodbc.sourceforge.net/. This is the recommeded driver.
- *pymssql* - http://pymssql.sourceforge.net/
- *adodbapi* - http://adodbapi.sourceforge.net/

Drivers are loaded in the order listed above based on availability.

If you need to load a specific driver pass `module_name` when creating the engine:

```
engine = create_engine('mssql://dsn', module_name='pymssql')
```

`module_name` currently accepts: `pyodbc`, `pymssql`, and `adodbapi`.

Currently the pyodbc driver offers the greatest level of compatibility.

Connecting

Connecting with create_engine() uses the standard URL approach of `mssql://user:pass@host/dbname[?key=value&key=`

If the database name is present, the tokens are converted to a connection string with the specified values. If the database is not present, then the host token is taken directly as the DSN name.

Examples of pyodbc connection string URLs:

- *mssql://mydsn* - connects using the specified DSN named `mydsn`. The connection string that is created will appear like:

  ```
  dsn=mydsn;TrustedConnection=Yes
  ```

- *mssql://user:pass@mydsn* - connects using the DSN named `mydsn` passing in the `UID` and `PWD` information. The connection string that is created will appear like:

  ```
  dsn=mydsn;UID=user;PWD=pass
  ```

- *mssql://user:pass@mydsn/?LANGUAGE=us_english* - connects using the DSN named `mydsn` passing in the `UID` and `PWD` information, plus the additional connection configuration option `LANGUAGE`. The connection string that is created will appear like:

  ```
  dsn=mydsn;UID=user;PWD=pass;LANGUAGE=us_english
  ```

- *mssql://user:pass@host/db* - connects using a connection string dynamically created that would appear like:

  ```
  DRIVER={SQL Server};Server=host;Database=db;UID=user;PWD=pass
  ```

- *mssql://user:pass@host:123/db* - connects using a connection string that is dynamically created, which also includes the port information using the comma syntax. If your connection string requires the port information to be passed as a `port` keyword see the next example. This will create the following connection string:

  ```
  DRIVER={SQL Server};Server=host,123;Database=db;UID=user;PWD=pass
  ```

- *mssql://user:pass@host/db?port=123* - connects using a connection string that is dynamically created that includes the port information as a separate `port` keyword. This will create the following connection string:

  ```
  DRIVER={SQL Server};Server=host;Database=db;UID=user;PWD=pass;port=123
  ```

If you require a connection string that is outside the options presented above, use the `odbc_connect` keyword to pass in a urlencoded connection string. What gets passed in will be urldecoded and passed directly.

For example:

```
mssql:///?odbc_connect=dsn%3Dmydsn%3BDatabase%3Ddb
```

would create the following connection string:

```
dsn=mydsn;Database=db
```

Encoding your connection string can be easily accomplished through the python shell. For example:

```
>>> import urllib
>>> urllib.quote_plus('dsn=mydsn;Database=db')
'dsn%3Dmydsn%3BDatabase%3Ddb'
```

Additional arguments which may be specified either as query string arguments on the URL, or as keyword argument to `create_engine()` are:

- *auto_identity_insert* - enables support for IDENTITY inserts by automatically turning IDENTITY INSERT ON and OFF as required. Defaults to `True`.
- *query_timeout* - allows you to override the default query timeout. Defaults to `None`. This is only supported on pymssql.
- *text_as_varchar* - if enabled this will treat all TEXT column types as their equivalent VARCHAR(max) type. This is often used if you need to compare a VARCHAR to a TEXT field, which is not supported directly on MSSQL. Defaults to `False`.
- *use_scope_identity* - allows you to specify that SCOPE_IDENTITY should be used in place of the non-scoped version @@IDENTITY. Defaults to `False`. On pymssql this defaults to `True`, and on pyodbc this defaults to `True` if the version of pyodbc being used supports it.
- *has_window_funcs* - indicates whether or not window functions (LIMIT and OFFSET) are supported on the version of MSSQL being used. If you're running MSSQL 2005 or later turn this on to get OFFSET support. Defaults to `False`.
- *max_identifier_length* - allows you to se the maximum length of identfiers supported by the database. Defaults to 128. For pymssql the default is 30.
- *schema_name* - use to set the schema name. Defaults to `dbo`.

Auto Increment Behavior

`IDENTITY` columns are supported by using SQLAlchemy `schema.Sequence()` objects. In other words:

```
Table('test', mss_engine,
       Column('id', Integer,
              Sequence('blah',100,10), primary_key=True),
       Column('name', String(20))
     ).create()
```

would yield:

```
CREATE TABLE test (
  id INTEGER NOT NULL IDENTITY(100,10) PRIMARY KEY,
  name VARCHAR(20) NULL,
  )
```

Note that the `start` and `increment` values for sequences are optional and will default to 1,1.

- Support for `SET IDENTITY_INSERT ON` mode (automagic on / off for `INSERT` s)
- Support for auto-fetching of `@@IDENTITY/@@SCOPE_IDENTITY()` on `INSERT`

Collation Support

MSSQL specific string types support a collation parameter that creates a column-level specific collation for the column. The collation parameter accepts a Windows Collation Name or a SQL Collation Name. Supported types are MSChar, MSNChar, MSString, MSNVarchar, MSText, and MSNText. For example:

```
Column('login', String(32, collation='Latin1_General_CI_AS'))
```

will yield:

```
login VARCHAR(32) COLLATE Latin1_General_CI_AS NULL
```

LIMIT/OFFSET Support

MSSQL has no support for the LIMIT or OFFSET keysowrds. LIMIT is supported directly through the `TOP` Transact SQL keyword:

```
select.limit
```

will yield:

```
SELECT TOP n
```

If the `has_window_funcs` flag is set then LIMIT with OFFSET support is available through the `ROW_NUMBER OVER` construct. This construct requires an `ORDER BY` to be specified as well and is only available on MSSQL 2005 and later.

Nullability

MSSQL has support for three levels of column nullability. The default nullability allows nulls and is explicit in the CREATE TABLE construct:

```
name VARCHAR(20) NULL
```

If `nullable=None` is specified then no specification is made. In other words the database's configured default is used. This will render:

```
name VARCHAR(20)
```

If `nullable` is `True` or `False` then the column will be `NULL` or ``NOT NULL` respectively.

Date / Time Handling

For MSSQL versions that support the `DATE` and `TIME` types (MSSQL 2008+) the data type is used. For versions that do not support the `DATE` and `TIME` types a `DATETIME` type is used instead and the MSSQL dialect handles converting the results properly. This means `Date()` and `Time()` are fully supported on all versions of MSSQL. If you do not desire this behavior then do not use the `Date()` or `Time()` types.

Compatibility Levels

MSSQL supports the notion of setting compatibility levels at the database level. This allows, for instance, to run a database that is compatibile with SQL2000 while running on a SQL2005 database server. `server_version_info` will always retrun the database server version information (in this case SQL2005) and not the compatibiility level information. Because of this, if running under a backwards compatibility mode SQAlchemy may attempt to use T-SQL statements that are unable to be parsed by the database server.

Known Issues

- No support for more than one `IDENTITY` column per table
- pymssql has problems with binary and unicode data that this module does **not** work around

8.3.6 MySQL

Support for the MySQL database.

Overview

For normal SQLAlchemy usage, importing this module is unnecessary. It will be loaded on-demand when a MySQL connection is needed. The generic column types like `String` and `Integer` will automatically be adapted to the optimal matching MySQL column type.

But if you would like to use one of the MySQL-specific or enhanced column types when creating tables with your `Table` definitions, then you will need to import them from this module:

```
from sqlalchemy.databases import mysql

Table('mytable', metadata,
      Column('id', Integer, primary_key=True),
      Column('ittybittyblob', mysql.MSTinyBlob),
      Column('biggy', mysql.MSBigInteger(unsigned=True)))
```

All standard MySQL column types are supported. The OpenGIS types are available for use via table reflection but have no special support or mapping to Python classes. If you're using these types and have opinions about how OpenGIS can be smartly integrated into SQLAlchemy please join the mailing list!

Supported Versions and Features

SQLAlchemy supports 6 major MySQL versions: 3.23, 4.0, 4.1, 5.0, 5.1 and 6.0, with capabilities increasing with more modern servers.

Versions 4.1 and higher support the basic SQL functionality that SQLAlchemy uses in the ORM and SQL expressions. These versions pass the applicable tests in the suite 100%. 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.

Currently, the only DB-API driver supported is *MySQL-Python* (also referred to as *MySQLdb*). Either 1.2.1 or 1.2.2 are recommended. The alpha, beta and gamma releases of 1.2.1 and 1.2.2 should be avoided. Support for Jython and IronPython is planned.

Feature	Minimum Version
sqlalchemy.orm	4.1.1
Table Reflection	3.23.x
DDL Generation	4.1.1
utf8/Full Unicode Connections	4.1.1
Transactions	3.23.15
Two-Phase Transactions	5.0.3
Nested Transactions	5.0.3

See the official MySQL documentation for detailed information about features supported in any given server release.

Character Sets

Many MySQL server installations default to a `latin1` encoding for client connections. All data sent through the connection will be converted into `latin1`, even if you have `utf8` or another character set on your tables and columns. With versions 4.1 and higher, you can change the connection character set either through server configuration or by including the `charset` parameter in the URL used for `create_engine`. The `charset` option is passed through to MySQL-Python and has the side-effect of also enabling `use_unicode` in the driver by default. For regular encoded strings, also pass `use_unicode=0` in the connection arguments:

```
# set client encoding to utf8; all strings come back as unicode
create_engine('mysql:///mydb?charset=utf8')

# set client encoding to utf8; all strings come back as utf8 str
create_engine('mysql:///mydb?charset=utf8&use_unicode=0')
```

Storage Engines

Most MySQL server installations have a default table type of `MyISAM`, a non-transactional table type. During a transaction, non-transactional storage engines do not participate and continue to store table changes in autocommit mode. For fully atomic transactions, all participating tables must use a transactional engine such as `InnoDB`, `Falcon`, `SolidDB`, *PBXT*, etc.

Storage engines can be elected when creating tables in SQLAlchemy by supplying a `mysql_engine='whatever'` to the `Table` constructor. Any MySQL table creation option can be specified in this syntax:

```
Table('mytable', metadata,
      Column('data', String(32)),
      mysql_engine='InnoDB',
      mysql_charset='utf8'
     )
```

Keys

Not all MySQL storage engines support foreign keys. For `MyISAM` and similar engines, the information loaded by table reflection will not include foreign keys. For these tables, you may supply a `ForeignKeyConstraint` at reflection time:

```
Table('mytable', metadata,
      ForeignKeyConstraint(['other_id'], ['othertable.other_id']),
      autoload=True
     )
```

When creating tables, SQLAlchemy will automatically set AUTO_INCREMENT` on an integer primary key column:

```
>>> t = Table('mytable', metadata,
...   Column('mytable_id', Integer, primary_key=True)
... )
>>> t.create()
CREATE TABLE mytable (
        id INTEGER NOT NULL AUTO_INCREMENT,
        PRIMARY KEY (id)
)
```

You can disable this behavior by supplying autoincrement=False to the Column. This flag can also be used to enable auto-increment on a secondary column in a multi-column key for some storage engines:

```
Table('mytable', metadata,
      Column('gid', Integer, primary_key=True, autoincrement=False),
      Column('id', Integer, primary_key=True)
     )
```

SQL Mode

MySQL SQL modes are supported. Modes that enable ANSI_QUOTES (such as ANSI) require an engine option to modify SQLAlchemy's quoting style. When using an ANSI-quoting mode, supply use_ansiquotes=True when creating your Engine:

```
create_engine('mysql://localhost/test', use_ansiquotes=True)
```

This is an engine-wide option and is not toggleable on a per-connection basis. SQLAlchemy does not presume to SET sql_mode for you with this option. For the best performance, set the quoting style server-wide in my.cnf or by supplying --sql-mode to mysqld. You can also use a sqlalchemy.pool.Pool listener hook to issue a SET SESSION sql_mode='...' on connect to configure each connection.

If you do not specify use_ansiquotes, the regular MySQL quoting style is used by default.

If you do issue a SET sql_mode through SQLAlchemy, the dialect must be updated if the quoting style is changed. Again, this change will affect all connections:

```
connection.execute('SET sql_mode="ansi"')
connection.dialect.use_ansiquotes = True
```

MySQL SQL Extensions

Many of the MySQL SQL extensions are handled through SQLAlchemy's generic function and operator support:

```
table.select(table.c.password==func.md5('plaintext'))
table.select(table.c.username.op('regexp')('^[a-d]'))
```

And of course any valid MySQL statement can be executed as a string as well.

Some limited direct support for MySQL extensions to SQL is currently available.

- SELECT pragma:

```
select(..., prefixes=['HIGH_PRIORITY', 'SQL_SMALL_RESULT'])
```

- UPDATE with LIMIT:

```
update(..., mysql_limit=10)
```

Troubleshooting

If you have problems that seem server related, first check that you are using the most recent stable MySQL-Python package available. The Database Notes page on the wiki at http://www.sqlalchemy.org is a good resource for timely information affecting MySQL in SQLAlchemy.

MySQL Column Types

class `MSNumeric`(*precision=10, scale=2, asdecimal=True, **kw*)

Bases: `sqlalchemy.types.Numeric`, `sqlalchemy.databases.mysql._NumericType`

MySQL NUMERIC type.

`__init__`(*precision=10, scale=2, asdecimal=True, **kw*)

Construct a NUMERIC.

Parameters
- *precision* – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- *scale* – The number of digits after the decimal point.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `MSDecimal`(*precision=10, scale=2, asdecimal=True, **kw*)

Bases: `sqlalchemy.databases.mysql.MSNumeric`

MySQL DECIMAL type.

`__init__`(*precision=10, scale=2, asdecimal=True, **kw*)

Construct a DECIMAL.

Parameters
- *precision* – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- *scale* – The number of digits after the decimal point.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `MSDouble`(*precision=None, scale=None, asdecimal=True, **kw*)

Bases: `sqlalchemy.types.Float`, `sqlalchemy.databases.mysql._NumericType`

MySQL DOUBLE type.

`__init__`(*precision=None, scale=None, asdecimal=True, **kw*)

Construct a DOUBLE.

Parameters
- *precision* – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- *scale* – The number of digits after the decimal point.
- *unsigned* – a boolean, optional.

- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class MSReal (*precision=None, scale=None, asdecimal=True, **kw*)

Bases: `sqlalchemy.databases.mysql.MSDouble`

MySQL REAL type.

__init__ (*precision=None, scale=None, asdecimal=True, **kw*)

Construct a REAL.

Parameters
- *precision* – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- *scale* – The number of digits after the decimal point.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class MSFloat (*precision=None, scale=None, asdecimal=False, **kw*)

Bases: `sqlalchemy.types.Float`, `sqlalchemy.databases.mysql._NumericType`

MySQL FLOAT type.

__init__ (*precision=None, scale=None, asdecimal=False, **kw*)

Construct a FLOAT.

Parameters
- *precision* – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- *scale* – The number of digits after the decimal point.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class MSInteger (*display_width=None, **kw*)

Bases: `sqlalchemy.types.Integer`, `sqlalchemy.databases.mysql._NumericType`

MySQL INTEGER type.

__init__ (*display_width=None, **kw*)

Construct an INTEGER.

Parameters
- *display_width* – Optional, maximum display width for this number.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class MSBigInteger (*display_width=None, **kw*)

Bases: `sqlalchemy.databases.mysql.MSInteger`

MySQL BIGINTEGER type.

__init__ (*display_width=None, **kw*)

Construct a BIGINTEGER.

Parameters
- *display_width* – Optional, maximum display width for this number.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `MSMediumInteger`(*display_width=None, **kw*)

Bases: `sqlalchemy.databases.mysql.MSInteger`

MySQL MEDIUMINTEGER type.

`__init__`(*display_width=None, **kw*)

Construct a MEDIUMINTEGER

Parameters
- *display_width* – Optional, maximum display width for this number.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `MSTinyInteger`(*display_width=None, **kw*)

Bases: `sqlalchemy.databases.mysql.MSInteger`

MySQL TINYINT type.

`__init__`(*display_width=None, **kw*)

Construct a TINYINT.

Note: following the usual MySQL conventions, TINYINT(1) columns reflected during Table(..., autoload=True) are treated as Boolean columns.

Parameters
- *display_width* – Optional, maximum display width for this number.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `MSSmallInteger`(*display_width=None, **kw*)

Bases: `sqlalchemy.types.SmallInteger`, `sqlalchemy.databases.mysql.MSInteger`

MySQL SMALLINTEGER type.

`__init__`(*display_width=None, **kw*)

Construct a SMALLINTEGER.

Parameters
- *display_width* – Optional, maximum display width for this number.
- *unsigned* – a boolean, optional.
- *zerofill* – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `MSBit`(*length=None*)

Bases: `sqlalchemy.types.TypeEngine`

MySQL BIT type.

This type is for MySQL 5.0.3 or greater for MyISAM, and 5.0.5 or greater for MyISAM, MEMORY, InnoDB and BDB. For older versions, use a MSTinyInteger() type.

`__init__`(*length=None*)

Construct a BIT.

Parameter *length* – Optional, number of bits.

class `MSDateTime`(*timezone=False*)

Bases: `sqlalchemy.types.DateTime`

MySQL DATETIME type.

`__init__`(*timezone=False*)

class `MSDate`(**args, **kwargs*)

Bases: `sqlalchemy.types.Date`

MySQL DATE type.

`__init__`(**args, **kwargs*)

class `MSTime`(*timezone=False*)

Bases: `sqlalchemy.types.Time`

MySQL TIME type.

`__init__`(*timezone=False*)

class `MSTimeStamp`(*timezone=False*)

Bases: `sqlalchemy.types.TIMESTAMP`

MySQL TIMESTAMP type.

To signal the orm to automatically re-select modified rows to retrieve the updated timestamp, add a `server_default` to your `Column` specification:

```
from sqlalchemy.databases import mysql
Column('updated', mysql.MSTimeStamp,
       server_default=sql.text('CURRENT_TIMESTAMP')
      )
```

The full range of MySQL 4.1+ TIMESTAMP defaults can be specified in the the default:

```
server_default=sql.text('CURRENT TIMESTAMP ON UPDATE CURRENT_TIMESTAMP')
```

`__init__`(*timezone=False*)

class `MSYear`(*display_width=None*)

Bases: `sqlalchemy.types.TypeEngine`

MySQL YEAR type, for single byte storage of years 1901-2155.

`__init__`(*display_width=None*)

class `MSText`(*length=None, **kwargs*)

Bases: `sqlalchemy.databases.mysql._StringType`, `sqlalchemy.types.Text`

MySQL TEXT type, for text up to 2^16 characters.

`__init__`(*length=None, **kwargs*)

Construct a TEXT.

Parameters
- *length* – Optional, if provided the server may optimize storage by substituting the smallest TEXT type sufficient to store `length` characters.
- *charset* – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- *collation* – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- *ascii* – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- *unicode* – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- *national* – Optional. If true, use the server's configured national character set.
- *binary* – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `MSTinyText`(***kwargs*)

Bases: `sqlalchemy.databases.mysql.MSText`

MySQL TINYTEXT type, for text up to 2^8 characters.

`__init__`(***kwargs*)
Construct a TINYTEXT.

Parameters
- *charset* – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- *collation* – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- *ascii* – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- *unicode* – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- *national* – Optional. If true, use the server's configured national character set.
- *binary* – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `MSMediumText`(***kwargs*)
Bases: `sqlalchemy.databases.mysql.MSText`

MySQL MEDIUMTEXT type, for text up to 2^24 characters.

`__init__`(***kwargs*)
Construct a MEDIUMTEXT.

Parameters
- *charset* – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- *collation* – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- *ascii* – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- *unicode* – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- *national* – Optional. If true, use the server's configured national character set.
- *binary* – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `MSLongText`(***kwargs*)
Bases: `sqlalchemy.databases.mysql.MSText`

MySQL LONGTEXT type, for text up to 2^32 characters.

`__init__`(***kwargs*)
Construct a LONGTEXT.

Parameters
- *charset* – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- *collation* – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- *ascii* – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- *unicode* – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- *national* – Optional. If true, use the server's configured national character set.
- *binary* – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class **`MSString`** (*length=None, **kwargs*)

Bases: `sqlalchemy.databases.mysql._StringType`, `sqlalchemy.types.String`

MySQL VARCHAR type, for variable-length character data.

`__init__` (*length=None, **kwargs*)

Construct a VARCHAR.

Parameters
- *charset* – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- *collation* – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- *ascii* – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- *unicode* – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- *national* – Optional. If true, use the server's configured national character set.
- *binary* – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class **`MSChar`** (*length, **kwargs*)

Bases: `sqlalchemy.databases.mysql._StringType`, `sqlalchemy.types.CHAR`

MySQL CHAR type, for fixed-length character data.

`__init__` (*length, **kwargs*)

Construct an NCHAR.

Parameters
- *length* – Maximum data length, in characters.
- *binary* – Optional, use the default binary collation for the national character set. This does not affect the type of data stored, use a BINARY type for binary data.
- *collation* – Optional, request a particular collation. Must be compatible with the national character set.

class **`MSNVarChar`** (*length=None, **kwargs*)

Bases: `sqlalchemy.databases.mysql._StringType`, `sqlalchemy.types.String`

MySQL NVARCHAR type.

For variable-length character data in the server's configured national character set.

`__init__` (*length=None, **kwargs*)

Construct an NVARCHAR.

Parameters
- *length* – Maximum data length, in characters.
- *binary* – Optional, use the default binary collation for the national character set. This does not affect the type of data stored, use a BINARY type for binary data.
- *collation* – Optional, request a particular collation. Must be compatible with the national character set.

class **`MSNChar`** (*length=None, **kwargs*)

Bases: `sqlalchemy.databases.mysql._StringType`, `sqlalchemy.types.CHAR`

MySQL NCHAR type.

For fixed-length character data in the server's configured national character set.

`__init__` (*length=None, **kwargs*)

Construct an NCHAR. Arguments are:

Parameters
- *length* – Maximum data length, in characters.
- *binary* – Optional, use the default binary collation for the national character set. This does not affect the type of data stored, use a BINARY type for binary data.

- *collation* – Optional, request a particular collation. Must be compatible with the national character set.

class **`MSVarBinary`**(*length=None, **kw*)

Bases: `sqlalchemy.databases.mysql._BinaryType`

MySQL VARBINARY type, for variable length binary data.

`__init__`(*length=None, **kw*)

Construct a VARBINARY. Arguments are:

Parameter *length* – Maximum data length, in characters.

class **`MSBinary`**(*length=None, **kw*)

Bases: `sqlalchemy.databases.mysql._BinaryType`

MySQL BINARY type, for fixed length binary data

`__init__`(*length=None, **kw*)

Construct a BINARY.

This is a fixed length type, and short values will be right-padded with a server-version-specific pad value.

Parameter *length* – Maximum data length, in bytes. If length is not specified, this will generate a BLOB. This usage is deprecated.

class **`MSBlob`**(*length=None, **kw*)

Bases: `sqlalchemy.databases.mysql._BinaryType`

MySQL BLOB type, for binary data up to 2^16 bytes

`__init__`(*length=None, **kw*)

Construct a BLOB. Arguments are:

Parameter *length* – Optional, if provided the server may optimize storage by substituting the smallest TEXT type sufficient to store `length` characters.

class **`MSTinyBlob`**(*length=None, **kw*)

Bases: `sqlalchemy.databases.mysql.MSBlob`

MySQL TINYBLOB type, for binary data up to 2^8 bytes.

`__init__`(*length=None, **kw*)

Construct a BLOB. Arguments are:

Parameter *length* – Optional, if provided the server may optimize storage by substituting the smallest TEXT type sufficient to store `length` characters.

class **`MSMediumBlob`**(*length=None, **kw*)

Bases: `sqlalchemy.databases.mysql.MSBlob`

MySQL MEDIUMBLOB type, for binary data up to 2^24 bytes.

`__init__`(*length=None, **kw*)

Construct a BLOB. Arguments are:

Parameter *length* – Optional, if provided the server may optimize storage by substituting the smallest TEXT type sufficient to store `length` characters.

class **`MSLongBlob`**(*length=None, **kw*)

Bases: `sqlalchemy.databases.mysql.MSBlob`

MySQL LONGBLOB type, for binary data up to 2^32 bytes.

`__init__`(*length=None, **kw*)

Construct a BLOB. Arguments are:

Parameter *length* – Optional, if provided the server may optimize storage by substituting the smallest TEXT type sufficient to store `length` characters.

class MSEnum (**enums, **kw*)

Bases: `sqlalchemy.databases.mysql.MSString`

MySQL ENUM type.

`__init__` (**enums, **kw*)

Construct an ENUM.

Example:

Column('myenum', MSEnum("foo", "bar", "baz"))

Arguments are:

Parameters
- *enums* – The range of valid values for this ENUM. Values will be quoted when generating the schema according to the quoting flag (see below).
- *strict* – Defaults to False: ensure that a given value is in this ENUM's range of permissible values when inserting or updating rows. Note that MySQL will not raise a fatal error if you attempt to store an out of range value- an alternate value will be stored instead. (See MySQL ENUM documentation.)
- *charset* – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- *collation* – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- *ascii* – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- *unicode* – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- *binary* – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.
- *quoting* – Defaults to 'auto': automatically determine enum value quoting. If all enum values are surrounded by the same quoting character, then use 'quoted' mode. Otherwise, use 'unquoted' mode.
  'quoted': values in enums are already quoted, they will be used directly when generating the schema.
  'unquoted': values in enums are not quoted, they will be escaped and surrounded by single quotes when generating the schema.
  Previous versions of this type always required manually quoted values to be supplied; future versions will always quote the string literals for you. This is a transitional option.

class MSSet (**values, **kw*)

Bases: `sqlalchemy.databases.mysql.MSString`

MySQL SET type.

`__init__` (**values, **kw*)

Construct a SET.

Example:

```
Column('myset', MSSet("'foo'", "'bar'", "'baz'"))
```

Arguments are:

Parameters
- *values* – The range of valid values for this SET. Values will be used exactly as they appear when generating schemas. Strings must be quoted, as in the example above. Single-quotes are suggested for ANSI compatibility and are required for portability to servers with ANSI_QUOTES enabled.
- *charset* – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.

- *collation* – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- *ascii* – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- *unicode* – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- *binary* – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class **`MSBoolean`**(**args, **kwargs*)

Bases: `sqlalchemy.types.Boolean`

MySQL BOOLEAN type.

`__init__`(**args, **kwargs*)

8.3.7 Oracle

Support for the Oracle database.

Oracle version 8 through current (11g at the time of this writing) are supported.

Driver

The Oracle dialect uses the cx_oracle driver, available at http://cx-oracle.sourceforge.net/ . The dialect has several behaviors which are specifically tailored towards compatibility with this module.

Connecting

Connecting with create_engine() uses the standard URL approach of `oracle://user:pass@host:port/dbname[?key=valu` If dbname is present, the host, port, and dbname tokens are converted to a TNS name using the cx_oracle `makedsn()` function. Otherwise, the host token is taken directly as a TNS name.

Additional arguments which may be specified either as query string arguments on the URL, or as keyword arguments to `create_engine()` are:

- *allow_twophase* - enable two-phase transactions. Defaults to `True`.
- *auto_convert_lobs* - defaults to True, see the section on LOB objects.
- *auto_setinputsizes* - the cx_oracle.setinputsizes() call is issued for all bind parameters. This is required for LOB datatypes but can be disabled to reduce overhead. Defaults to `True`.
- *mode* - This is given the string value of SYSDBA or SYSOPER, or alternatively an integer value. This value is only available as a URL query string argument.
- *threaded* - enable multithreaded access to cx_oracle connections. Defaults to `True`. Note that this is the opposite default of cx_oracle itself.
- *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.

Auto Increment Behavior

SQLAlchemy Table objects which include integer primary keys are usually assumed to have "autoincrementing" behavior, meaning they can generate their own primary key values upon INSERT. Since Oracle has no "autoincrement" feature, SQLAlchemy relies upon sequences to produce these values. With the Oracle dialect, *a sequence must always be explicitly specified to enable autoincrement*. This is divergent with the majority of documentation examples which assume the usage of an autoincrement-capable database. To specify sequences, use the sqlalchemy.schema.Sequence object which is passed to a Column construct:

```
t = Table('mytable', metadata,
      Column('id', Integer, Sequence('id_seq'), primary_key=True),
      Column(...), ...
)
```

This step is also required when using table reflection, i.e. autoload=True:

```
t = Table('mytable', metadata,
      Column('id', Integer, Sequence('id_seq'), primary_key=True),
      autoload=True
)
```

LOB Objects

cx_oracle presents some challenges when fetching LOB objects. A LOB object in a result set is presented by cx_oracle as a cx_oracle.LOB object which has a read() method. By default, SQLAlchemy converts these LOB objects into Python strings. This is for two reasons. First, the LOB object requires an active cursor association, meaning if you were to fetch many rows at once such that cx_oracle had to go back to the database and fetch a new batch of rows, the LOB objects in the already-fetched rows are now unreadable and will raise an error. SQLA "pre-reads" all LOBs so that their data is fetched before further rows are read. The size of a "batch of rows" is controlled by the cursor.arraysize value, which SQLAlchemy defaults to 50 (cx_oracle normally defaults this to one).

Secondly, the LOB object is not a standard DBAPI return value so SQLAlchemy seeks to "normalize" the results to look more like other DBAPIs.

The conversion of LOB objects by this dialect is unique in SQLAlchemy in that it takes place for all statement executions, even plain string-based statements for which SQLA has no awareness of result typing. This is so that calls like fetchmany() and fetchall() can work in all cases without raising cursor errors. The conversion of LOB in all cases, as well as the "prefetch" of LOB objects, can be disabled using auto_convert_lobs=False.

LIMIT/OFFSET Support

Oracle has no support for the LIMIT or OFFSET keywords. Whereas previous versions of SQLAlchemy used the "ROW NUMBER OVER..." construct to simulate LIMIT/OFFSET, SQLAlchemy 0.5 now uses a wrapped subquery approach in conjunction with ROWNUM. The exact methodology is taken from http://www.oracle.com/technology/oramag/oracle/06-sep/o56asktom.html . Note that the "FIRST ROWS()" optimization keyword mentioned is not used by default, as the user community felt this was stepping into the bounds of optimization that is better left on the DBA side, but this prefix can be added by enabling the optimize_limits=True flag on create_engine().

Two Phase Transaction Support

Two Phase transactions are implemented using XA transactions. Success has been reported of them working successfully but this should be regarded as an experimental feature.

Oracle 8 Compatibility

When using Oracle 8, a "use_ansi=False" flag is available which converts all JOIN phrases into the WHERE clause, and in the case of LEFT OUTER JOIN makes use of Oracle's (+) operator.

Synonym/DBLINK Reflection

When using reflection with Table objects, the dialect can optionally search for tables indicated by synonyms that reference DBLINK-ed tables by passing the flag oracle_resolve_synonyms=True as a keyword argument to the Table construct. If DBLINK is not in use this flag should be left off.

8.3.8 PostgreSQL

Support for the PostgreSQL database.

Driver

The psycopg2 driver is supported, available at http://pypi.python.org/pypi/psycopg2/ . The dialect has several behaviors which are specifically tailored towards compatibility with this module.

Note that psycopg1 is **not** supported.

Connecting

URLs are of the form *postgres://user:password@host:port/dbname[?key=value&key=value...]*.

Postgres-specific keyword arguments which are accepted by `create_engine()` are:

- *server_side_cursors* - Enable the usage of "server side cursors" for SQL statements which support this feature. What this essentially means from a psycopg2 point of view is that the cursor is created using a name, e.g. *connection.cursor('some name')*, which has the effect that result rows are not immediately pre-fetched and buffered after statement execution, but are instead left on the server and only retrieved as needed. SQLAlchemy's `ResultProxy` uses special row-buffering behavior when this feature is enabled, such that groups of 100 rows at a time are fetched over the wire to reduce conversational overhead.

Sequences/SERIAL

Postgres 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 corresponding to the column and associated with it based on a naming convention.

To specify a specific named sequence to be used for primary key generation, use the `Sequence()` construct:

```
Table('sometable', metadata,
        Column('id', Integer, Sequence('some_id_seq'), primary_key=True)
    )
```

Currently, when SQLAlchemy issues a single insert statement, to fulfill the contract of having the "last insert identifier" available, the sequence is executed independently beforehand and the new value is retrieved, to be used in the subsequent insert. Note that when an `insert()` construct is executed using "executemany" semantics, the sequence is not pre-executed and normal PG SERIAL behavior is used.

Postgres 8.3 supports an `INSERT...RETURNING` syntax which SQLAlchemy supports as well. A future release of SQLA will use this feature by default in lieu of sequence pre-execution in order to retrieve new primary key values, when available.

INSERT/UPDATE...RETURNING

The dialect supports PG 8.3's `INSERT..RETURNING` and `UPDATE..RETURNING` syntaxes, but must be explicitly enabled on a per-statement basis:

```
# INSERT..RETURNING
result = table.insert(postgres_returning=[table.c.col1, table.c.col2]).\
    values(name='foo')
print result.fetchall()

# UPDATE..RETURNING
result = table.update(postgres_returning=[table.c.col1, table.c.col2]).\
    where(table.c.name=='foo').values(name='bar')
print result.fetchall()
```

Indexes

PostgreSQL supports partial indexes. To create them pass a postgres_where option to the Index constructor:

```
Index('my_index', my_table.c.id, postgres_where=tbl.c.value > 10)
```

Transactions

The Postgres dialect fully supports SAVEPOINT and two-phase commit operations.

8.3.9 SQLite

Support for the SQLite database.

Driver

When using Python 2.5 and above, the built in `sqlite3` driver is already installed and no additional installation is needed. Otherwise, the `pysqlite2` driver needs to be present. This is the same driver as `sqlite3`, just with a different name.

The `pysqlite2` driver will be loaded first, and if not found, `sqlite3` is loaded. This allows an explicitly installed pysqlite driver to take precedence over the built in one. As with all dialects, a specific DBAPI module may be provided to `create_engine()` to control this explicitly:

```
from sqlite3 import dbapi2 as sqlite
e = create_engine('sqlite:///file.db', module=sqlite)
```

Full documentation on pysqlite is available at: http://www.initd.org/pub/software/pysqlite/doc/usage-guide.html

Connect Strings

The file specification for the SQLite database is taken as the "database" portion of the URL. Note that the format of a url is:

```
driver://user:pass@host/database
```

This means that the actual filename to be used starts with the characters to the **right** of the third slash. So connecting to a relative filepath looks like:

```
# relative path
e = create_engine('sqlite:///path/to/database.db')
```

An absolute path, which is denoted by starting with a slash, means you need **four** slashes:

```
# absolute path
e = create_engine('sqlite:////path/to/database.db')
```

To use a Windows path, regular drive specifications and backslashes can be used. Double backslashes are probably needed:

```
# absolute path on Windows
e = create_engine('sqlite:///C:\\path\\to\\database.db')
```

The sqlite `:memory:` identifier is the default if no filepath is present. Specify `sqlite://` and nothing else:

```
# in-memory database
e = create_engine('sqlite://')
```

Threading Behavior

Pysqlite connections do not support being moved between threads, unless the `check_same_thread` Pysqlite flag is set to `False`. In addition, when using an in-memory SQLite database, the full database exists only within the scope of a single connection. It is reported that an in-memory database does not support being shared between threads regardless of the `check_same_thread` flag - which means that a multithreaded application **cannot** share data from a `:memory:` database across threads unless access to the connection is limited to a single worker thread which communicates through a queueing mechanism to concurrent threads.

To provide a default which accomodates SQLite's default threading capabilities somewhat reasonably, the SQLite dialect will specify that the `SingletonThreadPool` be used by default. This pool maintains a single SQLite connection per thread that is held open up to a count of five concurrent threads. When more than five threads are used, a cleanup mechanism will dispose of excess unused connections.

Two optional pool implementations that may be appropriate for particular SQLite usage scenarios:

- the `sqlalchemy.pool.StaticPool` might be appropriate for a multithreaded application using an in-memory database, assuming the threading issues inherent in pysqlite are somehow accomodated for. This pool holds persistently onto a single connection which is never closed, and is returned for all requests.
- the `sqlalchemy.pool.NullPool` might be appropriate for an application that makes use of a file-based sqlite database. This pool disables any actual "pooling" behavior, and simply opens and closes real connections corresonding to the `connect()` and `close()` methods. SQLite can "connect" to a particular file with very high efficiency, so this option may actually perform better without the extra overhead of `SingletonThreadPool`. NullPool will of course render a `:memory:` connection useless since the database would be lost as soon as the connection is "returned" to the pool.

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 `SLDateTime`, `SLDate` and `SLTime`. 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.

Unicode

In contrast to SQLAlchemy's active handling of date and time types for pysqlite, pysqlite's default behavior regarding Unicode is that all strings are returned as Python unicode objects in all cases. So even if the `Unicode` type is *not* used, you will still always receive unicode data back from a result set. It is **strongly** recommended that you do use the `Unicode` type to represent strings, since it will raise a warning if a non-unicode Python string is passed from the user application. Mixing the usage of non-unicode objects with returned unicode objects can quickly create confusion, particularly when using the ORM as internal data is not always represented by an actual database result string.

8.3.10 Sybase

Sybase database backend.

Known issues / TODO:

- Uses the mx.ODBC driver from egenix (version 2.1.0)
- The current version of sqlalchemy.databases.sybase only supports mx.ODBC.Windows (other platforms such as mx.ODBC.unixODBC still need some development)
- Support for pyodbc has been built in but is not yet complete (needs further development)
- **Results of running tests/alltests.py:** Ran 934 tests in 287.032s FAILED (failures=3, errors=1)
- Tested on 'Adaptive Server Anywhere 9' (version 9.0.1.1751)

8.4 sqlalchemy.ext

SQLAlchemy has a variety of extensions available which provide extra functionality to SA, either via explicit usage or by augmenting the core behavior.

8.4.1 declarative

A simple declarative layer for SQLAlchemy ORM.

Synopsis

SQLAlchemy object-relational configuration involves the usage of `Table`, `mapper()`, and class objects to define the three areas of configuration. `declarative` moves these three types of configuration underneath the individual mapped class. Regular SQLAlchemy schema and ORM constructs are used in most cases:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class SomeClass(Base):
    __tablename__ = 'some_table'
    id = Column(Integer, primary_key=True)
    name =  Column(String(50))
```

Above, the `declarative_base()` callable produces a new base class from which all mapped classes inherit from. When the class definition is completed, a new `Table` and `mapper` have been generated, accessible via the `__table__` and `__mapper__` attributes on the `SomeClass` class.

Defining Attributes

`Column` objects may be explicitly named, including using a different name than the attribute in which they are associated. The column will be assigned to the `Table` using the given name, and mapped to the class using the attribute name:

```
class SomeClass(Base):
    __tablename__ = 'some_table'
    id = Column("some_table_id", Integer, primary_key=True)
    name = Column("name", String(50))
```

Otherwise, you may omit the names from the Column definitions. Declarative will set the `name` attribute on the column when the class is initialized:

```
class SomeClass(Base):
    __tablename__ = 'some_table'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

Attributes may be added to the class after its construction, and they will be added to the underlying `Table` and `mapper()` definitions as appropriate:

```
SomeClass.data = Column('data', Unicode)
SomeClass.related = relation(RelatedInfo)
```

Classes which are mapped explicitly using `mapper()` can interact freely with declarative classes. It is recommended, though not required, that all tables share the same underlying `MetaData` object, so that string-configured `ForeignKey` references can be resolved without issue.

Association of Metadata and Engine

The `declarative_base()` base class contains a `MetaData` object where newly defined `Table` objects are collected. This is accessed via the `MetaData` class level accessor, so to create tables we can say:

```
engine = create_engine('sqlite://')
Base.metadata.create_all(engine)
```

The Engine created above may also be directly associated with the declarative base class using the bind keyword argument, where it will be associated with the underlying MetaData object and allow SQL operations involving that metadata and its tables to make use of that engine automatically:

```
Base = declarative_base(bind=create_engine('sqlite://'))
```

Or, as MetaData allows, at any time using the bind attribute:

```
Base.metadata.bind = create_engine('sqlite://')
```

The declarative_base() can also receive a pre-created MetaData object, which allows a declarative setup to be associated with an already existing traditional collection of Table objects:

```
mymetadata = MetaData()
Base = declarative_base(metadata=mymetadata)
```

Configuring Relations

Relations to other classes are done in the usual way, with the added feature that the class specified to relation() may be a string name. The "class registry" associated with Base is used at mapper compilation time to resolve the name into the actual class object, which is expected to have been defined once the mapper configuration is used:

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

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    addresses = relation("Address", backref="user")

class Address(Base):
    __tablename__ = 'addresses'

    id = Column(Integer, primary_key=True)
    email = Column(String(50))
    user_id = Column(Integer, ForeignKey('users.id'))
```

Column constructs, since they are just that, are immediately usable, as below where we define a primary join condition on the Address class using them:

```
class Address(Base):
    __tablename__ = 'addresses'

    id = Column(Integer, primary_key=True)
    email = Column(String(50))
    user_id = Column(Integer, ForeignKey('users.id'))
    user = relation(User, primaryjoin=user_id == User.id)
```

In addition to the main argument for relation(), other arguments which depend upon the columns present on an as-yet undefined class may also be specified as strings. These strings are evaluated as Python expressions. The full namespace available within this evaluation includes all classes mapped for this declarative base, as well as the contents of the sqlalchemy package, including expression functions like desc() and func:

```
class User(Base):
    # ....
    addresses = relation("Address", order_by="desc(Address.email)",
        primaryjoin="Address.user_id==User.id")
```

As an alternative to string-based attributes, attributes may also be defined after all classes have been created. Just add them to the target class after the fact:

```
User.addresses = relation(Address, primaryjoin=Address.user_id == User.id)
```

Configuring Many-to-Many Relations

There's nothing special about many-to-many with declarative. The `secondary` argument to `relation()` still requires a `Table` object, not a declarative class. The `Table` should share the same `MetaData` object used by the declarative base:

```
keywords = Table('keywords', Base.metadata,
                    Column('author_id', Integer, ForeignKey('authors.id')),
                    Column('keyword_id', Integer, ForeignKey('keywords.id'))
            )

class Author(Base):
    __tablename__ = 'authors'
    id = Column(Integer, primary_key=True)
    keywords = relation("Keyword", secondary=keywords)
```

You should generally **not** map a class and also specify its table in a many-to-many relation, since the ORM may issue duplicate INSERT and DELETE statements.

Defining Synonyms

Synonyms are introduced in *Using Descriptors*. To define a getter/setter which proxies to an underlying attribute, use `synonym()` with the `descriptor` argument:

```
class MyClass(Base):
    __tablename__ = 'sometable'

    _attr = Column('attr', String)

    def _get_attr(self):
        return self._some_attr
    def _set_attr(self, attr):
        self._some_attr = attr
    attr = synonym('_attr', descriptor=property(_get_attr, _set_attr))
```

The above synonym is then usable as an instance attribute as well as a class-level expression construct:

```
x = MyClass()
x.attr = "some value"
session.query(MyClass).filter(MyClass.attr == 'some other value').all()
```

For simple getters, the `synonym_for()` decorator can be used in conjunction with `@property`:

```
class MyClass(Base):
    __tablename__ = 'sometable'

    _attr = Column('attr', String)

    @synonym_for('_attr')
    @property
    def attr(self):
        return self._some_attr
```

Similarly, `comparable_using()` is a front end for the `comparable_property()` ORM function:

```
class MyClass(Base):
    __tablename__ = 'sometable'

    name = Column('name', String)

    @comparable_using(MyUpperCaseComparator)
    @property
    def uc_name(self):
        return self.name.upper()
```

Table Configuration

As an alternative to `__tablename__`, a direct `Table` construct may be used. The `Column` objects, which in this case require their names, will be added to the mapping just like a regular mapping to a table:

```
class MyClass(Base):
    __table__ = Table('my_table', Base.metadata,
        Column('id', Integer, primary_key=True),
        Column('name', String(50))
    )
```

Other table-based attributes include `__table_args__`, which is either a dictionary as in:

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = {'mysql_engine':'InnoDB'}
```

or a dictionary-containing tuple in the form `(arg1, arg2, ..., {kwarg1:value, ...})`, as in:

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = (ForeignKeyConstraint(['id'], ['remote_table.id']), {'autoload':True}
```

Mapper Configuration

Mapper arguments are specified using the `__mapper_args__` class variable, which is a dictionary that accepts the same names as the `mapper` function accepts as keywords:

```
class Widget(Base):
    __tablename__ = 'widgets'
    id = Column(Integer, primary_key=True)

    __mapper_args__ = {'extension': MyWidgetExtension()}
```

Inheritance Configuration

Declarative supports all three forms of inheritance as intuitively as possible. The `inherits` mapper keyword argument is not needed, as declarative will determine this from the class itself. The various "polymorphic" keyword arguments are specified using `__mapper_args__`.

Joined Table Inheritance

Joined table inheritance is defined as a subclass that defines its own table:

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = 'engineers'
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    id = Column(Integer, ForeignKey('people.id'), primary_key=True)
    primary_language = Column(String(50))
```

Note that above, the `Engineer.id` attribute, since it shares the same attribute name as the `Person.id` attribute, will in fact represent the `people.id` and `engineers.id` columns together, and will render inside a query as `"people.id"`. To provide the `Engineer` class with an attribute that represents only the `engineers.id` column, give it a different attribute name:

```
class Engineer(Person):
    __tablename__ = 'engineers'
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    engineer_id = Column('id', Integer, ForeignKey('people.id'), primary_key=True)
    primary_language = Column(String(50))
```

Single Table Inheritance

Single table inheritance is defined as a subclass that does not have its own table; you just leave out the `__table__` and `__tablename__` attributes:

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}
```

```
class Engineer(Person):
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    primary_language = Column(String(50))
```

When the above mappers are configured, the `Person` class is mapped to the `people` table *before* the `primary_language` column is defined, and this column will not be included in its own mapping. When `Engineer` then defines the `primary_language` column, the column is added to the `people` table so that it is included in the mapping for `Engineer` and is also part of the table's full set of columns. Columns which are not mapped to `Person` are also excluded from any other single or joined inheriting classes using the `exclude_properties` mapper argument. Below, `Manager` will have all the attributes of `Person` and `Manager` but *not* the `primary_language` attribute of `Engineer`:

```
class Manager(Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}
    golf_swing = Column(String(50))
```

The attribute exclusion logic is provided by the `exclude_properties` mapper argument, and declarative's default behavior can be disabled by passing an explicit `exclude_properties` collection (empty or otherwise) to the `__mapper_args__`.

Concrete Table Inheritance

Concrete is defined as a subclass which has its own table and sets the `concrete` keyword argument to `True`:

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

class Engineer(Person):
    __tablename__ = 'engineers'
    __mapper_args__ = {'concrete':True}
    id = Column(Integer, primary_key=True)
    primary_language = Column(String(50))
    name = Column(String(50))
```

Usage of an abstract base class is a little less straightforward as it requires usage of `polymorphic_union()`:

```
engineers = Table('engineers', Base.metadata,
                Column('id', Integer, primary_key=True),
                Column('name', String(50)),
                Column('primary_language', String(50))
            )
managers = Table('managers', Base.metadata,
                Column('id', Integer, primary_key=True),
                Column('name', String(50)),
                Column('golf_swing', String(50))
            )

punion = polymorphic_union({
    'engineer':engineers,
    'manager':managers
}, 'type', 'punion')

class Person(Base):
    __table__ = punion
    __mapper_args__ = {'polymorphic_on':punion.c.type}

class Engineer(Person):
    __table__ = engineers
    __mapper_args__ = {'polymorphic_identity':'engineer', 'concrete':True}

class Manager(Person):
    __table__ = managers
    __mapper_args__ = {'polymorphic_identity':'manager', 'concrete':True}
```

Class Usage

As a convenience feature, the `declarative_base()` sets a default constructor on classes which takes keyword arguments, and assigns them to the named attributes:

```
e = Engineer(primary_language='python')
```

Note that `declarative` has no integration built in with sessions, and is only intended as an optional syntax for the regular usage of mappers and Table objects. A typical application setup using `scoped_session()` might look like:

```
engine = create_engine('postgres://scott:tiger@localhost/test')
Session = scoped_session(sessionmaker(autocommit=False,
                                      autoflush=False,
                                      bind=engine))
Base = declarative_base()
```

Mapped instances then make usage of `Session` in the usual way.

`declarative_base`(*bind=None, metadata=None, mapper=None, cls=<type 'object'>, name='Base', constructor=<function __init__ at 0x533adb0>, metaclass=<class 'sqlalchemy.ext.declarative.DeclarativeMeta'>, engine=None*)

Construct a base class for declarative class definitions.

The new base class will be given a metaclass that invokes `instrument_declarative()` upon each subclass definition, and routes later Column- and Mapper-related attribute assignments made on the class into Table and Mapper assignments.

Parameters

- *bind* – An optional `Connectable`, will be assigned the `bind` attribute on the `MetaData` instance. The *engine* keyword argument is a deprecated synonym for *bind*.
- *metadata* – An optional `MetaData` instance. All `Table` objects implicitly declared by subclasses of the base will share this MetaData. A MetaData instance will be create if none is provided. The MetaData instance will be available via the *metadata* attribute of the generated declarative base class.
- *mapper* – An optional callable, defaults to `mapper()`. Will be used to map subclasses to their Tables.
- *cls* – Defaults to `object`. A type to use as the base for the generated declarative base class. May be a type or tuple of types.
- *name* – Defaults to `Base`. The display name for the generated class. Customizing this is not required, but can improve clarity in tracebacks and debugging.

- *constructor* – Defaults to declarative._declarative_constructor, an __init__ implementation that assigns **kwargs for declared fields and relations to an instance. If `None` is supplied, no __init__ will be installed and construction will fall back to cls.__init__ with normal Python semantics.
- *metaclass* – Defaults to `DeclarativeMeta`. A metaclass or __metaclass__ compatible callable to use as the meta type of the generated declarative base class.

synonym_for(*name, map_column=False*)

Decorator, make a Python @property a query synonym for a column.

A decorator version of `synonym()`. The function being decorated is the 'descriptor', otherwise passes its arguments through to synonym():

```
@synonym_for('col')
@property
def prop(self):
    return 'special sauce'
```

The regular `synonym()` is also usable directly in a declarative setting and may be convenient for read/write properties:

```
prop = synonym('col', descriptor=property(_read_prop, _write_prop))
```

comparable_using(*comparator_factory*)

Decorator, allow a Python @property to be used in query criteria.

A decorator front end to `comparable_property()`, passes through the comparator_factory and the function being decorated:

```
@comparable_using(MyComparatorType)
@property
def prop(self):
    return 'special sauce'
```

The regular `comparable_property()` is also usable directly in a declarative setting and may be convenient for read/write properties:

```
prop = comparable_property(MyComparatorType)
```

instrument_declarative(*cls, registry, metadata*)

Given a class, configure the class declaratively, using the given registry (any dictionary) and MetaData object. This operation does not assume any kind of class hierarchy.

8.4.2 associationproxy

`associationproxy` is used to create a simplified, read/write view of a relationship. It can be used to cherry-pick fields from a collection of related objects or to greatly simplify access to associated objects in an association relationship.

Simplifying Relations

Consider this "association object" mapping:

```
users_table = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(64)),
)

keywords_table = Table('keywords', metadata,
    Column('id', Integer, primary_key=True),
    Column('keyword', String(64))
)

userkeywords_table = Table('userkeywords', metadata,
    Column('user_id', Integer, ForeignKey("users.id"),
           primary_key=True),
    Column('keyword_id', Integer, ForeignKey("keywords.id"),
           primary_key=True)
)

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

class Keyword(object):
    def __init__(self, keyword):
        self.keyword = keyword

mapper(User, users_table, properties={
    'kw': relation(Keyword, secondary=userkeywords_table)
    })
mapper(Keyword, keywords_table)
```

Above are three simple tables, modeling users, keywords and a many-to-many relationship between the two. These `Keyword` objects are little more than a container for a name, and accessing them via the relation is awkward:

```
user = User('jek')
user.kw.append(Keyword('cheese inspector'))
print user.kw
# [<__main__.Keyword object at 0xb791ea0c>]
print user.kw[0].keyword
# 'cheese inspector'
print [keyword.keyword for keyword in user.kw]
# ['cheese inspector']
```

With `association_proxy` you have a "view" of the relation that contains just the `.keyword` of the related objects. The proxy is a Python property, and unlike the mapper relation, is defined in your class:

```
from sqlalchemy.ext.associationproxy import association_proxy

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

    # proxy the 'keyword' attribute from the 'kw' relation
    keywords = association_proxy('kw', 'keyword')
```

```
# ...
>>> user.kw
[<__main__.Keyword object at 0xb791ea0c>]
>>> user.keywords
['cheese inspector']
>>> user.keywords.append('snack ninja')
>>> user.keywords
['cheese inspector', 'snack ninja']
>>> user.kw
[<__main__.Keyword object at 0x9272a4c>, <__main__.Keyword object at 0xb7b396ec>]
```

The proxy is read/write. New associated objects are created on demand when values are added to the proxy, and modifying or removing an entry through the proxy also affects the underlying collection.

- The association proxy property is backed by a mapper-defined relation, either a collection or scalar.
- You can access and modify both the proxy and the backing relation. Changes in one are immediate in the other.
- The proxy acts like the type of the underlying collection. A list gets a list-like proxy, a dict a dict-like proxy, and so on.
- Multiple proxies for the same relation are fine.
- Proxies are lazy, and won't trigger a load of the backing relation until they are accessed.
- The relation is inspected to determine the type of the related objects.
- To construct new instances, the type is called with the value being assigned, or key and value for dicts.
- A ``creator`` function can be used to create instances instead.

Above, the `Keyword.__init__` takes a single argument `keyword`, which maps conveniently to the value being set through the proxy. A `creator` function could have been used instead if more flexibility was required.

Because the proxies are backed by a regular relation collection, all of the usual hooks and patterns for using collections are still in effect. The most convenient behavior is the automatic setting of "parent"-type relationships on assignment. In the example above, nothing special had to be done to associate the Keyword to the User. Simply adding it to the collection is sufficient.

Simplifying Association Object Relations

Association proxies are also useful for keeping `association objects` out the way during regular use. For example, the `userkeywords` table might have a bunch of auditing columns that need to get updated when changes are made- columns that are updated but seldom, if ever, accessed in your application. A proxy can provide a very natural access pattern for the relation.

```
from sqlalchemy.ext.associationproxy import association_proxy

# users_table and keywords_table tables as above, then:

def get_current_uid():
    """Return the uid of the current user."""
    return 1  # hardcoded for this example

userkeywords_table = Table('userkeywords', metadata,
    Column('user_id', Integer, ForeignKey("users.id"), primary_key=True),
    Column('keyword_id', Integer, ForeignKey("keywords.id"), primary_key=True),
    # add some auditing columns
    Column('updated_at', DateTime, default=datetime.now),
    Column('updated_by', Integer, default=get_current_uid, onupdate=get_current_uid),
)

def _create_uk_by_keyword(keyword):
    """A creator function."""
    return UserKeyword(keyword=keyword)

class User(object):
    def __init__(self, name):
        self.name = name
    keywords = association_proxy('user_keywords', 'keyword', creator=_create_uk_by_keyword)

class Keyword(object):
    def __init__(self, keyword):
        self.keyword = keyword
    def __repr__(self):
        return 'Keyword(%s)' % repr(self.keyword)

class UserKeyword(object):
    def __init__(self, user=None, keyword=None):
        self.user = user
        self.keyword = keyword

mapper(User, users_table)
mapper(Keyword, keywords_table)
mapper(UserKeyword, userkeywords_table, properties={
    'user': relation(User, backref='user_keywords'),
    'keyword': relation(Keyword),
})

user = User('log')
kw1  = Keyword('new_from_blammo')

# Adding a Keyword requires creating a UserKeyword association object
user.user_keywords.append(UserKeyword(user, kw1))

# And accessing Keywords requires traversing UserKeywords
print user.user_keywords[0]
# <__main__.UserKeyword object at 0xb79bbbec>

print user.user_keywords[0].keyword
# Keyword('new_from_blammo')

# Lots of work.

# It's much easier to go through the association proxy!
for kw in (Keyword('its_big'), Keyword('its_heavy'), Keyword('its_wood')):
    user.keywords.append(kw)

print user.keywords
```

```
# [Keyword('new_from_blammo'), Keyword('its_big'), Keyword('its_heavy'), Keyword('its_wood
```

Building Complex Views

```
stocks_table = Table("stocks", meta,
   Column('symbol', String(10), primary_key=True),
   Column('last_price', Numeric)
)

brokers_table = Table("brokers", meta,
   Column('id', Integer,primary_key=True),
   Column('name', String(100), nullable=False)
)

holdings_table = Table("holdings", meta,
  Column('broker_id', Integer, ForeignKey('brokers.id'), primary_key=True),
  Column('symbol', String(10), ForeignKey('stocks.symbol'), primary_key=True),
  Column('shares', Integer)
)
```

Above are three tables, modeling stocks, their brokers and the number of shares of a stock held by each broker. This situation is quite different from the association example above. `shares` is a *property of the relation*, an important one that we need to use all the time.

For this example, it would be very convenient if `Broker` objects had a dictionary collection that mapped `Stock` instances to the shares held for each. That's easy:

```
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.orm.collections import attribute_mapped_collection

def _create_holding(stock, shares):
    """A creator function, constructs Holdings from Stock and share quantity."""
    return Holding(stock=stock, shares=shares)

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

    holdings = association_proxy('by_stock', 'shares', creator=_create_holding)

class Stock(object):
    def __init__(self, symbol):
        self.symbol = symbol
        self.last_price = 0

class Holding(object):
    def __init__(self, broker=None, stock=None, shares=0):
        self.broker = broker
        self.stock = stock
        self.shares = shares

mapper(Stock, stocks_table)
mapper(Broker, brokers_table, properties={
    'by_stock': relation(Holding,
        collection_class=attribute_mapped_collection('stock'))
})
mapper(Holding, holdings_table, properties={
    'stock': relation(Stock),
    'broker': relation(Broker)
})
```

Above, we've set up the `by_stock` relation collection to act as a dictionary, using the `.stock` property of each Holding as a key.

Populating and accessing that dictionary manually is slightly inconvenient because of the complexity of the Holdings association object:

```
stock = Stock('ZZK')
broker = Broker('paj')

broker.by_stock[stock] = Holding(broker, stock, 10)
print broker.by_stock[stock].shares
# 10
```

The `holdings` proxy we've added to the `Broker` class hides the details of the `Holding` while also giving access to `.shares`:

```
for stock in (Stock('JEK'), Stock('STPZ')):
    broker.holdings[stock] = 123

for stock, shares in broker.holdings.items():
    print stock, shares

session.add(broker)
session.commit()

# lets take a peek at that holdings_table after committing changes to the db
print list(holdings_table.select().execute())
# [(1, 'ZZK', 10), (1, 'JEK', 123), (1, 'STEPZ', 123)]
```

Further examples can be found in the `examples/` directory in the SQLAlchemy distribution.

API

`association_proxy`(*target_collection, attr, **kw*)

Return a Python property implementing a view of *attr* over a collection.

Implements a read/write view over an instance's *target_collection*, extracting *attr* from each member of the collection. The property acts somewhat like this list comprehension:

```
[getattr(member, *attr*)
 for member in getattr(instance, *target_collection*)]
```

Unlike the list comprehension, the collection returned by the property is always in sync with *target_collection*, and mutations made to either collection will be reflected in both.

Implements a Python property representing a relation as a collection of simpler values. The proxied property will mimic the collection type of the target (list, dict or set), or, in the case of a one to one relation, a simple scalar value.

Parameters
- *target_collection* – Name of the relation attribute we'll proxy to, usually created with `relation()`.
- *attr* – Attribute on the associated instances we'll proxy for.
  For example, given a target collection of [obj1, obj2], a list created by this proxy property would look like [getattr(obj1, *attr*), getattr(obj2, *attr*)]
  If the relation is one-to-one or otherwise uselist=False, then simply: getattr(obj, *attr*)
- *creator* – optional.
  When new items are added to this proxied collection, new instances of the class collected by the target collection will be created. For list and set collections, the target class constructor will be called with the 'value' for the new instance. For dict types, two arguments are passed: key and value.
  If you want to construct instances differently, supply a *creator* function that takes arguments as above and returns instances.
  For scalar relations, creator() will be called if the target is None. If the target is present, set operations are proxied to setattr() on the associated object.
  If you have an associated object with multiple attributes, you may set up multiple association proxies mapping to different attributes. See the unit tests for examples, and for examples of how creator() functions can be used to construct the scalar relation on-demand in this situation.
- ***kw* – Passes along any other keyword arguments to `AssociationProxy`.

class `AssociationProxy`(*target_collection, attr, creator=None, getset_factory=None, proxy_factory=None, proxy_bulk_set=None*)

A descriptor that presents a read/write view of an object attribute.

`__init__`(*target_collection, attr, creator=None, getset_factory=None, proxy_factory=None, proxy_bulk_set=None*)

Arguments are:

target_collection Name of the collection we'll proxy to, usually created with 'relation()' in a mapper setup.

attr Attribute on the collected instances we'll proxy for. For example, given a target collection of [obj1, obj2], a list created by this proxy property would look like [getattr(obj1, attr), getattr(obj2, attr)]

creator Optional. When new items are added to this proxied collection, new instances of the class collected by the target collection will be created. For list and set collections, the target class constructor will be called with the 'value' for the new instance. For dict types, two arguments are passed: key and value.

If you want to construct instances differently, supply a 'creator' function that takes arguments as above and returns instances.

getset_factory Optional. Proxied attribute access is automatically handled by routines that get and set values based on the *attr* argument for this proxy.

If you would like to customize this behavior, you may supply a *getset_factory* callable that produces a tuple of *getter* and *setter* functions. The factory is called with two arguments, the abstract type of the underlying collection and this proxy instance.

proxy_factory Optional. The type of collection to emulate is determined by sniffing the target collection. If your collection type can't be determined by duck typing or you'd like to use a different collection implementation, you may supply a factory function to produce those collections. Only applicable to non-scalar relations.

proxy_bulk_set Optional, use with proxy_factory. See the _set() method for details.

`target_class`

The class the proxy is attached to.

8.4.3 orderinglist

author Jason Kirtland

`orderinglist` is a helper for mutable ordered relations. It will intercept list operations performed on a relation collection and automatically synchronize changes in list position with an attribute on the related objects. (See *Custom Collection Implementations* for more information on the general pattern.)

Example: Two tables that store slides in a presentation. Each slide has a number of bullet points, displayed in order by the 'position' column on the bullets table. These bullets can be inserted and re-ordered by your end users, and you need to update the 'position' column of all affected rows when changes are made.

```
slides_table = Table('Slides', metadata,
                     Column('id', Integer, primary_key=True),
                     Column('name', String))

bullets_table = Table('Bullets', metadata,
                      Column('id', Integer, primary_key=True),
                      Column('slide_id', Integer, ForeignKey('Slides.id')),
                      Column('position', Integer),
                      Column('text', String))

 class Slide(object):
     pass
 class Bullet(object):
     pass

 mapper(Slide, slides_table, properties={
       'bullets': relation(Bullet, order_by=[bullets_table.c.position])
 })
 mapper(Bullet, bullets_table)
```

The standard relation mapping will produce a list-like attribute on each Slide containing all related Bullets, but coping with changes in ordering is totally your responsibility. If you insert a Bullet into that list, there is no magic- it won't have a position attribute unless you assign it it one, and you'll need to manually renumber all the subsequent Bullets in the list to accommodate the insert.

An `orderinglist` can automate this and manage the 'position' attribute on all related bullets for you.

```
mapper(Slide, slides_table, properties={
       'bullets': relation(Bullet,
                           collection_class=ordering_list('position'),
                           order_by=[bullets_table.c.position])
})
mapper(Bullet, bullets_table)

s = Slide()
s.bullets.append(Bullet())
s.bullets.append(Bullet())
s.bullets[1].position
>>> 1
s.bullets.insert(1, Bullet())
s.bullets[2].position
>>> 2
```

Use the `ordering_list` function to set up the `collection_class` on relations (as in the mapper example above). This implementation depends on the list starting in the proper order, so be SURE to put an order_by on your relation.

`ordering_list` takes the name of the related object's ordering attribute as an argument. By default, the zero-based integer index of the object's position in the `ordering_list` is synchronized with the ordering attribute: index 0 will get position 0, index 1 position 1, etc. To start numbering at 1 or some other integer, provide `count_from=1`.

Ordering values are not limited to incrementing integers. Almost any scheme can implemented by supplying a custom `ordering_func` that maps a Python list index to any value you require. See the [module documentation](rel:docstrings_sqlalchemy.ext.orderinglist) for more information, and also check out the unit tests for examples of stepped numbering, alphabetical and Fibonacci numbering. A custom list that manages index/position information for its children.

`orderinglist` is a custom list collection implementation for mapped relations that keeps an arbitrary "position" attribute on contained objects in sync with each object's position in the Python list.

The collection acts just like a normal Python `list`, with the added behavior that as you manipulate the list (via `insert`, `pop`, assignment, deletion, what have you), each of the objects it contains is updated as needed to reflect its position. This is very useful for managing ordered relations which have a user-defined, serialized order:

```
>>> from sqlalchemy import MetaData, Table, Column, Integer, String, ForeignKey
>>> from sqlalchemy.orm import mapper, relation
>>> from sqlalchemy.ext.orderinglist import ordering_list
```

A simple model of users their "top 10" things:

```
>>> metadata = MetaData()
>>> users = Table('users', metadata,
...               Column('id', Integer, primary_key=True))
>>> blurbs = Table('user_top_ten_list', metadata,
...               Column('id', Integer, primary_key=True),
...               Column('user_id', Integer, ForeignKey('users.id')),
...               Column('position', Integer),
...               Column('blurb', String(80)))
>>> class User(object):
...   pass
...
>>> class Blurb(object):
...    def __init__(self, blurb):
...        self.blurb = blurb
...
>>> mapper(User, users, properties={
...  'topten': relation(Blurb, collection_class=ordering_list('position'),
...                           order_by=[blurbs.c.position])})
<Mapper ...>
>>> mapper(Blurb, blurbs)
<Mapper ...>
```

Acts just like a regular list:

```
>>> u = User()
>>> u.topten.append(Blurb('Number one!'))
>>> u.topten.append(Blurb('Number two!'))
```

But the `.position` attibute is set automatically behind the scenes:

```
>>> assert [blurb.position for blurb in u.topten] == [0, 1]
```

The objects will be renumbered automaticaly after any list-changing operation, for example an `insert()`:

```
>>> u.topten.insert(1, Blurb('I am the new Number Two.'))
>>> assert [blurb.position for blurb in u.topten] == [0, 1, 2]
>>> assert u.topten[1].blurb == 'I am the new Number Two.'
>>> assert u.topten[1].position == 1
```

Numbering and serialization are both highly configurable. See the docstrings in this module and the main SQLAlchemy documentation for more information and examples.

The `ordering_list` factory function is the ORM-compatible constructor for *OrderingList* instances.

`ordering_list`(*attr, count_from=None, **kw*)

Prepares an OrderingList factory for use in mapper definitions.

Returns an object suitable for use as an argument to a Mapper relation's `collection_class` option. Arguments are:

attr Name of the mapped attribute to use for storage and retrieval of ordering information

count_from (optional) Set up an integer-based ordering, starting at `count_from`. For example, `ordering_list('pos', count_from=1)` would create a 1-based list in SQL, storing the value in the 'pos' column. Ignored if `ordering_func` is supplied.

Passes along any keyword arguments to `OrderingList` constructor.

8.4.4 serializer

author Mike Bayer

Serializer/Deserializer objects for usage with SQLAlchemy structures.

Any SQLAlchemy structure, including Tables, Columns, expressions, mappers, Query objects etc. can be serialized in a minimally-sized format, and deserialized when given a Metadata and optional ScopedSession object to use as context on the way out.

Usage is nearly the same as that of the standard Python pickle module:

```
from sqlalchemy.ext.serializer import loads, dumps
metadata = MetaData(bind=some_engine)
Session = scoped_session(sessionmaker())

# ... define mappers

query = Session.query(MyClass).filter(MyClass.somedata=='foo').order_by(MyClass.sortkey)

# pickle the query
serialized = dumps(query)

# unpickle.  Pass in metadata + scoped_session
query2 = loads(serialized, metadata, Session)

print query2.all()
```

Similar restrictions as when using raw pickle apply; mapped classes must be themselves be pickleable, meaning they are importable from a module-level namespace.

Note that instances of user-defined classes do not require this extension in order to be pickled; these contain no references to engines, sessions or expression constructs in the typical case and can be serialized directly. This module is specifically for ORM and expression constructs.

Serializer (**args, **kw*)

Deserializer (*file, metadata=None, scoped_session=None, engine=None*)

dumps (*obj*)

loads (*data, metadata=None, scoped_session=None, engine=None*)

8.4.5 SqlSoup

author Jonathan Ellis

SqlSoup creates mapped classes on the fly from tables, which are automatically reflected from the database based on name. It is essentially a nicer version of the "row data gateway" pattern.

```
>>> from sqlalchemy.ext.sqlsoup import SqlSoup
>>> soup = SqlSoup('sqlite:///')

>>> db.users.select(order_by=[db.users.c.name])
[MappedUsers(name='Bhargan Basepair',email='basepair@example.edu',password='basepair',clas
 MappedUsers(name='Joe Student',email='student@example.edu',password='student',classname=No
```

Full SqlSoup documentation is on the SQLAlchemy Wiki.

Introduction

SqlSoup provides a convenient way to access database tables without having to declare table or mapper classes ahead of time.

Suppose we have a database with users, books, and loans tables (corresponding to the PyWebOff dataset, if you're curious). For testing purposes, we'll create this db as follows:

```
>>> from sqlalchemy import create_engine
>>> e = create_engine('sqlite:///:memory:')
>>> for sql in _testsql: e.execute(sql)
<...
```

Creating a SqlSoup gateway is just like creating an SQLAlchemy engine:

```
>>> from sqlalchemy.ext.sqlsoup import SqlSoup
>>> db = SqlSoup('sqlite:///:memory:')
```

or, you can re-use an existing metadata or engine:

```
>>> db = SqlSoup(MetaData(e))
```

You can optionally specify a schema within the database for your SqlSoup:

```
# >>> db.schema = myschemaname
```

Loading objects

Loading objects is as easy as this:

```
>>> users = db.users.all()
>>> users.sort()
>>> users
[MappedUsers(name=u'Joe Student',email=u'student@example.edu',password=u'student',classname
```

Of course, letting the database do the sort is better:

```
>>> db.users.order_by(db.users.name).all()
[MappedUsers(name=u'Bhargan Basepair',email=u'basepair@example.edu',password=u'basepair',cl
```

Field access is intuitive:

```
>>> users[0].email
u'student@example.edu'
```

Of course, you don't want to load all users very often. Let's add a WHERE clause. Let's also switch the order_by to DESC while we're at it:

```
>>> from sqlalchemy import or_, and_, desc
>>> where = or_(db.users.name=='Bhargan Basepair', db.users.email=='student@example.edu')
>>> db.users.filter(where).order_by(desc(db.users.name)).all()
[MappedUsers(name=u'Joe Student',email=u'student@example.edu',password=u'student',classname
```

You can also use .first() (to retrieve only the first object from a query) or .one() (like .first when you expect exactly one user – it will raise an exception if more were returned):

```
>>> db.users.filter(db.users.name=='Bhargan Basepair').one()
MappedUsers(name=u'Bhargan Basepair',email=u'basepair@example.edu',password=u'basepair',cla
```

Since name is the primary key, this is equivalent to

```
>>> db.users.get('Bhargan Basepair')
MappedUsers(name=u'Bhargan Basepair',email=u'basepair@example.edu',password=u'basepair',cla
```

This is also equivalent to

```
>>> db.users.filter_by(name='Bhargan Basepair').one()
MappedUsers(name=u'Bhargan Basepair',email=u'basepair@example.edu',password=u'basepair',cla
```

filter_by is like filter, but takes kwargs instead of full clause expressions. This makes it more concise for simple queries like this, but you can't do complex queries like the or_ above or non-equality based comparisons this way.

Full query documentation

Get, filter, filter_by, order_by, limit, and the rest of the query methods are explained in detail in the SQLAlchemy documentation.

Modifying objects

Modifying objects is intuitive:

```
>>> user = _
>>> user.email = 'basepair+nospam@example.edu'
>>> db.flush()
```

(SqlSoup leverages the sophisticated SQLAlchemy unit-of-work code, so multiple updates to a single object will be turned into a single `UPDATE` statement when you flush.)

To finish covering the basics, let's insert a new loan, then delete it:

```
>>> book_id = db.books.filter_by(title='Regional Variation in Moss').first().id
>>> db.loans.insert(book_id=book_id, user_name=user.name)
MappedLoans(book_id=2,user_name=u'Bhargan Basepair',loan_date=None)
>>> db.flush()

>>> loan = db.loans.filter_by(book_id=2, user_name='Bhargan Basepair').one()
>>> db.delete(loan)
>>> db.flush()
```

You can also delete rows that have not been loaded as objects. Let's do our insert/delete cycle once more, this time using the loans table's delete method. (For SQLAlchemy experts: note that no flush() call is required since this delete acts at the SQL level, not at the Mapper level.) The same where-clause construction rules apply here as to the select methods.

```
>>> db.loans.insert(book_id=book_id, user_name=user.name)
MappedLoans(book_id=2,user_name=u'Bhargan Basepair',loan_date=None)
>>> db.flush()
>>> db.loans.delete(db.loans.book_id==2)
```

You can similarly update multiple rows at once. This will change the book_id to 1 in all loans whose book_id is 2:

```
>>> db.loans.update(db.loans.book_id==2, book_id=1)
>>> db.loans.filter_by(book_id=1).all()
[MappedLoans(book_id=1,user_name=u'Joe Student',loan_date=datetime.datetime(2006, 7, 12, 0,
```

Joins

Occasionally, you will want to pull out a lot of data from related tables all at once. In this situation, it is far more efficient to have the database perform the necessary join. (Here we do not have *a lot of data* but hopefully the concept is still clear.) SQLAlchemy is smart enough to recognize that loans has a foreign key to users, and uses that as the join condition automatically.

```
>>> join1 = db.join(db.users, db.loans, isouter=True)
>>> join1.filter_by(name='Joe Student').all()
[MappedJoin(name=u'Joe Student',email=u'student@example.edu',password=u'student',classname=
```

If you're unfortunate enough to be using MySQL with the default MyISAM storage engine, you'll have to specify the join condition manually, since MyISAM does not store foreign keys. Here's the same join again, with the join condition explicitly specified:

```
>>> db.join(db.users, db.loans, db.users.name==db.loans.user_name, isouter=True)
<class 'sqlalchemy.ext.sqlsoup.MappedJoin'>
```

You can compose arbitrarily complex joins by combining Join objects with tables or other joins. Here we combine our first join with the books table:

```
>>> join2 = db.join(join1, db.books)
>>> join2.all()
[MappedJoin(name=u'Joe Student',email=u'student@example.edu',password=u'student',classname=
```

If you join tables that have an identical column name, wrap your join with *with_labels*, to disambiguate columns with their table name (.c is short for .columns):

```
>>> db.with_labels(join1).c.keys()
[u'users_name', u'users_email', u'users_password', u'users_classname', u'users_admin', u'lo
```

You can also join directly to a labeled object:

```
>>> labeled_loans = db.with_labels(db.loans)
>>> db.join(db.users, labeled_loans, isouter=True).c.keys()
[u'name', u'email', u'password', u'classname', u'admin', u'loans_book_id', u'loans_user_nam
```

Relations

You can define relations on SqlSoup classes:

```
>>> db.users.relate('loans', db.loans)
```

These can then be used like a normal SA property:

```
    >>> db.users.get('Joe Student').loans
    [MappedLoans(book_id=1,user_name=u'Joe Student',loan_date=datetime.datetime(2006, 7, 1

    >>> db.users.filter(~db.users.loans.any()).all()
    [MappedUsers(name=u'Bhargan Basepair',email='basepair+nospam@example.edu',password=u'ba
```

relate can take any options that the relation function accepts in normal mapper definition:

```
>>> del db._cache['users']
>>> db.users.relate('loans', db.loans, order_by=db.loans.loan_date, cascade='all, delete-o
```

Advanced Use

Accessing the Session

SqlSoup uses a ScopedSession to provide thread-local sessions. You can get a reference to the current one like this:

```
>>> from sqlalchemy.ext.sqlsoup import Session
>>> session = Session()
```

Now you have access to all the standard session-based SA features, such as transactions. (SqlSoup's `flush()` is normally transactionalized, but you can perform manual transaction management if you need a transaction to span multiple flushes.)

Mapping arbitrary Selectables

SqlSoup can map any SQLAlchemy `Selectable` with the map method. Let's map a `Select` object that uses an aggregate function; we'll use the SQLAlchemy `Table` that SqlSoup introspected as the basis. (Since we're not mapping to a simple table or join, we need to tell SQLAlchemy how to find the *primary key* which just needs to be unique within the select, and not necessarily correspond to a *real* PK in the database.)

```
>>> from sqlalchemy import select, func
>>> b = db.books._table
>>> s = select([b.c.published_year, func.count('*').label('n')], from_obj=[b], group_by=[b
>>> s = s.alias('years_with_count')
>>> years_with_count = db.map(s, primary_key=[s.c.published_year])
>>> years_with_count.filter_by(published_year='1989').all()
[MappedBooks(published_year=u'1989',n=1)]
```

Obviously if we just wanted to get a list of counts associated with book years once, raw SQL is going to be less work. The advantage of mapping a Select is reusability, both standalone and in Joins. (And if you go to full SQLAlchemy, you can perform mappings like this directly to your object models.)

An easy way to save mapped selectables like this is to just hang them on your db object:

```
>>> db.years_with_count = years_with_count
```

Python is flexible like that!

Raw SQL

SqlSoup works fine with SQLAlchemy's text block support.

You can also access the SqlSoup's *engine* attribute to compose SQL directly. The engine's `execute` method corresponds to the one of a DBAPI cursor, and returns a `ResultProxy` that has `fetch` methods you would also see on a cursor:

```
>>> rp = db.bind.execute('select name, email from users order by name')
>>> for name, email in rp.fetchall(): print name, email
Bhargan Basepair basepair+nospam@example.edu
Joe Student student@example.edu
```

You can also pass this engine object to other SQLAlchemy constructs.

Dynamic table names

You can load a table whose name is specified at runtime with the entity() method:

```
>>> tablename = 'loans'
>>> db.entity(tablename) == db.loans
True
```

entity() also takes an optional schema argument. If none is specified, the default schema is used.

Extra tests

Boring tests here. Nothing of real expository value.

```
>>> db.users.filter_by(classname=None).order_by(db.users.name).all()
[MappedUsers(name=u'Bhargan Basepair',email=u'basepair+nospam@example.edu',password=u'basep

>>> db.nopk
...
PKNotFoundError: table 'nopk' does not have a primary key defined [columns: i]

>>> db.nosuchtable
...
NoSuchTableError: nosuchtable

>>> years_with_count.insert(published_year='2007', n=1)
...
InvalidRequestError: SQLSoup can only modify mapped Tables (found: Alias)

[tests clear()]
>>> db.loans.count()
1
>>> _ = db.loans.insert(book_id=1, user_name='Bhargan Basepair')
>>> db.expunge_all()
>>> db.flush()
>>> db.loans.count()
1
```

exception PKNotFoundError

class SqlSoup (**args*, ***kwargs*)

> **__init__** (**args*, ***kwargs*)
>
> Initialize a new `SqlSoup`.
>
> *args* may either be an `SQLEngine` or a set of arguments suitable for passing to `create_engine`.
>
> **bind**
>
> **clear** ()
>
> **delete** (**args*, ***kwargs*)
>
> **engine**
>
> **entity** (*attr*, *schema=None*)
>
> **expunge_all** ()
>
> **flush** ()
>
> **join** (**args*, ***kwargs*)
>
> **map** (*selectable*, ***kwargs*)
>
> **with_labels** (*item*)

8.4.6 compiler

Provides an API for creation of custom ClauseElements and compilers.

Synopsis

Usage involves the creation of one or more `ClauseElement` subclasses and one or more callables defining its compilation:

```
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.sql.expression import ColumnClause

class MyColumn(ColumnClause):
    pass

@compiles(MyColumn)
def compile_mycolumn(element, compiler, **kw):
    return "[%s]" % element.name
```

Above, `MyColumn` extends `ColumnClause`, the base expression element for column objects. The `compiles` decorator registers itself with the `MyColumn` class so that it is invoked when the object is compiled to a string:

```
from sqlalchemy import select

s = select([MyColumn('x'), MyColumn('y')])
print str(s)
```

Produces:

```
SELECT [x], [y]
```

Compilers can also be made dialect-specific. The appropriate compiler will be invoked for the dialect in use:

```
from sqlalchemy.schema import DDLElement  # this is a SQLA 0.6 construct

class AlterColumn(DDLElement):

    def __init__(self, column, cmd):
        self.column = column
        self.cmd = cmd

@compiles(AlterColumn)
def visit_alter_column(element, compiler, **kw):
    return "ALTER COLUMN %s ..." % element.column.name

@compiles(AlterColumn, 'postgres')
def visit_alter_column(element, compiler, **kw):
    return "ALTER TABLE %s ALTER COLUMN %s ..." % (element.table.name, element.column.name
```

The second `visit_alter_table` will be invoked when any `postgres` dialect is used.

The `compiler` argument is the `Compiled` object in use. This object can be inspected for any information about the in-progress compilation, including `compiler.dialect`, `compiler.statement` etc. The `SQLCompiler` and `DDLCompiler` (DDLCompiler is 0.6. only) both include a `process()` method which can be used for compilation of embedded attributes:

```
class InsertFromSelect(ClauseElement):
    def __init__(self, table, select):
        self.table = table
        self.select = select

@compiles(InsertFromSelect)
def visit_insert_from_select(element, compiler, **kw):
    return "INSERT INTO %s (%s)" % (
        compiler.process(element.table, asfrom=True),
        compiler.process(element.select)
    )

insert = InsertFromSelect(t1, select([t1]).where(t1.c.x>5))
print insert
```

Produces:

```
"INSERT INTO mytable (SELECT mytable.x, mytable.y, mytable.z FROM mytable WHERE mytable.x
```

CHAPTER

NINE

INDICES AND TABLES

MODULE INDEX

INDEX