SQLAlchemy Documentation

Release 0.6.6

Mike Bayer

December 05, 2010

CONTENTS

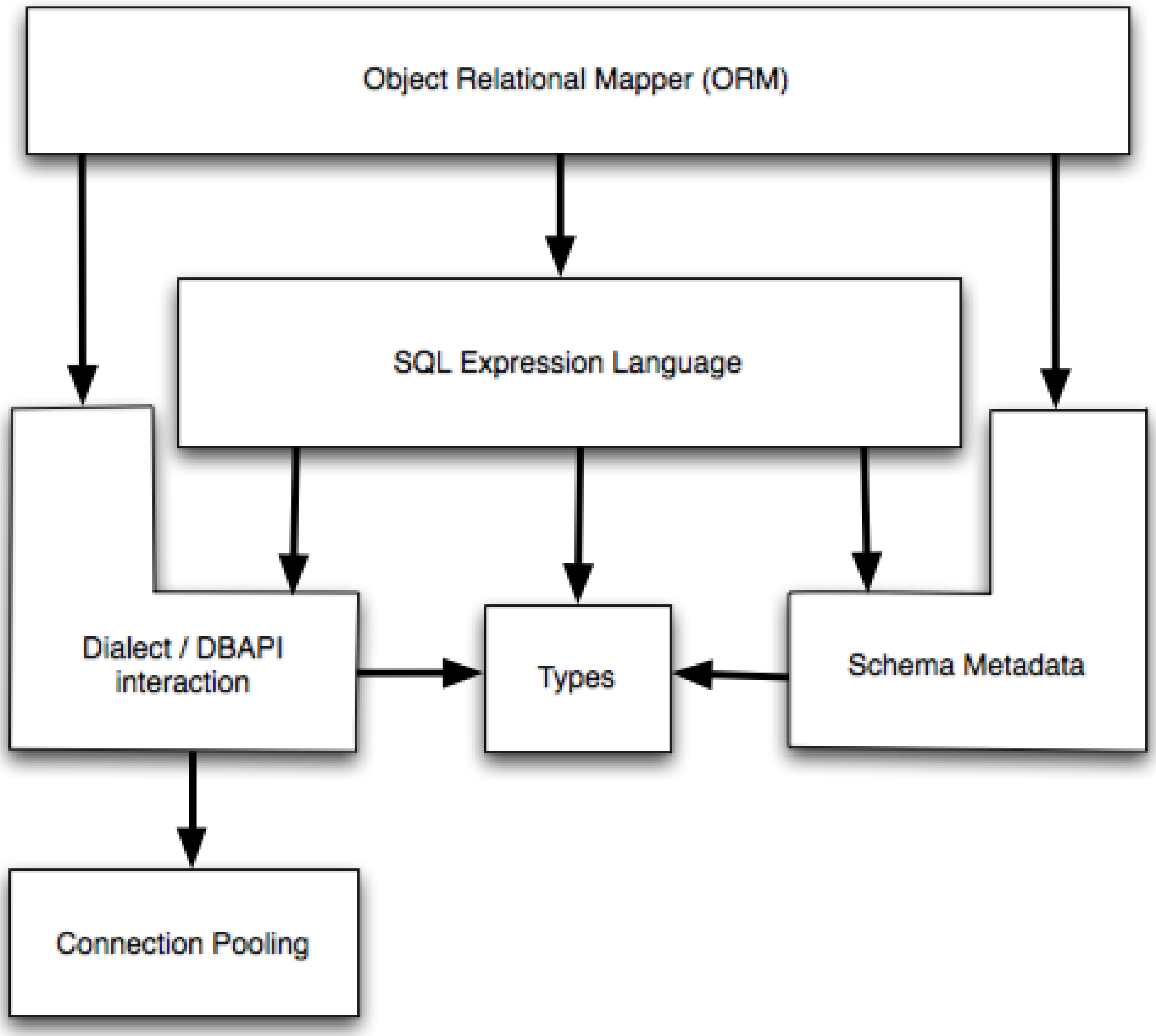
CHAPTER

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 components are illustrated below. The arrows represent the general dependencies of components:



Above, the two most significant front-facing portions of SQLAlchemy are the **Object Relational Mapper** and the **SQL Expression Language**. SQL Expressions can be used independently of the ORM. When using the ORM, the SQL Expression language remains part of the public facing API as it is used within object-relational configurations and queries.

1.2 Documentation Overview

The documentation is separated into three sections: *SQLAlchemy ORM*, *SQLAlchemy Core*, and *Dialects*.

In *SQLAlchemy ORM*, the Object Relational Mapper is introduced and fully described. New users should begin with the *Object Relational Tutorial*. 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.

In *SQLAlchemy Core*, the breadth of SQLAlchemy's SQL and database integration and description services are documented, the core of which is the 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. In contrast to the ORM's domain-centric mode of usage, the expression language provides a schema-centric usage paradigm. New users should begin here with *SQL Expression Language Tutorial*. SQLAlchemy engine, connection, and pooling services are also described in *SQLAlchemy Core*.

In *Dialects*, reference documentation for all provided database and DBAPI backends is provided.

1.3 Code Examples

Working code examples, mostly regarding the ORM, are included in the SQLAlchemy distribution. A description of all the included example applications is at *Examples*.

There is also a wide variety of examples involving both core SQLAlchemy constructs as well as the ORM on the wiki. See http://www.sqlalchemy.org/trac/wiki/UsageRecipes.

1.4 Installing SQLAlchemy

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

```
# easy_install SQLAlchemy
```

Or with pip:

```
# pip install SQLAlchemy
```

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

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

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

1.5 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*.

1.6 Checking the Installed SQLAlchemy Version

This documentation covers SQLAlchemy version 0.6. 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__
0.6.0
```

1.7 0.5 to 0.6 Migration

Notes on what's changed from 0.5 to 0.6 is available on the SQLAlchemy wiki at 06Migration.

CHAPTER

TWO

SQLALCHEMY ORM

2.1 Object Relational Tutorial

2.1.1 Introduction

The SQLAlchemy Object Relational Mapper presents a method of associating user-defined Python classes with database tables, and instances of those classes (objects) with rows in their corresponding tables. It includes a system that transparently synchronizes all changes in state between objects and their related rows, called a unit of work, as well as a system for expressing database queries in terms of the user defined classes and their defined relationships between each other.

The ORM is in contrast to the SQLAlchemy Expression Language, upon which the ORM is constructed. Whereas the SQL Expression Language, introduced in *SQL Expression Language Tutorial*, presents a system of representing the primitive constructs of the relational database directly without opinion, the ORM presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language.

While there is overlap among the usage patterns of the ORM and the Expression Language, the similarities are more superficial than they may at first appear. One approaches the structure and content of data from the perspective of a user-defined domain model which is transparently persisted and refreshed from its underlying storage model. The other approaches it from the perspective of literal schema and SQL expression representations which are explicitly composed into messages consumed individually by the database.

A successful application may be constructed using the Object Relational Mapper exclusively. In advanced situations, an application constructed with the ORM may make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

The following 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.

2.1.2 Version Check

A quick check to verify that we are on at least **version 0.6** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
0.6.0
```

2.1.3 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.

2.1.4 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)
... )
```

Schema Definition Language covers all about how to define `Table` objects, as well as how to load their definition from an existing database (known as **reflection**).

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
```

Note: Users familiar with the syntax of CREATE TABLE may notice that the VARCHAR columns were generated without a length; on SQLite and Postgresql, this is a valid datatype, but on others, it's not allowed. So if running this tutorial on one of those databases, and you wish to use SQLAlchemy to issue CREATE TABLE, 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.

Additionally, Firebird and Oracle require sequences to generate new primary key identifiers, and SQLAlchemy doesn't generate or assume these without being instructed. For that, you use the `Sequence` construct:

```
from sqlalchemy import Sequence
Column('id', Integer, Sequence('user_id_seq'), primary_key=True)
```

A full, foolproof `Table` is therefore:

```
users_table = Table('users', metadata,
    Column('id', Integer, Sequence('user_id_seq'), primary_key=True),
    Column('name', String(50)),
    Column('fullname', String(50)),
    Column('password', String(12))
)
```

2.1.5 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.

2.1.6 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.

2.1.7 Creating Table, Class and Mapper All at Once Declaratively

The preceding approach to configuration involved a Table, a user-defined class, and a call to‘‘mapper()‘‘. This illustrates 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__
```

The owning MetaData object is available as well:

```
>>> metadata = Base.metadata
```

Full documentation for declarative can be found in the reference/index section for reference/ext/declarative.

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.

2.1.8 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.

2.1.9 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 (implicit)
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, us
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 (implicit)
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.

2.1.10 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 (implicit)
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')>]
```

2.1.11 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, us
FROM users
WHERE users.name = ? AND users.fullname = ?
('ed', 'Ed Jones')<User('ed','Ed Jones', 'f8s7ccs')>
```

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']))

  # works with query objects too:

  query.filter(User.name.in_(session.query(User.name).filter(User.name.like('%ed%'))))
  ```

- NOT IN:

  ```
  query.filter(~User.name.in_(['ed', 'wendy', 'jack']))
  ```

- IS NULL:

  ```
  filter(User.name == None)
  ```

- IS NOT 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.

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()`, fully fetches all rows, and if not exactly one object identity or composite row is present in the result, 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
('%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
('%ed', 99)No row was found for one()
```

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, us
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, us
FROM users
WHERE id<? and name=? ORDER BY users.id
(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')>]
```

You can use `from_statement()` to go completely "raw", using string names to identify desired columns:

```
>>> session.query("id", "name", "thenumber12").from_statement("SELECT id, name, 12 as thenu
SELECT id, name, 12 as thenumber12 FROM users where name=?
('ed',)[(1, u'ed', 12)]
```

Counting

`Query` includes a convenience method for counting called `count()`:

```
>>> session.query(User).filter(User.name.like('%ed')).count()
SELECT count(1) AS count_1
FROM users
WHERE users.name LIKE ?
('%ed',)2
```

The `count()` method is used to determine how many rows the SQL statement would return, and is mainly intended to return a simple count of a single type of entity, in this case `User`. For more complicated sets of columns or entities where the "thing to be counted" needs to be indicated more specifically, `count()` is probably not what you want. Below, a query for individual columns does return the expected result:

```
>>> session.query(User.id, User.name).filter(User.name.like('%ed')).count()
SELECT count(1) AS count_1
FROM (SELECT users.id AS users_id, users.name AS users_name
FROM users
WHERE users.name LIKE ?) AS anon_1
('%ed',)2
```

...but if you look at the generated SQL, SQLAlchemy saw that we were placing individual column expressions and decided to wrap whatever it was we were doing in a subquery, so as to be assured that it returns the "number of rows". This defensive behavior is not really needed here and in other cases is not what we want at all, such as if we wanted a grouping of counts per name:

```
>>> session.query(User.name).group_by(User.name).count()
SELECT count(1) AS count_1
FROM (SELECT users.name AS users_name
FROM users GROUP BY users.name) AS anon_1
()4
```

We don't want the number 4, we wanted some rows back. So for detailed queries where you need to count something specific, use the `func.count()` function as a column expression:

```
>>> from sqlalchemy import func
>>> session.query(func.count(User.name), User.name).group_by(User.name).all()
SELECT count(users.name) AS count_1, users.name AS users_name
FROM users GROUP BY users.name()
[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]
```

2.1.12 Building a Relationship

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 relationship, 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 = relationship(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 `relationship()` 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 relationship 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.

Note: The `relationship()` function has historically been known as `relation()`, which is the name that's available in all versions of SQLAlchemy prior to 0.6beta2, including the 0.5 and 0.4 series. `relationship()` is only available starting with SQLAlchemy 0.6beta2. `relation()` will remain available in SQLAlchemy for the foreseeable future to enable cross-compatibility.

The `relationship()` function is extremely flexible, and could just have easily been defined on the `User` class:

```
class User(Base):
    # ....
    addresses = relationship(Address, order_by=Address.id, backref="user")
```

We are also free to not define a backref, and to define the `relationship()` only on one class and not the other. It is also possible to define two separate `relationship()` constructs for either direction, which is generally safe for many-to-one and one-to-many relationships, but not for many-to-many relationships.

When using the `declarative` extension, `relationship()` 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 = relationship("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
```

2.1.13 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 *alternate_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 (implicit)
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname, us
FROM users
WHERE users.name = ?
('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 relationship**. 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, using the `joinedload()` function. This function is a **query option** that gives additional instructions to the query on how we would like it to load, in this case we'd like to indicate 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 joinedload

>>> jack = session.query(User).\
...                            options(joinedload('addresses')).\
...                            filter_by(name='jack').one()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname,
users.password AS users_password, addresses_1.id AS addresses_1_id, addresses_1.email_addre
AS addresses_1_email_address, addresses_1.user_id AS addresses_1_user_id
FROM users LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = ? ORDER BY addresses_1.id
('jack',)>>> jack
<User('jack','Jack Bean', 'gjffdd')>

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

See *mapper_loader_strategies* for information on `joinedload()` and its new brother, `subqueryload()`. We'll also see another way to "eagerly" load in the next section.

2.1.14 Querying with Joins

While `joinedload()` 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; the most common way is to use `join()`:

```
>>> session.query(User).join(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()` works better when one of the following forms are used:

```
query.join((Address, User.id==Address.user_id))  # explicit condition (note the tuple)
query.join(User.addresses)                       # specify relationship from left to right
query.join((Address, User.addresses))            # same, with explicit target
query.join('addresses')                          # same, using a string
```

Note that when `join()` is called with an explicit target as well as an ON clause, we use a tuple as the argument. This is so that multiple joins can be chained together, as in:

```
session.query(Foo).join(
                        Foo.bars,
                        (Bat, bar.bats),
                        (Widget, Bat.widget_id==Widget.id)
                        )
```

The above would produce SQL something like `foo JOIN bars ON <onclause> JOIN bats ON <onclause> JOIN widgets ON <onclause>`.

The general functionality of `join()` is also available as a standalone function `join()`, which is an ORM-enabled version of the same function present in the SQL expression language. This function accepts two or three arguments (left side, right side, optional ON clause) and can be used in conjunction with the `select_from()` method to set an explicit FROM clause:

```
>>> from sqlalchemy.orm import join
>>> session.query(User).\
...                 select_from(join(User, Address, 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')>]
```

Using join() to Eagerly Load Collections/Attributes

The "eager loading" capabilities of the `joinedload()` function and the join-construction capabilities of `join()` or an equivalent can be combined together using the `contains_eager()` option. This is typically used for a query that is already joining to some related entity (more often than not via many-to-one), and you'd like the related entity to also be loaded onto the resulting objects in one step without the need for additional queries and without the "automatic" join embedded by the `joinedload()` function:

```
>>> from sqlalchemy.orm import contains_eager
>>> for address in session.query(Address).\
...                 join(Address.user).\
...                 filter(User.name=='jack').\
...                 options(contains_eager(Address.user)):
...         print address, address.user
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 addresses JOIN users ON users.id = addresses.user_id
WHERE users.name = ?
('jack',)<Address('jack@google.com')> <User('jack','Jack Bean', 'gjffdd')>
<Address('j25@yahoo.com')> <User('jack','Jack Bean', 'gjffdd')>
```

Note that above the join was used both to limit the rows to just those `Address` objects which had a related `User` object with the name "jack". It's safe to have the `Address.user` attribute populated with this user using an inner join. However, when filtering on a join that is filtering on a particular member of a collection, using `contains_eager()` to populate a related collection may populate the collection with only part of what it actually references, since the collection itself is filtered.

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
```

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 Expression Language Tutorial*:

```
>>> 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
```

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').subquer
>>> 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, addr
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')>
```

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` relationship 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 relationships (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',)[]
```

Common Relationship Operators

Here's all the operators which build on relationships:

- 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 relationship):

```
session.query(Address).with_parent(someuser, 'addresses')
```

2.1.15 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.

Configuring delete/delete-orphan Cascade

We will configure **cascade** options on the `User.addresses` relationship to change the behavior. While SQLAlchemy allows you to add new attributes and relationships to mappings at any point in time, in this case the existing relationship needs to be removed, so we need to tear down the mappings completely and start again.

Note: Tearing down mappers with `clear_mappers()` is not a typical operation, and normal applications do not need to use this function. It is here so that the tutorial code can be executed as a whole.

```
>>> session.close()  # roll back and close the transaction
>>> from sqlalchemy.orm import clear_mappers
>>> clear_mappers() # remove all class mappings
```

Below, we use `mapper()` to reconfigure an ORM mapping for `User` and `Address`, on our existing but currently un-mapped classes. The `User.addresses` relationship 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:

```
>>> users_table = User.__table__
>>> mapper(User, users_table, properties={
...     'addresses':relationship(Address, backref='user', cascade="all, delete, delete-orph
... })
<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 (implicit)
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
```

2.1.16 Building a Many To Many Relationship

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 = relationship('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 relationship is `BlogPost.keywords`. The defining feature of a many-to-many relationship 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 relationship; 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 relationship 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` relationship were added to the `Keyword` entity, both relationships 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 `relationship()` called `lazy='dynamic'`, which configures an alternate **loader strategy** on the attribute. To use it on the "reverse" side of a `relationship()`, we use the `backref()` function:

```
>>> from sqlalchemy.orm import backref
>>> # "dynamic" loading relationship to User
>>> BlogPost.author = relationship(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 = ?
('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, 1), (1, 2))
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 Willian
```

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` relationship, which is a "dynamic" relationship, 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_headl
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 keyw
(2, 'firstpost')[BlogPost("Wendy's Blog Post", 'This is a test', <User('wendy','Wendy Will
```

2.1.17 Further Reference

Query Reference: *Querying*

Mapper Reference: *Mapper Configuration*

Relationship Reference: *Relationship Configuration*

Session Reference: *Using the Session*.

2.2 Mapper Configuration

This section describes a variety of configurational patterns that are usable with mappers. It assumes you've worked through *Object Relational Tutorial* and know how to construct and use rudimentary mappers and relationships.

Note that all patterns here apply both to the usage of explicit `mapper()` and `Table` objects as well as when using the `sqlalchemy.ext.declarative` extension. Any example in this section which takes a form such as:

```
mapper(User, users_table, primary_key=[users_table.c.id])
```

Would translate into declarative as:

```
class User(Base):
    __table__ = users_table
    __mapper_args__ = {
        'primary_key':[users_table.c.id]
    }
```

Or if using `__tablename__`, `Column` objects are declared inline with the class definition. These are usable as is within `__mapper_args__`:

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

    id = Column(Integer)

    __mapper_args__ = {
        'primary_key':[id]
    }
```

2.2.1 Customizing Column Properties

The default behavior of `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.

Mapping a Subset of Table Columns

To reference a subset of columns referenced by a table as mapped attributes, use the `include_properties` or `exclude_properties` arguments. For example:

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

...will map the `User` class to the `users_table` table, only including the "user_id" and "user_name" columns - the rest are not refererenced. Similarly:

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

...will map the `Address` class to the `addresses_table` table, including all columns present except "street", "city", "state", and "zip".

When this mapping is used, the columns that are not included will not be referenced in any SELECT statements emitted by `Query`, nor will there be any mapped attribute on the mapped class which represents the column; assigning an attribute of that name will have no effect beyond that of a normal Python attribute assignment.

In some cases, multiple columns may have the same name, such as when mapping to a join of two or more tables that share some column name. To exclude or include individual columns, `Column` objects may also be placed within the "include_properties" and "exclude_properties" collections (new feature as of 0.6.4):

```
mapper(UserAddress, users_table.join(addresses_table),
            exclude_properties=[addresses_table.c.id],
            primary_key=[users_table.c.id]
        )
```

It should be noted that insert and update defaults configured on individual `Column` objects, such as those configured by the "default", "on_update", "server_default" and "server_onupdate" arguments, will continue to function normally even if those `Column` objects are not mapped. This functionality is part of the SQL expression and execution system and occurs below the level of the ORM.

Attribute Names for Mapped Columns

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,
})
```

When using `declarative`, the above configuration is more succinct - place the full column name in the `Column` definition, using the desired attribute name in the class definition:

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

class User(Base):
    __tablename__ = 'user'
    id = Column('user_id', Integer, primary_key=True)
    name = Column('user_name', String(50))
```

To change the names of all attributes using a prefix, use the `column_prefix` option. This is useful for some schemes that would like to declare alternate attributes:

```
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.

Mapping Multiple Columns to a Single Attribute

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()`:

```
from sqlalchemy.sql import join

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

# user_id columns are equated under the 'user_id' attribute
mapper(User, usersaddresses, properties={
    'id':[users_table.c.user_id, addresses_table.c.user_id],
})
```

For further examples on this particular use case, see *Mapping a Class against Multiple Tables*.

column_property API

The establishment of a `Column` on a `mapper()` can be further customized using the `column_property()` function, as specified to the `properties` dictionary. This function is usually invoked implicitly for each mapped `Column`. Explicit usage looks like:

```
from sqlalchemy.orm import mapper, column_property

mapper(User, users, properties={
    'name':column_property(users.c.name, active_history=True)
})
```

or with declarative:

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

    id = Column(Integer, primary_key=True)
    name = column_property(Column(String(50)), active_history=True)
```

Further examples of `column_property()` are at *SQL Expressions as Mapped Attributes*.

`sqlalchemy.orm.`**`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 `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.

Parameters

- ***cols** – list of Column objects to be mapped.

- **active_history=False** – When `True`, indicates that the "previous" value for a scalar attribute should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple non-primary-key scalar values only needs to be aware of the "new" value in order to perform a flush. This flag is available for applications that make use of `attributes.get_history()` which also need to know the "previous" value of the attribute. (new in 0.6.6)
- **comparator_factory** – a class which extends `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()`.
- **doc** – optional string that will be applied as the doc on the class-bound descriptor.
- **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.

2.2.2 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', metadata,
    Column('book_id', Integer, primary_key=True),
    Column('title', String(200), nullable=False),
    Column('summary', String(2000)),
    Column('excerpt', Text),
    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)
})
```

With declarative, `Column` objects can be declared directly inside of `deferred()`:

```
class Book(Base):
    __tablename__ = 'books'

    book_id = Column(Integer, primary_key=True)
    title = Column(String(200), nullable=False)
    summary = Column(String(2000))
```

```
    excerpt = deferred(Column(Text))
    photo = deferred(Column(Binary))
```

Deferred columns can be associted with a "group" name, so that they load together when any of them are first accessed:

```
book_excerpts = Table('books', metadata,
  Column('book_id', Integer, primary_key=True),
  Column('title', String(200), nullable=False),
  Column('summary', String(2000)),
  Column('excerpt', Text),
  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()` query 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()
```

`sqlalchemy.orm.`**`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()`.

`sqlalchemy.orm.`**`defer`**`(`**keys*`)`

Return a `MapperOption` that will convert the column property of the given name into a deferred load.

Used with `options()`.

`sqlalchemy.orm.`**`undefer`**`(`**keys*`)`

Return a `MapperOption` that will convert the column property of the given name into a non-deferred (regular column) load.

Used with `options()`.

`sqlalchemy.orm.`**`undefer_group`**`(`*name*`)`

Return a `MapperOption` that will convert the given group of deferred column properties into a non-deferred (regular column) load.

Used with `options()`.

2.2.3 SQL Expressions as Mapped Attributes

Any SQL expression that relates to the primary mapped selectable can be mapped as a read-only attribute which will be bundled into the SELECT emitted for the target mapper when rows are loaded. This effect is achieved using the `column_property()` function. Any scalar-returning `ClauseElement` may be used. Unlike older versions of SQLAlchemy, there is no `label()` requirement:

```
from sqlalchemy.orm import column_property

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

Correlated subqueries may be used as well:

```
from sqlalchemy.orm import column_property
from sqlalchemy import select, func

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

The declarative form of the above is described in *Defining SQL Expressions*.

Note that `column_property()` is used to provide the effect of a SQL expression that is actively rendered into the SELECT generated for a particular mapped class. Alternatively, for the typical attribute that represents a composed value, its usually simpler to define it as a Python property which is evaluated as it is invoked on instances after they've been loaded:

```
class User(object):
    @property
    def fullname(self):
        return self.firstname + " " + self.lastname
```

To invoke a SQL statement from an instance that's already been loaded, the session associated with the instance can be acquired using `object_session()` which will provide the appropriate transactional context from which to emit a statement:

```
from sqlalchemy.orm import object_session
from sqlalchemy import select, func

class User(object):
    @property
    def address_count(self):
        return object_session(self).\
            scalar(
                select([func.count(addresses_table.c.address_id)]).\
                    where(addresses_table.c.user_id==self.user_id)
            )
```

On the subject of object-level methods, be sure to see the `derived_attributes` example, which provides a simple method of reusing instance-level expressions simultaneously as SQL expressions. The `derived_attributes` example is slated to become a built-in feature of SQLAlchemy in a future release.

2.2.4 Changing Attribute Behavior

Simple Validators

A quick way to add a "validation" routine to an attribute is to use the `validates()` decorator. 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.

```
from sqlalchemy.orm import validates

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
```

`sqlalchemy.orm.`**`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.

Note that a validator for a collection **cannot** issue a load of that collection within the validation routine - this usage raises an assertion to avoid recursion overflows. This is a reentrant condition which is not supported.

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. Below we illustrate this using Python 2.6-style properties:

```
class EmailAddress(object):
```

```
    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, email):
        self._email = email

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

The approach above will work, but there's more we can add. 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.

To use synonyms with `declarative`, see the section *Defining Synonyms*.

Note that the "synonym" feature is eventually to be replaced by the superior "hybrid attributes" approach, slated to become a built in feature of SQLAlchemy in a future release. "hybrid" attributes are simply Python properties that evaulate at both the class level and at the instance level. For an example of their usage, see the `derived_attributes` example.

`sqlalchemy.orm.`**`synonym`**`(`*name*, *map_column=False*, *descriptor=None*, *comparator_factory=None*, *doc=None*`)`

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 relationships.

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.

Custom Comparators

The expressions returned by comparison operations, such as `User.name=='ed'`, can be customized, by implementing an object that explicitly defines each comparison method needed. This is a relatively rare use case. For most needs, the approach in *SQL Expressions as Mapped Attributes* will often suffice, or alternatively a scheme like that of the `derived_attributes` example. Those approaches should be tried first before resorting to custom comparison objects.

Each of `column_property()`, `composite()`, `relationship()`, and `comparable_property()` accept an argument called `comparator_factory`. A subclass of `PropComparator` can be provided for this argument, which can then reimplement basic Python comparison methods such as `__eq__()`, `__ne__()`, `__lt__()`, and so on.

It's best to subclass the `PropComparator` subclass provided by each type of property. For example, to allow a column-mapped attribute to do case-insensitive comparison:

```
from sqlalchemy.orm.properties import ColumnProperty
from sqlalchemy.sql import func

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)
```

When building a `PropComparator`, the `__clause_element__()` method should be used in order to acquire the underlying mapped column. This will return a column that is appropriately wrapped in any kind of subquery or aliasing that has been applied in the context of the generated SQL statement.

class `sqlalchemy.orm.interfaces.`**`PropComparator`**(*prop*, *mapper*, *adapter=None*)

Bases: `sqlalchemy.sql.expression.ColumnOperators`

Defines comparison operations for MapperProperty objects.

User-defined subclasses of `PropComparator` may be created. The built-in Python comparison and math operator methods, such as `__eq__()`, `__lt__()`, `__add__()`, can be overridden to provide new operator behaivor. The custom PropComparator is passed to the mapper property via the comparator_factory argument. In each case, the appropriate subclass of PropComparator should be used:

```
from sqlalchemy.orm.properties import \
                        ColumnProperty,\
                        CompositeProperty,\
                        RelationshipProperty

class MyColumnComparator(ColumnProperty.Comparator):
    pass

class MyCompositeComparator(CompositeProperty.Comparator):
    pass

class MyRelationshipComparator(RelationshipProperty.Comparator):
    pass
```

sqlalchemy.orm.**comparable_property**(*comparator_factory*, *descriptor=None*)

Provides a method of applying a PropComparator to any Python descriptor 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:

```
from sqlalchemy.orm import mapper, comparable_property
from sqlalchemy.orm.interfaces import PropComparator
from sqlalchemy.sql import func

class MyClass(object):
    @property
    def myprop(self):
        return 'foo'

class MyComparator(PropComparator):
    def __eq__(self, other):
        return func.lower(other) == foo

mapper(MyClass, mytable, properties={
         'myprop': comparable_property(MyComparator)})
```

Used with the properties dictionary sent to mapper().

Note that comparable_property() is usually not needed for basic needs. The recipe at derived_attributes offers a simpler pure-Python method of achieving a similar result using class-bound attributes with SQLAlchemy expression constructs.

Parameters

- **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.

2.2.5 Composite Column Types

Sets of columns can be associated with a single user-defined datatype. The ORM provides a single attribute which represents the group of columns using the class you provide.

A simple example represents pairs of columns as a "Point" object. Starting with a table that represents two points as x1/y1 and x2/y2:

```
from sqlalchemy import Table, Column

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

We create a new class, `Point`, that will represent each x/y as a pair:

```
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 is not None and \
                other.x == self.x and \
                other.y == self.y
    def __ne__(self, other):
        return not self.__eq__(other)
```

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.

The `__set_composite_values__()` method is optional. If it's 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.

The `composite()` function is then used in the mapping:

```
from sqlalchemy.orm import composite

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.add(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*. Below we illustrate the "greater than" operator, implementing the same expression that the base "greater than" does:

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

class PointComparator(CompositeProperty.Comparator):
    def __gt__(self, other):
        """redefine 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=PointComparator),
    'end': composite(Point, vertices.c.x2, vertices.c.y2,
                                comparator_factory=PointComparator)
})
```

`sqlalchemy.orm.`**`composite`**`(`*class_*, **cols*, ***kwargs*`)`

Return a composite column-based property for use with a Mapper.

See the mapping documention section *Composite Column Types* for a full usage example.

Parameters

- **class_** – The "composite type" class.
- ***cols** – List of Column objects to be mapped.
- **active_history=False** – When `True`, indicates that the "previous" value for a scalar attribute should be loaded when replaced, if not already loaded. Note that attributes generated by `composite()` properties load the "previous" value in any case, however this is being changed in 0.7, so the flag is introduced here for forwards compatibility. (new in 0.6.6)
- **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 `CompositeProperty.Comparator` which provides custom SQL clause generation for comparison operations.
- **doc** – optional string that will be applied as the doc on the class-bound descriptor.
- **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.

2.2.6 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.

```
from sqlalchemy.orm import mapper
from sqlalchemy.sql import join

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]
})
```

Note that the list of columns is equivalent to the usage of `column_property()` with multiple columns:

```
from sqlalchemy.orm import mapper, column_property

mapper(AddressUser, j, properties={
    'user_id': column_property(users_table.c.user_id, addresses_table.c.user_id)
})
```

The usage of `column_property()` is required when using declarative to map to multiple columns, since the declarative class parser won't recognize a plain list of columns:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class AddressUser(Base):
    __table__ = j

    user_id = column_property(users_table.c.user_id, addresses_table.c.user_id)
```

A second example:

```
from sqlalchemy.sql import join

# 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.

2.2.7 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:

```
from sqlalchemy.sql import select

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.

2.2.8 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 `relationship()` 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 `relationship()` of a parent mapper.

2.2.9 Multiple "Persistence" Mappers for One Class

The non_primary mapper defines alternate mappers for the purposes of loading objects. What if we want the same class to be *persisted* differently, such as to different tables ? SQLAlchemy refers to this as the "entity name" pattern, and in Python one can use a recipe which creates anonymous subclasses which are distinctly mapped. See the recipe at Entity Name.

2.2.10 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` objects may listen for. `MapperExtension` objects can also listen for a `reconstruct_instance` event, analogous to the `reconstructor()` decorator above.

`sqlalchemy.orm.`**`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.

2.2.11 The `mapper()` API

`sqlalchemy.orm.`**`mapper`**(*class_*, *local_table=None*, **args*, ***params*)

Return a new `Mapper` object.

Parameters

- **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 `Query.populate_existing()`.
- **allow_null_pks** – This flag is deprecated - this is stated as allow_partial_pks which defaults to True.
- **allow_partial_pks** – Defaults to True. Indicates that a composite primary key with some NULL values should be considered as possibly existing within the database. This affects whether a mapper will assign an incoming row to an existing identity, as well as if `Session.merge()` will check the database first for a particular primary key value. A "partial primary key" can occur if one has mapped to an OUTER JOIN, for example.
- **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 `Column` objects 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.
- **exclude_properties** – A list or set of string column names to be excluded from mapping. As of SQLAlchemy 0.6.4, this collection may also include `Column` objects. Columns named or present in this list will not be automatically mapped. Note that neither this option nor include_properties will allow one to circumvent plan 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`.
- **extension** – A `MapperExtension` instance or list of `MapperExtension` instances which will be applied to all operations by this `Mapper`.

- **include_properties** – An inclusive list or set of string column names to map. As of SQLAlchemy 0.6.4, this collection may also include `Column` objects in order to disambiguate between same-named columns in a selectable (such as a `join()`). If this list is not `None`, columns present in the mapped table but not named or present in this list will not be automatically mapped. See also "exclude_properties".
- **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.
- **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.
- **order_by** – A single `Column` or list of `Column` objects 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.
- **passive_updates** – Indicates UPDATE behavior of foreign keys when a primary key changes on a joined-table inheritance or other joined table mapping.

  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. PostgreSQL, MySQL with InnoDB tables), ON UPDATE CASCADE is required for this operation. The relationship() 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 relationship() 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).

  Also see the passive_updates flag on `relationship()`.

  A future SQLAlchemy release will provide a "detect" feature for this flag.
- **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_identity** – A value which will be stored in the Column denoted by polymorphic_on, corresponding to the class identity of this mapper.
- **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).

- **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.
- **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 `StaleDataError` exception is thrown.
- **version_id_generator** – A callable which defines the algorithm used to generate new version ids. Defaults to an integer generator. Can be replaced with one that generates timestamps, uuids, etc. e.g.:

```
import uuid

mapper(Cls, table,
version_id_col=table.c.version_uuid,
version_id_generator=lambda version:uuid.uuid4().hex
)
```

  The callable receives the current version identifier as its single argument.

- **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.

`sqlalchemy.orm.`**`object_mapper`**`(`*instance*`)`
Given an object, return the primary Mapper associated with the object instance.

Raises UnmappedInstanceError if no mapping is configured.

`sqlalchemy.orm.`**`class_mapper`**`(`*class_*, *compile=True*`)`
Given a class, return the primary Mapper associated with the key.

Raises UnmappedClassError if no mapping is configured.

`sqlalchemy.orm.`**`compile_mappers`**`()`
Compile all mappers that have been defined.

This is equivalent to calling `compile()` on any individual mapper.

`sqlalchemy.orm.`**`clear_mappers`**`()`
Remove all mappers from all classes.

This function removes all instrumentation from classes and disposes of their associated mappers. Once called, the classes are unmapped and can be later re-mapped with new mappers.

`clear_mappers()` is *not* for normal use, as there is literally no valid usage for it outside of very specific testing scenarios. Normally, mappers are permanent structural components of user-defined classes, and are never discarded independently of their class. If a mapped class itself is garbage collected, its mapper is automatically disposed of as well. As such, `clear_mappers()` is only for usage in test suites that re-use the same classes with different mappings, which is itself an extremely rare use case - the only such use case is in fact SQLAlchemy's own test suite, and possibly the test suites of other ORM extension libraries which intend to test various combinations of mapper construction upon a fixed set of classes.

`sqlalchemy.orm.util.`**`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)

`sqlalchemy.orm.util.`**`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.

class `sqlalchemy.orm.mapper.`**`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*, *version_id_generator=None*, *polymorphic_on=None*, *_polymorphic_map=None*, *polymorphic_identity=None*, *concrete=False*, *with_polymorphic=None*, *allow_null_pks=None*, *allow_partial_pks=True*, *batch=True*, *column_prefix=None*, *include_properties=None*, *exclude_properties=None*, *passive_updates=True*, *eager_defaults=False*, *_compiled_cache_size=100*)

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*, *version_id_generator=None*, *polymorphic_on=None*, *_polymorphic_map=None*, *polymorphic_identity=None*, *concrete=False*, *with_polymorphic=None*, *allow_null_pks=None*, *allow_partial_pks=True*, *batch=True*, *column_prefix=None*, *include_properties=None*, *exclude_properties=None*, *passive_updates=True*, *eager_defaults=False*, *_compiled_cache_size=100*)

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 relationships that meet the given cascade rule.

Parameters

- **type** – The name of the cascade rule (i.e. save-update, delete, etc.)
- **state** – The lead InstanceState. child items will be processed per the relationships 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*, *_compile_mappers=True*)
return a `MapperProperty` associated with the given key.

resolve_synonyms=False and raiseerr=False are deprecated.

get_property_by_column(*column*)
Given a `Column` object, return the `MapperProperty` which maps this column.

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).

self_and_descendants
The collection including this mapper and all descendant mappers.

This includes not just the immediately inheriting mappers but all their inheriting mappers as well.

2.3 Relationship Configuration

This section describes the `relationship()` function and in depth discussion of its usage. The reference material here continues into the next section, *Collection Configuration and Techniques*, which has additional detail on configuration of collections via `relationship()`.

2.3.1 Basic Relational Patterns

A quick walkthrough of the basic relational patterns. In this section we illustrate the classical mapping using `mapper()` in conjunction with `relationship()`. Then (by popular demand), we illustrate the declarative form using the `declarative` module.

Note that `relationship()` is historically known as `relation()` in older versions of SQLAlchemy.

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': relationship(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': relationship(Child, backref='parent')
})

mapper(Child, child_table)
```

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

Declarative:

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

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", backref="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

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': relationship(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:

```
mapper(Parent, parent_table, properties={
    'child': relationship(Child, backref="parents")
})
```

Declarative:

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

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref="parents")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

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:

```
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'))
)

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

mapper(Child, child_table)
```

Or to turn a one-to-many backref into one-to-one, use the `backref()` function to provide arguments for the reverse side:

```
from sqlalchemy.orm import backref

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)
)

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

mapper(Child, child_table)
```

The second example above as declarative:

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

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref=backref("parent", uselist=False))

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

Many To Many

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

```
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': relationship(Child, secondary=association_table)
})

mapper(Child, right_table)
```

For a bi-directional relationship, both sides of the relationship contain a collection. The `backref` keyword will automatically use the same `secondary` argument for the reverse relationship:

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

With declarative, we still use the `Table` for the `secondary` argument. A class is not mapped to this table, so it remains in its plain schematic form:

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

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

```
class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table,
                    backref="parents")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

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 relationship 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':relationship(Association)
})

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

mapper(Child, right_table)
```

The bi-directional version adds backrefs to both relationships:

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

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

mapper(Child, right_table)
```

Declarative:

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

class Association(Base):
    __tablename__ = 'association'
    left_id = Column(Integer, ForeignKey('left.id'), primary_key=True)
    right_id = Column(Integer, ForeignKey('right.id'), primary_key=True)
    child = relationship("Child", backref="parent_assocs")

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship(Association, backref="parent")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

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 *Association Proxy* extension. This extension allows the configuration of attributes which will access two "hops" with a single access, one "hop" to the associated object, and a second to a target attribute.

Note: When using the association object pattern, it is advisable that the association-mapped table not be used as the `secondary` argument on a `relationship()` elsewhere, unless that `relationship()` contains the option `viewonly=True`. SQLAlchemy otherwise may attempt to emit redundant INSERT and DELETE statements on the same table, if similar state is detected on the related attribute as well as the associated object.

2.3.2 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 relationship as a **self-referential mapper**. In this example, we'll work with a single table called `nodes` to represent a tree structure:

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

```
    Column('parent_id', Integer, ForeignKey('nodes.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 relationship from `nodes` to `nodes`, it assumes one-to-many unless told otherwise:

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

mapper(Node, nodes, properties={
    'children': relationship(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 relationship:

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

And the bi-directional version combines both:

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

For comparison, the declarative version typically uses the inline `id` `Column` attribute to declare remote_side (note the list form is optional when the collection is only one column):

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

class Node(Base):
    __tablename__ = 'nodes'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('nodes.id'))
    data = Column(String(50))
```

```
    children = relationship("Node",
                    backref=backref('parent', remote_side=id)
                )
```

There are several examples included with SQLAlchemy illustrating self-referential strategies; these include *Adjacency List* and *XML Persistence*.

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 nodes.id AS nodes_id, nodes.parent_id AS nodes_parent_id, nodes.data AS nodes_data
FROM nodes, nodes AS nodes_1
WHERE nodes.data = ? AND nodes.parent_id = nodes_1.id AND nodes_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 nodes.id AS nodes_id, nodes.parent_id AS nodes_parent_id, nodes.data AS nodes_data
FROM nodes JOIN nodes AS nodes_1 ON nodes_1.id = nodes.parent_id
WHERE nodes.data = ? AND nodes_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 nodes.id AS nodes_id, nodes.parent_id AS nodes_parent_id, nodes.data AS nodes_data
FROM nodes JOIN nodes AS nodes_1 ON nodes_1.id = nodes.parent_id JOIN nodes AS nodes_2 ON n
WHERE nodes.data = ? AND nodes_1.data = ? AND nodes_2.data = ?
['subchild1', 'child2', 'root']
```

Configuring Eager Loading

Eager loading of relationships 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, or a second statement for all collections at once. SQLAlchemy's joined and subquery 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 relationship, 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': relationship(Node, lazy='joined', join_depth=2)
})

session.query(Node).all()
SELECT nodes_1.id AS nodes_1_id, nodes_1.parent_id AS nodes_1_parent_id, nodes_1.data AS no
FROM nodes LEFT OUTER JOIN nodes AS nodes_2 ON nodes.id = nodes_2.parent_id LEFT OUTER JOIN
[]
```

2.3.3 Specifying Alternate Join Conditions to relationship()

The `relationship()` 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**:

```
one to many/many to one:
------------------------

parent_table -->  parent_table.c.id == child_table.c.parent_id -->  child_table
                               primaryjoin

many to many:
-------------

parent_table -->  parent_table.c.id == secondary_table.c.parent_id -->
                               primaryjoin

                  secondary_table.c.child_id == child_table.c.id --> child_table
                              secondaryjoin
```

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 relationship `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': relationship(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': relationship(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 relationship 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 `relationship()` needs to be told specifically which columns are "foreign" using the `foreign_keys` collection:

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

2.3.4 Rows that point to themselves / Mutually Dependent Rows

This is a very specific case where relationship() must perform an INSERT and a second UPDATE in order to properly populate a row (and vice versa an UPDATE and DELETE in order to delete without violating foreign key constraints). The two use cases are:

- A table contains a foreign key to itself, and a single row will have a foreign key value pointing to its own primary key.
- Two tables each contain a foreign key referencing the other table, with a row in each table referencing the other.

For example:

```
          user
---------------------------------
user_id     name   related_user_id
   1        'ed'          1
```

Or:

```
             widget                                                  entry
-------------------------------------------                ---------------------------------
widget_id      name          favorite_entry_id             entry_id      name       widget_id
   1       'somewidget'            5                          5       'someentry'      1
```

In the first case, a row points to itself. Technically, a database that uses sequences such as PostgreSQL or Oracle can INSERT the row at once using a previously generated value, but databases which rely upon autoincrement-style primary key identifiers cannot. The `relationship()` always assumes a "parent/child" model of row population during flush, so unless you are populating the primary key/foreign key columns directly, `relationship()` needs to use two statements.

In the second case, the "widget" row must be inserted before any referring "entry" rows, but then the "favorite_entry_id" column of that "widget" row cannot be set until the "entry" rows have been generated. In this case, it's typically impossible to insert the "widget" and "entry" rows using just two INSERT statements; an UPDATE must be performed in order to keep foreign key constraints fulfilled. The exception is if the foreign keys are configured as "deferred until commit" (a feature some databases support) and if the identifiers were populated manually (again essentially bypassing `relationship()`).

To enable the UPDATE after INSERT / UPDATE before DELETE behavior on `relationship()`, use the `post_update` flag on *one* of the relationships, preferably the many-to-one side:

```
mapper(Widget, widget, properties={
    'entries':relationship(Entry, primaryjoin=widget.c.widget_id==entry.c.widget_id),
    'favorite_entry':relationship(Entry, primaryjoin=widget.c.favorite_entry_id==entry.c.en
})
```

When a structure using the above mapping is flushed, the "widget" row will be INSERTed minus the "favorite_entry_id" value, then all the "entry" rows will be INSERTed referencing the parent "widget" row, and then an UPDATE statement will populate the "favorite_entry_id" column of the "widget" table (it's one row at a time for the time being).

2.3.5 Mutable Primary Keys / Update Cascades

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 to referenced foreign keys - the values cannot be out of sync for any moment.

For databases that don't support this, such as SQLite and MySQL without their referential integrity options turned on, the `passive_updates` flag can be set to `False`, most preferably on a one-to-many or many-to-many `relationship()`, which instructs SQLAlchemy to issue UPDATE statements individually for objects referenced in the collection, loading them into memory if not already locally present. The `passive_updates` flag can also be `False` in conjunction with ON UPDATE CASCADE functionality, although in that case the unit of work will be issuing extra SELECT and 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

# passive_updates=False *only* needed if the database
# does not implement ON UPDATE CASCADE

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

`passive_updates` is set to `True` by default, indicating that ON UPDATE CASCADE is expected to be in place in the usual case for foreign keys that expect to have a mutating parent key.

`passive_updates=False` may be configured on any direction of relationship, i.e. one-to-many, many-to-one, and many-to-many, although it is much more effective when placed just on the one-to-many or many-to-many side. Configuring the `passive_updates=False` only on the many-to-one side will have only a partial effect, as the unit of work searches only through the current identity map for objects that may be referencing the one with a mutating primary key, not throughout the database.

2.3.6 The `relationship()` API

`sqlalchemy.orm.`**`relationship`**`(`*argument*, *secondary=None*, ***kwargs*`)`

Provide a relationship of a primary Mapper to a secondary Mapper.

Note: `relationship()` is historically known as `relation()` prior to version 0.6.

This corresponds to a parent-child or associative table relationship. The constructed class is an instance of `RelationshipProperty`.

A typical `relationship()`:

```
mapper(Parent, properties={
  'children': relationship(Children)
})
```

Parameters

- **argument** – a class or `Mapper` instance, representing the target of the relationship.
- **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.
- **active_history=False** – When `True`, indicates that the "previous" value for a many-to-one reference should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple many-to-ones only needs to be aware of the "new" value in order to perform a flush. This flag is available for applications that make use of `attributes.get_history()` which also need to know the "previous" value of the attribute. (New in 0.6.6)
- **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 relationship.
- **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 relationship 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 `Session.add()` operation. This cascade applies both to future and past calls to `add()`, meaning new items added to a collection or scalar relationship get placed into the same session as that of the parent, and also applies to items which have been removed from this relationship but are still part of unflushed history.

  – `merge` - cascade the `merge()` operation

  – `expunge` - cascade the `Session.expunge()` operation

  – `delete` - cascade the `Session.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 `Session.expire()` and `refresh()` operations

  – `all` - shorthand for "save-update,merge, refresh-expire, expunge, delete"
- **cascade_backrefs=True** – a boolean value indicating if the `save-update` cascade should operate along a backref event. When set to `False` on a one-to-many relationship that has a many-to-one backref, assigning a persistent object to the many-to-one attribute on a transient object will not add the transient to the session. Similarly, when set to `False` on a many-to-one relationship that has a one-to-many backref, appending a persistent object to the one-to-many collection on a transient object will not add the transient to the session.

  `cascade_backrefs` is new in 0.6.5.
- **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. Behavior of this attribute is described in detail at *Customizing Collection Access*.

- **comparator_factory** – a class which extends `RelationshipProperty.Comparator` which provides custom SQL clause generation for comparison operations.
- **doc** – docstring which will be applied to the resulting descriptor.
- **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. Normally, `relationship()` uses the `ForeignKey` and `ForeignKeyConstraint` objects present within the mapped or secondary `Table` to determine the "foreign" side of the join condition. This is used to construct SQL clauses in order to load objects, as well as to "synchronize" values from primary key columns to referencing foreign key columns. The `foreign_keys` parameter overrides the notion of what's "foreign" in the table metadata, allowing the specification of a list of `Column` objects that should be considered part of the foreign key.

  There are only two use cases for `foreign_keys` - one, when it is not convenient for `Table` metadata to contain its own foreign key metadata (which should be almost never, unless reflecting a large amount of tables from a MySQL MyISAM schema, or a schema that doesn't actually have foreign keys on it). The other is for extremely rare and exotic composite foreign key setups where some columns should artificially not be considered as foreign.
- **innerjoin=False** – when `True`, joined eager loads will use an inner join to join against related tables instead of an outer join. The purpose of this option is strictly one of performance, as inner joins generally perform better than outer joins. This flag can be set to `True` when the relationship references an object via many-to-one using local foreign keys that are not nullable, or when the reference is one-to-one or a collection that is guaranteed to have one or at least one entry.
- **join_depth** – when non-`None`, an integer value indicating how many levels deep "eager" loaders should join 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 loaders will stop chaining when they encounter a the same target mapper which is already higher up in the chain. This option applies both to joined- and subquery- eager loaders.
- **lazy='select'** – specifies how the related items should be loaded. Default value is `select`. Values include:
  - `select` - items should be loaded lazily when the property is first accessed, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
  - `immediate` - items should be loaded as the parents are loaded, using a separate SELECT statement, or identity map fetch for simple many-to-one references. (new as of 0.6.5)
  - `joined` - items should be loaded "eagerly" in the same query as that of the parent, using a JOIN or LEFT OUTER JOIN. Whether the join is "outer" or not is determined by the `innerjoin` parameter.
  - `subquery` - items should be loaded "eagerly" within the same query as that of the parent, using a second SQL statement which issues a JOIN to a subquery of the original statement.
  - `noload` - 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` - the attribute will return a pre-configured `Query` object for all read operations, onto which further filtering operations can be applied before iterating the results. The dynamic collection supports a limited set of mutation operations, allowing `append()` and `remove()`. Changes to the collection will not be visible until flushed to the database, where it is then refetched upon iteration.
  - True - a synonym for 'select'
  - False - a synonyn for 'joined'
  - None - a synonym for 'noload'

  Detailed discussion of loader strategies is at *Relationship Loading Techniques*.

- **load_on_pending=False** – Indicates loading behavior for transient or pending parent objects.

  When set to `True`, causes the lazy-loader to issue a query for a parent object that is not persistent, meaning it has never been flushed. This may take effect for a pending object when autoflush is disabled, or for a transient object that has been "attached" to a `Session` but is not part of its pending collection. Attachment of transient objects to the session without moving to the "pending" state is not a supported behavior at this time.

  Note that the load of related objects on a pending or transient object also does not trigger any attribute change events - no user-defined events will be emitted for these attributes, and if and when the object is ultimately flushed, only the user-specific foreign key attributes will be part of the modified state.

  The load_on_pending flag does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before flush() proceeds. This flag is not not intended for general use.

  New in 0.6.5.

- **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. PostgreSQL, MySQL with InnoDB tables), ON UPDATE CASCADE is required for this operation. The relationship() 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 relationship() 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).

Also see the passive_updates flag on `mapper()`.

A future SQLAlchemy release will provide a "detect" feature for this flag.

- **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 ColumnElement (i.e. WHERE criterion) 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 ColumnElement (i.e. WHERE criterion) 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 relationship(), 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 `relationship()`, 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 relationship 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.

`sqlalchemy.orm.backref(name, **kwargs)`
Create a back reference with explicit arguments, which are the same arguments one can send to `relationship()`.

Used with the *backref* keyword argument to `relationship()` in place of a string argument.

`sqlalchemy.orm.relation(*arg, **kw)`
A synonym for `relationship()`.

2.4 Collection Configuration and Techniques

The `relationship()` function defines a linkage between two classes. When the linkage defines a one-to-many or many-to-many relationship, it's represented as a Python collection when objects are loaded and manipulated. This section presents additional information about collection configuration and techniques.

2.4.1 Working with Large Collections

The default behavior of `relationship()` is to fully load the collection of items in, as according to the loading strategy of the relationship. 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 Relationship Loaders

The most useful by far is the `dynamic_loader()` relationship. This is a variant of `relationship()` 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 relationship supports limited write operations, via the `append()` and `remove()` methods:

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

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

Since the read side of the dynamic relationship always queries the database, changes to the underlying collection will not be visible until the data has been flushed. However, as long as "autoflush" is enabled on the `Session` in use, this will occur automatically each time the collection is about to emit a query.

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

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

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

`sqlalchemy.orm.`**`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, doc=None, order_by=None, comparator_factory=None, query_class=None*`)`

Construct a dynamically-loading mapper property.

This property is similar to `relationship()`, 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 `relationship()` are available here.

Parameters

- **argument** – a class or `Mapper` instance, representing the target of the relationship.
- **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.

Setting Noload

The opposite of the dynamic relationship is simply "noload", specified using `lazy='noload'`:

```
mapper(MyClass, table, properties={
    'children': relationship(MyOtherClass, lazy='noload')
})
```

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': relationship(MyOtherClass, cascade="all, delete-orphan", passive_deletes=T
})
```

When `passive_deletes` is applied, the `children` relationship 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.

2.4.2 Customizing Collection Access

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' : relationship(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, by specifying the `collection_class` option on `relationship()`.

```
# use a set
mapper(Parent, properties={
    'children' : relationship(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_collecti

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

# ...
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)
```

When subclassing `MappedCollection`, user-defined versions of `__setitem__()` or `__delitem__()` should be decorated with `collection.internally_instrumented()`, **if** they call down to those same methods on `MappedCollection`. This because the methods on `MappedCollection` are already instrumented - calling them from within an already instrumented call can cause events to be fired off repeatedly, or inappropriately, leading to internal state corruption in rare cases:

```
from sqlalchemy.orm.collections import MappedCollection,\
                                    collection

class MyMappedCollection(MappedCollection):
    """Use @internally_instrumented when your methods
    call down to already-instrumented methods.

    """

    @collection.internally_instrumented
    def __setitem__(self, key, value, _sa_initiator=None):
        # do something with key, value
        super(MyMappedCollection, self).__setitem__(key, value, _sa_initiator)

    @collection.internally_instrumented
    def __delitem__(self, key, _sa_initiator=None):
        # do something with key
        super(MyMappedCollection, self).__delitem__(key, _sa_initiator)
```

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 relationships, 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 relationships. For example:

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

# ... relationship(..., 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` package for more information and discussion of advanced usage and Python 2.3-compatible decoration options.

Collections API

`sqlalchemy.orm.collections.`**`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 `sqlalchemy.orm.collections.`**`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.

static **`adds`**`(`*arg*`)`

Mark the method as adding an entity to the collection.

Adds "add to collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value. Arguments can be specified positionally (i.e. integer) or by name:

```
@collection.adds(1)
def push(self, item): ...

@collection.adds('entity')
def do_stuff(self, thing, entity=None): ...
```

static **`appender`**`(`*fn*`)`

Tag the method as the collection appender.

The appender method is called with one positional argument: the value to append. The method will be automatically decorated with 'adds(1)' if not already decorated:

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

# or, equivalently
@collection.appender
@collection.adds(1)
def add(self, append): ...

# for mapping type, an 'append' may kick out a previous value
# that occupies that slot.  consider d['a'] = 'foo'- any previous
# value in d['a'] is discarded.
@collection.appender
@collection.replaces(1)
def add(self, entity):
    key = some_key_func(entity)
    previous = None
    if key in self:
        previous = self[key]
    self[key] = entity
    return previous
```

If the value to append is not allowed in the collection, you may raise an exception. Something to remember is that the appender will be called for each object mapped by a database query. If the database contains rows that violate your collection semantics, you will need to get creative to fix the problem, as access via the collection will not work.

If the appender method is internally instrumented, you must also receive the keyword argument '_sa_initiator' and ensure its promulgation to collection events.

static converter (*fn*)

Tag the method as the collection converter.

This optional method will be called when a collection is being replaced entirely, as in:

```
myobj.acollection = [newvalue1, newvalue2]
```

The converter method will receive the object being assigned and should return an iterable of values suitable for use by the `appender` method. A converter must not assign values or mutate the collection, it's sole job is to adapt the value the user provides into an iterable of values for the ORM's use.

The default converter implementation will use duck-typing to do the conversion. A dict-like collection will be convert into an iterable of dictionary values, and other types will simply be iterated:

```
@collection.converter
def convert(self, other): ...
```

If the duck-typing of the object does not match the type of this collection, a TypeError is raised.

Supply an implementation of this method if you want to expand the range of possible types that can be assigned in bulk or perform validation on the values about to be assigned.

static internally_instrumented (*fn*)

Tag the method as instrumented.

This tag will prevent any decoration from being applied to the method. Use this if you are orchestrating your own calls to `collection_adapter()` in one of the basic SQLAlchemy interface methods, or to prevent an automatic ABC method decoration from wrapping your implementation:

```
# normally an 'extend' method on a list-like class would be
# automatically intercepted and re-implemented in terms of
# SQLAlchemy events and append().  your implementation will
# never be called, unless:
@collection.internally_instrumented
def extend(self, items): ...
```

static iterator (*fn*)
Tag the method as the collection remover.

The iterator method is called with no arguments. It is expected to return an iterator over all collection members:

```
@collection.iterator
def __iter__(self): ...
```

static on_link (*fn*)
Tag the method as a the "linked to attribute" event handler.

This optional event handler will be called when the collection class is linked to or unlinked from the InstrumentedAttribute. It is invoked immediately after the '_sa_adapter' property is set on the instance. A single argument is passed: the collection adapter that has been linked, or None if unlinking.

static remover (*fn*)
Tag the method as the collection remover.

The remover method is called with one positional argument: the value to remove. The method will be automatically decorated with `removes_return()` if not already decorated:

```
@collection.remover
def zap(self, entity): ...

# or, equivalently
@collection.remover
@collection.removes_return()
def zap(self, ): ...
```

If the value to remove is not present in the collection, you may raise an exception or return None to ignore the error.

If the remove method is internally instrumented, you must also receive the keyword argument '_sa_initiator' and ensure its promulgation to collection events.

static removes (*arg*)
Mark the method as removing an entity in the collection.

Adds "remove from collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value to be removed. Arguments can be specified positionally (i.e. integer) or by name:

```
@collection.removes(1)
def zap(self, item): ...
```

For methods where the value to remove is not known at call-time, use collection.removes_return.

static removes_return ()
Mark the method as removing an entity in the collection.

Adds "remove from collection" handling to the method. The return value of the method, if any, is considered the value to remove. The method arguments are not inspected:

```
@collection.removes_return()
def pop(self): ...
```

For methods where the value to remove is known at call-time, use collection.remove.

static replaces (*arg*)

Mark the method as replacing an entity in the collection.

Adds "add to collection" and "remove from collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value to be added, and return value, if any will be considered the value to remove.

Arguments can be specified positionally (i.e. integer) or by name:

```
@collection.replaces(2)
def __setitem__(self, index, item): ...
```

`sqlalchemy.orm.collections.`**`collection_adapter`**(*collection*)

Fetch the `CollectionAdapter` for a collection.

`sqlalchemy.orm.collections.`**`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.

`sqlalchemy.orm.collections.`**`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.

class `sqlalchemy.orm.collections.`**`MappedCollection`**(*keyfunc*)

A basic dictionary-based collection class.

Extends dict with the minimal bag semantics that collection classes require. `set` and `remove` are implemented in terms of a keying function: any callable that takes an object and returns an object for use as a dictionary key.

`__init__`(*keyfunc*)

Create a new collection with keying provided by keyfunc.

keyfunc may be any callable any callable that takes an object and returns an object for use as a dictionary key.

The keyfunc will be called every time the ORM needs to add a member by value-only (such as when loading instances from the database) or remove a member. The usual cautions about dictionary keying apply- `keyfunc(object)` should return the same output for the life of the collection. Keying based on mutable properties can result in unreachable instances "lost" in the collection.

remove (*value*, *_sa_initiator=None*)
Remove an item by value, consulting the keyfunc for the key.

set (*value*, *_sa_initiator=None*)
Add an item by value, consulting the keyfunc for the key.

2.5 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
```

2.5.1 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-relationship, 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.

Basic Control of 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 `Query.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 criteria that are specific to one or more subclasses; it makes the subclasses' columns 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.

Advanced Control of Which Tables are Queried

The `Query.with_polymorphic()` method and configuration works fine for simplistic scenarios. However, it currently does not work with any `Query` that selects against individual columns or against multiple classes - it also has to be called at the outset of a query.

For total control of how `Query` joins along inheritance relationships, use the `Table` objects directly and construct joins manually. For example, to query the name of employees with particular criterion:

```
session.query(Employee.name).\
    outerjoin((engineer, engineer.c.employee_id==Employee.employee_id)).\
    outerjoin((manager, manager.c.employee_id==Employee.employee_id)).\
    filter(or_(Engineer.engineer_info=='w', Manager.manager_data=='q'))
```

The base table, in this case the "employees" table, isn't always necessary. A SQL query is always more efficient with fewer joins. Here, if we wanted to just load information specific to managers or engineers, we can instruct `Query` to use only those tables. The `FROM` clause is determined by what's specified in the `Session.query()`, `Query.filter()`, or `Query.select_from()` methods:

```
session.query(Manager.manager_data).select_from(manager)

session.query(engineer.c.id).filter(engineer.c.engineer_info==manager.c.manager_data)
```

Creating Joins to Specific Subtypes

The `of_type()` method is a helper which allows the construction of joins along `relationship()` 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': relationship(Employee)
})
```

When querying from `Company` onto the `Employee` relationship, 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.compan
        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.

2.5.2 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.

2.5.3 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_i
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
[]
```

2.5.4 Using Relationships with Inheritance

Both joined-table and single table inheritance scenarios produce mappings which are usable in `relationship()` functions; that is, it's possible to map a parent object to a child object which is polymorphic. Similarly, inheriting mappers can have `relationship()` objects of their own at any level, which are inherited to each child class. The only requirement for relationships 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': relationship(Employee, backref='company')
})
```

Relationships with Concrete Inheritance

In a concrete inheritance scenario, mapping relationships is more challenging 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.type,
                polymorphic_identity='employee')

mapper(Manager, managers_table,
                inherits=employee_mapper,
                concrete=True,
                polymorphic_identity='manager')

mapper(Engineer, engineers_table,
                inherits=employee_mapper,
                concrete=True,
                polymorphic_identity='engineer')

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

The big limitation with concrete table inheritance is that `relationship()` objects placed on each concrete mapper do **not** propagate to child mappers. If you want to have the same `relationship()` 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':relationship(C, back_populates='many_a')
})
```

```
mapper(B, b_table,inherits=A, concrete=True,
    polymorphic_identity='b',
    properties={
        'some_c':relationship(C, back_populates='many_a')
})
mapper(C, c_table, properties={
    'many_a':relationship(A, collection_class=set, back_populates='some_c'),
})
```

2.5.5 Using Inheritance with Declarative

Declarative makes inheritance configuration more intuitive. See the docs at *Inheritance Configuration*.

2.6 Using the Session

The `orm.mapper()` function and `declarative` extensions are the primary configurational interface for the ORM. Once mappings are configured, the primary usage interface for persistence operations is the `Session`.

2.6.1 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 objects which you've loaded or associated with it during its lifespan. It provides the entrypoint to acquire a `Query` object, which sends queries to the database using the `Session` object's current database connection, populating result rows into objects that are then stored in the `Session`, inside a structure called the Identity Map - a data structure that maintains unique copies of each object, where "unique" means "only one object with a particular primary key".

The `Session` begins in an essentially stateless form. Once queries are issued or other objects are persisted with it, it requests a connection resource from an `Engine` that is associated either with the `Session` itself or with the mapped `Table` objects being operated upon. This connection represents an ongoing transaction, which remains in effect until the `Session` is instructed to commit or roll back its pending state.

All changes to objects maintained by a `Session` are tracked - before the database is queried again or before the current transaction is committed, it **flushes** all pending changes to the database. This is known as the Unit of Work pattern.

When using a `Session`, it's important to note that the objects which are associated with it are **proxy objects** to the transaction being held by the `Session` - there are a variety of events that will cause objects to re-access the database in order to keep synchronized. It is possible to "detach" objects from a `Session`, and to continue using them, though this practice has its caveats. It's intended that usually, you'd re-associate detached objects another `Session` when you want to work with them again, so that they can resume their normal task of representing database state.

2.6.2 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.

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()
```

Above, the `sessionmaker()` call creates a class for us, which we assign to the name `Session`. This class is a subclass of the actual `Session` class, which when instantiated, will use the arguments we've given the function, in this case to use a particular `Engine` for connection resources.

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.

A typical setup will associate the `sessionmaker()` with an `Engine`, so that each `Session` generated will use this `Engine` to acquire connection resources. This association can be set up as in the example above, using the `bind` argument. You can also associate a `Engine` with an existing `sessionmaker()` using the `sessionmaker.configure()` method:

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine

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

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

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

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

you can also associate individual `Session` objects with an `Engine` on each invocation:

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

...or directly with a `Connection`:

```
conn = engine.connect()
session = Session(bind=conn)
```

While the rationale for the above example may not be apparent, the typical usage is in a test fixture that maintains an external transaction - see *Joining a Session into an External Transaction* below for a full example.

2.6.3 Using the Session

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).

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. It then remains in use for the lifespan of a particular database conversation, which includes not just the initial loading of objects but throughout the whole usage of those instances.

  Objects become detached if their owning session is discarded. They are still functional in the detached state if the user has ensured that their state has not been expired before detachment, but they will not be able to represent the current state of database data. Because of this, it's best to consider persisted objects as an extension of the state of a particular `Session`, and to keep that session around until all referenced objects have been discarded.

  An exception to this is when objects are placed in caches or otherwise shared among threads or processes, in which case their detached state can be stored, transmitted, or shared. However, the state of detached objects should still be transferred back into a new `Session` using `Session.add()` or `Session.merge()` before working with the object (or in the case of merge, its state) again.

  It is also very common that a `Session` as well as its associated objects are only referenced by a single thread. Sharing objects between threads is most safely accomplished by sharing their state among multiple instances of those objects, each associated with a distinct `Session` per thread, `Session.merge()` to transfer state between threads. This pattern is not a strict requirement by any means, but it has the least chance of introducing concurrency issues.

  To help with the recommended `Session` -per-thread, `Session` -per-set-of-objects patterns, the `scoped_session()` function is provided which produces a thread-managed registry of `Session` objects. It is commonly used in web applications so that a single global variable can be used to safely represent transactional sessions with sets of objects, localized to a single thread. More on this object is in *Contextual/Thread-local Sessions*.

- 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.

  The `Session` is not designed to be a global object from which everyone consults as a "registry" of objects. That's more the job of a **second level cache**. SQLAlchemy provides a pattern for implementing second level caching using Beaker, via the *Beaker Caching* example.

- 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 (meaning, `autocommit` is set to `True`, not the default setting) for read operations only, it's still not thread-"safe", but you also wont get any catastrophic failures either, since it checks out and returns connections to the connection pool 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.

  A multithreaded application is usually going to want to make usage of `scoped_session()` to transparently manage sessions per thread. More on this at *Contextual/Thread-local Sessions*.

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. 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.

The `Query` object is introduced in great detail in *Object Relational Tutorial*, and further documented in *Querying*.

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 *Cascades*.

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 `load=False`. When this boolean flag is set to `False`, `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.

Merge Tips

`merge()` is an extremely useful method for many purposes. However, it deals with the intricate border between objects that are transient/detached and those that are persistent, as well as the automated transferrence of state. The wide variety of scenarios that can present themselves here often require a more careful approach to the state of objects. Common problems with merge usually involve some unexpected state regarding the object being passed to `merge()`.

Lets use the canonical example of the User and Address objects:

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String(50), nullable=False)
    addresses = relationship("Address", backref="user")

class Address(Base):
    __tablename__ = 'address'

    id = Column(Integer, primary_key=True)
    email_address = Column(String(50), nullable=False)
    user_id = Column(Integer, ForeignKey('user.id'), nullable=False)
```

Assume a `User` object with one `Address`, already persistent:

```
>>> u1 = User(name='ed', addresses=[Address(email_address='ed@ed.com')])
>>> session.add(u1)
>>> session.commit()
```

We now create `a1`, an object outside the session, which we'd like to merge on top of the existing `Address`:

```
>>> existing_a1 = u1.addresses[0]
>>> a1 = Address(id=existing_a1.id)
```

A surprise would occur if we said this:

```
>>> a1.user = u1
>>> a1 = session.merge(a1)
>>> session.commit()
sqlalchemy.orm.exc.FlushError: New instance <Address at 0x1298f50>
with identity key (<class '__main__.Address'>, (1,)) conflicts with
persistent instance <Address at 0x12a25d0>
```

Why is that ? We weren't careful with our cascades. The assignment of `a1.user` to a persistent object cascaded to the backref of `User.addresses` and made our `a1` object pending, as though we had added it. Now we have *two* `Address` objects in the session:

```
>>> a1 = Address()
>>> a1.user = u1
>>> a1 in session
True
>>> existing_a1 in session
True
>>> a1 is existing_a1
False
```

Above, our `a1` is already pending in the session. The subsequent `merge()` operation essentially does nothing. Cascade can be configured via the `cascade` option on `relationship()`, although in this case it would mean removing the `save-update` cascade from the `User.addresses` relationship - and usually, that behavior is extremely convenient. The solution here would usually be to not assign `a1.user` to an object already persistent in the target session.

Note that a new `relationship()` option introduced in 0.6.5, `cascade_backrefs=False`, will also prevent the `Address` from being added to the session via the `a1.user = u1` assignment.

Further detail on cascade operation is at *Cascades*.

Another example of unexpected state:

```
>>> a1 = Address(id=existing_a1.id, user_id=u1.id)
>>> assert a1.user is None
>>> True
>>> a1 = session.merge(a1)
>>> session.commit()
sqlalchemy.exc.IntegrityError: (IntegrityError) address.user_id
may not be NULL
```

Here, we accessed a1.user, which returned its default value of `None`, which as a result of this access, has been placed in the `__dict__` of our object `a1`. Normally, this operation creates no change event, so the `user_id` attribute takes precedence during a flush. But when we merge the `Address` object into the session, the operation is equivalent to:

```
>>> existing_a1.id = existing_a1.id
>>> existing_a1.user_id = u1.id
>>> existing_a1.user = None
```

Where above, both `user_id` and `user` are assigned to, and change events are emitted for both. The `user` association takes precedence, and None is applied to `user_id`, causing a failure.

Most `merge()` issues can be examined by first checking - is the object prematurely in the session ?

```
>>> a1 = Address(id=existing_a1, user_id=user.id)
>>> assert a1 not in session
>>> a1 = session.merge(a1)
```

Or is there state on the object that we don't want ? Examining `__dict__` is a quick way to check:

```
>>> a1 = Address(id=existing_a1, user_id=user.id)
>>> a1.user
>>> a1.__dict__
{'_sa_instance_state': <sqlalchemy.orm.state.InstanceState object at 0x1298d10>,
    'user_id': 1,
    'id': 1,
    'user': None}
>>> # we don't want user=None merged, remove it
>>> del a1.user
>>> a1 = session.merge(a1)
>>> # success
>>> session.commit()
```

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':relationship(Address, cascade="all, delete, delete-orphan")
})
del user.addresses[1]
session.flush()
```

Deleting based on Filter Criterion

The caveat with `Session.delete()` is that you need to have an object handy already in order to delete. The Query includes a `delete()` method which deletes based on filtering criteria:

```
session.query(User).filter(User.id==7).delete()
```

The `Query.delete()` method includes functionality to "expire" objects already in the session which match the criteria. However it does have some caveats, including that "delete" and "delete-orphan" cascades won't be fully expressed for collections which are already loaded. See the API docs for `delete()` for more details.

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.

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

```
session.flush()
```

The "flush-on-Query" aspect of the behavior can be disabled by constructing `sessionmaker()` with the flag `autoflush=False`:

```
Session = sessionmaker(autoflush=False)
```

Additionally, autoflush can be temporarily disabled by setting the `autoflush` flag at any time:

```
mysession = Session()
mysession.autoflush = False
```

Some autoflush-disable recipes are available at DisableAutoFlush.

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. If the Session is not in `autocommit=True` mode, an explicit call to `rollback()` is required after a flush fails, even though the underlying transaction will have been rolled back already - this is so that the overall nesting pattern of so-called "subtransactions" is consistently maintained.

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.

Rolling Back

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

- All transactions are rolled back and all connections 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.

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`.

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 `expunge_all()` (this method was formerly known as `clear()`).

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.

Refreshing / Expiring

The Session normally works in the context of an ongoing transaction (with the default setting of autoflush=False). Most databases offer "isolated" transactions - this refers to a series of behaviors that allow the work within a transaction to remain consistent as time passes, regardless of the activities outside of that transaction. A key feature of a high degree of transaction isolation is that emitting the same SELECT statement twice will return the same results as when it was called the first time, even if the data has been modified in another transaction.

For this reason, the `Session` gains very efficient behavior by loading the attributes of each instance only once. Subsequent reads of the same row in the same transaction are assumed to have the same value. The user application also gains directly from this assumption, that the transaction is regarded as a temporary shield against concurrent changes - a good application will ensure that isolation levels are set appropriately such that this assumption can be made, given the kind of data being worked with.

To clear out the currently loaded state on an instance, the instance or its individual attributes can be marked as "expired", which results in a reload to occur upon next access of any of the instance's attrbutes. The instance can also be immediately reloaded from the database. The `expire()` and `refresh()` methods achieve this:

```
# 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)
```

When an expired object reloads, all non-deferred column-based attributes are loaded in one query. Current behavior for expired relationship-based attributes is that they load individually upon access - this behavior may be enhanced in a future release. When a refresh is invoked on an object, the ultimate operation is equivalent to a `Query.get()`, so any relationships configured with eager loading should also load within the scope of the refresh operation.

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

Note that `expire_all()` is called **automatically** whenever `commit()` or `rollback()` are called. If using the session in its default mode of autocommit=False and with a well-isolated transactional environment (which is provided by most backends with the notable exception of MySQL MyISAM), there is virtually *no reason* to ever call `expire_all()` directly - plenty of state will remain on the current transaction until it is rolled back or committed or otherwise removed.

`refresh()` and `expire()` similarly are usually only necessary when an UPDATE or DELETE has been issued manually within the transaction using `Session.execute()`.

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.

2.6.4 Cascades

Mappers support the concept of configurable *cascade* behavior on `relationship()` 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 `relationship()`:

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

The above mapper specifies two relationships, `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 a session will also be added to that same session. "save-update" cascade also cascades the *pending history* of a relationship()-based attribute, meaning that objects which were removed from a scalar or collection attribute whose changes have not yet been flushed are also placed into the new session - this so that foreign key clear operations and deletions will take place (new in 0.6).

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 `relationship()` 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 `relationship()` is `save-update, merge`.

`save-update` cascade also takes place on backrefs by default. This means that, given a mapping such as this:

```
mapper(Order, order_table, properties={
    'items' : relationship(Item, items_table, backref='order')
})
```

If an `Order` is already in the session, and is assigned to the `order` attribute of an `Item`, the backref appends the `Item` to the `orders` collection of that `Order`, resulting in the `save-update` cascade taking place:

```
>>> o1 = Order()
>>> session.add(o1)
>>> o1 in session
True

>>> i1 = Item()
```

```
>>> i1.order = o1
>>> i1 in o1.orders
True
>>> i1 in session
True
```

This behavior can be disabled as of 0.6.5 using the `cascade_backrefs` flag:

```
mapper(Order, order_table, properties={
    'items' : relationship(Item, items_table, backref='order',
                                cascade_backrefs=False)
})
```

So above, the assignment of `i1.order = o1` will append `i1` to the `orders` collection of `o1`, but will not add `i1` to the session. You can of course `add()` `i1` to the session at a later point. This option may be helpful for situations where an object needs to be kept out of a session until it's construction is completed, but still needs to be given associations to objects which are already persistent in the target session.

2.6.5 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 on next use.
    session.commit()
except:
    # rollback - will immediately go into
    # a new transaction on next use.
    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'
```

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.

Using Subtransactions

A subtransaction, as offered by the `subtransactions=True` flag of `Session.begin()`, is a non-transactional, delimiting construct that allows nesting of calls to `begin()` and `commit()`. It's purpose is to allow the construction of code that can function within a transaction both independently of any external code that starts a transaction, as well as within a block that has already demarcated a transaction. By "non-transactional", we mean that no actual transactional dialogue with the database is generated by this flag beyond that of a single call to `begin()`, regardless of how many times the method is called within a transaction.

The subtransaction feature is in fact intrinsic to any call to `flush()`, which uses it internally to ensure that the series of flush steps are enclosed within a transaction, regardless of the setting of `autocommit` or the presence of an existing transactional context. However, explicit usage of the `subtransactions=True` flag is generally only useful with an application that uses the `Session` in "autocommit=True" mode, and calls `begin()` explicitly in order to demarcate transactions. For this reason the subtransaction feature is not commonly used in an explicit way, except for apps that integrate SQLAlchemy-level transaction control with the transaction control of another library or subsystem. For true, general purpose "nested" transactions, where a rollback affects only a portion of the work which has proceeded, savepoints should be used, documented in *Using SAVEPOINT*.

The feature is the ORM equivalent to the pattern described at *Nesting of Transaction Blocks*, where any number of functions can call `Connection.begin()` and `Transaction.commit()` as though they are the initiator of the transaction, but in fact may be participating in an already ongoing transaction.

As is the case with the non-ORM `Transaction` object, calling `Session.rollback()` rolls back the **entire** transaction, which was initiated by the first call to `Session.begin()` (whether this call was explicit by the end user, or implicit in an `autocommit=False` scenario). However, the `Session` still considers itself to be in a "partially rolled back" state until `Session.rollback()` is called explicitly for each call that was made to `Session.begin()`, where "partially rolled back" means that no further SQL operations can proceed until each level of the transaction has been acounted for, unless the `close()` method is called which cancels all transactional markers. For a full exposition on the rationale for this, please see "But why isn't the one automatic call to ROLLBACK enough ? Why must I ROLLBACK again?". The general theme is that if subtransactions are used as intended, that is, as a means to nest multiple begin/commit pairs, the appropriate rollback calls naturally occur in any case, and allow the session's nesting of transactional pairs to function in a simple and predictable way without the need to guess as to what level is active.

An example of `subtransactions=True` is nearly identical to that of the non-ORM technique. The nesting of transactions, as well as the natural presence of "rollback" for all transactions should an exception occur, is illustrated:

```
# method_a starts a transaction and calls method_b
def method_a(session):
    session.begin(subtransactions=True) # open a transaction.  If there was
                                        # no previous call to begin(), this will
                                        # begin a real transaction (meaning, a
                                        # DBAPI connection is procured, which as
                                        # per the DBAPI specification is in a transactiona
                                        # state ready to be committed or rolled back)
    try:
        method_b(session)
        session.commit()  # transaction is committed here
    except:
        session.rollback() # rolls back the transaction
        raise

# method_b also starts a transaction
def method_b(connection):
    session.begin(subtransactions=True) # open a transaction - this
                                        # runs in the context of method_a()'s
                                        # transaction
    try:
        session.add(SomeObject('bat', 'lala'))
        session.commit()  # transaction is not committed yet
    except:
        session.rollback() # rolls back the transaction, in this case
                           # the one that was initiated in method_a().
        raise

# create a Session and call method_a
session = Session(autocommit=True)
method_a(session)
session.close()
```

Since the `Session.flush()` method uses a subtransaction, a failed flush will always issue a rollback which then affects the state of the outermost transaction (unless a SAVEPOINT is in use). This forces the need to issue `rollback()` for the full operation before subsequent SQL operations can proceed.

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('postgresql://db1')
engine2 = create_engine('postgresql://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()
```

2.6.6 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.

2.6.7 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=MyMappedClass

result = session.execute(select([mytable], mytable.c.id==7), mapper=MyMappedClass)

connection = session.connection(MyMappedClass)
```

2.6.8 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` established), a `Session` can be made to participate within that transaction by just binding the `Session` to that `Connection`. The usual rationale for this is a test suite that allows ORM code to work freely with a `Session`, including the ability to call `Session.commit()`, where afterwards the entire database interaction is rolled back:

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine
from unittest import TestCase

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

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

class SomeTest(TestCase):
    def setUp(self):
        # connect to the database
        self.connection = engine.connect()

        # begin a non-ORM transaction
        self.trans = connection.begin()

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

    def test_something(self):
```

```
        # use the session in tests.

        self.session.add(Foo())
        self.session.commit()

    def tearDown(self):
        # rollback - everything that happened with the
        # Session above (including calls to commit())
        # is rolled back.
        self.trans.rollback()
        self.session.close()
```

Above, we issue `Session.commit()` as well as `Transaction.rollback()`. This is an example of where we take advantage of the `Connection` object's ability to maintain *subtransactions*, or nested begin/commit-or-rollback pairs where only the outermost begin/commit pair actually commits the transaction, or if the outermost block rolls back, everything is rolled back.

2.6.9 The `Session` object and `sessionmaker()` function

`sqlalchemy.orm.session.`**`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:

Parameters

- **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:

  ```
  Session = sessionmaker(binds={
      SomeMappedClass: create_engine('postgresql://engine1'),
      somemapper: create_engine('postgresql://engine2'),
      some_table: create_engine('postgresql://engine3'),
      })
  ```

  Also see the `Session.bind_mapper()` and `Session.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`.

- **_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 as a "two phase" transaction, i.e. using the "two phase" semantics of the database in use along with an XID. 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 `sqlalchemy.orm.session.`**`Session`**`(`*bind=None, autoflush=True, expire_on_commit=True, _enable_transaction_accounting=True, autocommit=False, twophase=False, weak_identity_map=True, binds=None, extension=None, query_cls=<class 'sqlalchemy.orm.query.Query'>*`)`

Manages persistence operations for ORM-mapped objects.

The Session's usage paradigm is described at *Using the Session*.

`__init__``(`*bind=None, autoflush=True, expire_on_commit=True, _enable_transaction_accounting=True, autocommit=False, twophase=False, 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, which is the typical point of entry.

`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. For documentation on subtransactions, please see *Using Subtransactions*.

The `nested` flag begins a SAVEPOINT transaction and is equivalent to calling `begin_nested()`. For documentation on SAVEPOINT transactions, please see *Using SAVEPOINT*.

`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.

For documentation on SAVEPOINT transactions, please see *Using SAVEPOINT*.

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*.

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.

classmethod 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.

By default, the `Session` also expires all database loaded state on all ORM-managed attributes after transaction commit. This so that subsequent operations load the most recent data from the database. This behavior can be disabled using the `expire_on_commit=False` option to `sessionmaker()` or the `Session` constructor.

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` representing results of the statement execution, in the same manner as that of an `Engine` or `Connection`.

`Session.execute()` accepts any executable clause construct, such as `select()`, `insert()`, `update()`, `delete()`, and `text()`, and additionally accepts plain strings that represent SQL statements. If a plain string is passed, it is first converted to a `text()` construct, which here means that bind parameters should be specified using the format `:param`.

The statement is executed within the current transactional context of this `Session`. If this `Session` is set for "autocommit", and no transaction is in progress, an ad-hoc transaction will be created for the life of the result (i.e., a connection is checked out from the connection pool, which is returned when the result object is closed).

If the `Session` is not bound to an `Engine` or `Connection`, the given clause will be inspected for binds (i.e., looking for "bound metadata"). If the session is bound to multiple connectables, the `mapper` keyword argument is typically passed in to specify which bind should be used (since the `Session` keys multiple bind sources to a series of `mapper()` objects). See `get_bind()` for further details on bind resolution.

Parameters

- **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.

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, a query will be issued to the `Session` object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

To expire all objects in the `Session` simultaneously, use `Session.expire_all()`.

The `Session` object's default behavior is to expire all state whenever the `Session.rollback()` or `Session.commit()` methods are called, so that new state can be loaded for the new transaction. For this reason, calling `Session.expire()` only makes sense for the specific case that a non-ORM SQL statement was emitted in the current transaction.

Parameters

- **instance** – The instance to be refreshed.
- **attribute_names** – optional list of string attribute names indicating a subset of attributes to be expired.

expire_all()

Expires all persistent instances within this Session.

When any attributes on a persitent instance is next accessed, a query will be issued using the `Session` object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

To expire individual objects and individual attributes on those objects, use `Session.expire()`.

The `Session` object's default behavior is to expire all state whenever the `Session.rollback()` or `Session.commit()` methods are called, so that new state can be loaded for the new transaction. For this reason, calling `Session.expire_all()` should not be needed when autocommit is `False`, assuming the transaction is isolated.

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.

`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.

A few caveats to this method apply:

- Instances present in the 'dirty' collection may result in a value of `False` when tested with this method. This because while the object may have received attribute set events, there may be no net changes on its state.
- Scalar attributes may not have recorded the "previously" set value when a new value was applied, if the attribute was not loaded, or was expired, at the time the new value was received - in these cases, the attribute is assumed to have a change, even if there is ultimately no net change against its database value. SQLAlchemy in most cases does not need the "old" value when a set event occurs, so it skips the expense of a SQL call if the old value isn't present, based on the assumption that an UPDATE of the scalar value is usually needed, and in those few cases where it isn't, is less expensive on average than issuing a defensive SELECT.

  The "old" value is fetched unconditionally only if the attribute container has the "active_history" flag set to `True`. This flag is set typically for primary key attributes and scalar references that are not a simple many-to-one.

merge (*instance*, *load=True*, ***kw*)
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"`.

See *Merging* for a detailed discussion of merging.

new
The set of all instances marked as 'new' within this `Session`.

classmethod 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*, *lockmode=None*)
Expire and 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.

Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction - usage of `refresh()` usually only makes sense if non-ORM SQL statement were emitted in the ongoing transaction, or if autocommit mode is turned on.

Parameters

- **attribute_names** – optional. An iterable collection of string attribute names indicating a subset of attributes to be refreshed.
- **lockmode** – Passed to the `Query` as used by `with_lockmode()`.

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 subtransactions up to the first real transaction are closed. Subtransactions occur when begin() is called multiple times.

scalar (*clause*, *params=None*, *mapper=None*, ***kw*)
Like execute() but return a scalar result.

2.6.10 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.

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.

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 `ScopedSession.remove()`. This has the effect such that each web request starts fresh with a brand new session, and is the most definitive approach to closing out a request.

It's not strictly necessary to remove the session at the end of the request - other options include calling `Session.close()`, `Session.rollback()`, `Session.commit()` at the end so that the existing session returns its connections to the pool and removes any existing transactional context. Doing nothing is an option too, if individual controller methods take responsibility for ensuring that no transactions remain open after a request ends.

Contextual Session API

`sqlalchemy.orm.`**`scoped_session`**`(`*session_factory*, *scopefunc=None*`)`

Provides thread-local or scoped management of `Session` objects.

This is a front-end function to `ScopedSession`.

Parameters

- **session_factory** – a callable function that produces `Session` instances, such as `sessionmaker()`.
- **scopefunc** – Optional "scope" function which would be passed to the `ScopedRegistry`. If None, the `ThreadLocalRegistry` is used by default.

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()
```

class `sqlalchemy.orm.scoping.`**`ScopedSession`**`(`*session_factory*, *scopefunc=None*`)`

Provides thread-local management of Sessions.

Usage:

```
Session = scoped_session(sessionmaker())
```

... use Session normally.

The internal registry is accessible as well, and by default is an instance of `ThreadLocalRegistry`.

`__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. Deprecated since version 0.5: `ScopedSession.mapper()` is deprecated. Please see http://www.sqlalchemy.org/trac/wiki/UsageRecipes/SessionAwareMapper for information on how to replicate its behavior.

`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.

`remove`()
Dispose of the current contextual session.

class `sqlalchemy.util.`**`ScopedRegistry`**(*createfunc*, *scopefunc*)
A Registry that can store one or multiple instances of a single class on the basis of a "scope" function.

The object implements `__call__` as the "getter", so by calling `myregistry()` the contained object is returned for the current scope.

Parameters

- **createfunc** – a callable that returns a new object to be placed in the registry
- **scopefunc** – a callable that will return a key to store/retrieve an object.

`__init__`(*createfunc*, *scopefunc*)
Construct a new `ScopedRegistry`.

Parameters

- **createfunc** – A creation function that will generate a new value for the current scope, if none is present.
- **scopefunc** – A function that returns a hashable token representing the current scope (such as, current thread identifier).

`clear`()
Clear the current scope, if any.

has()
Return True if an object is present in the current scope.

set(*obj*)
Set the value forthe current scope.

class `sqlalchemy.util.`**`ThreadLocalRegistry`**(*createfunc*)
A `ScopedRegistry` that uses a `threading.local()` variable for storage.

2.6.11 Partitioning Strategies

Vertical Partitioning

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

```
engine1 = create_engine('postgresql://db1')
engine2 = create_engine('postgresql://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()
```

Horizontal Partitioning

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

See the "sharding" example: *Horizontal Sharding*.

2.6.12 Session Utilities

`sqlalchemy.orm.session.`**`make_transient`**(*instance*)
Make the given instance 'transient'.

This will remove its association with any session and additionally will remove its "identity key", such that it's as though the object were newly constructed, except retaining its values. It also resets the "deleted" flag on the state if this object had been explicitly deleted by its session.

Attributes which were "expired" or deferred at the instance level are reverted to undefined, and will not trigger any loads.

`sqlalchemy.orm.session.`**`object_session`**(*instance*)
Return the `Session` to which instance belongs.

If the instance is not a mapped instance, an error is raised.

2.6.13 Attribute and State Management Utilities

These functions are provided by the SQLAlchemy attribute instrumentation API to provide a detailed interface for dealing with instances, attribute values, and history. Some of them are useful when constructing event listener functions, such as those described in *ORM Event Interfaces*.

`sqlalchemy.orm.attributes.`**`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.

`sqlalchemy.orm.attributes.`**`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.

`sqlalchemy.orm.attributes.`**`get_history`**(*obj*, *key*, ***kwargs*)
Return a `History` record for the given object and attribute key.

Parameters

- **obj** – an object whose class is instrumented by the attributes package.
- **key** – string attribute name.
- **kwargs** – Optional keyword arguments currently include the `passive` flag, which indicates if the attribute should be loaded from the database if not already present (`PASSIVE_NO_FETCH`), and if the attribute should be not initialized to a blank value otherwise (`PASSIVE_NO_INITIALIZE`). Default is `PASSIVE_OFF`.

`sqlalchemy.orm.attributes.`**`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.

`sqlalchemy.orm.attributes.`**`instance_state`**()
Return the `InstanceState` for a given object.

`sqlalchemy.orm.attributes.`**`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.

`sqlalchemy.orm.attributes.`**`manager_of_class`**()
Return the `ClassManager` for a given class.

`sqlalchemy.orm.attributes.`**`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.

`sqlalchemy.orm.attributes.`**`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.

class `sqlalchemy.orm.attributes.`**`History`**

A 3-tuple of added, unchanged and deleted values, representing the changes which have occured on an instrumented attribute.

Each tuple member is an iterable sequence.

`added`

Return the collection of items added to the attribute (the first tuple element).

`deleted`

Return the collection of items that have been removed from the attribute (the third tuple element).

`empty``()`

Return True if this `History` has no changes and no existing, unchanged state.

`has_changes``()`

Return True if this `History` has changes.

`non_added``()`

Return a collection of unchanged + deleted.

`non_deleted``()`

Return a collection of added + unchanged.

`sum``()`

Return a collection of added + unchanged + deleted.

`unchanged`

Return the collection of items that have not changed on the attribute (the second tuple element).

`sqlalchemy.orm.attributes.`**`PASSIVE_NO_INITIALIZE`**

Symbol indicating that loader callables should not be fired off, and a non-initialized attribute should remain that way.

`sqlalchemy.orm.attributes.`**`PASSIVE_NO_FETCH`**

Symbol indicating that loader callables should not boe fired off. Non-initialized attributes should be initialized to an empty value.

`sqlalchemy.orm.attributes.`**`PASSIVE_OFF`**

Symbol indicating that loader callables should be executed.

2.7 Querying

This section provides API documentation for the `Query` object and related constructs.

For an in-depth introduction to querying with the SQLAlchemy ORM, please see the *Object Relational Tutorial*.

2.7.1 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 sqlalchemy.orm.query.**Query**(*entities*, *session=None*)

ORM-level SQL construction object.

Query is the source of all SELECT statements generated by the ORM, both those formulated by end-user query operations as well as by high level internal operations such as related collection loading. It features a generative interface whereby successive calls return a new Query object, a copy of the former with additional criteria and options associated with it.

Query objects are normally initially generated using the query() method of Session. For a full walk-through of Query usage, see the *Object Relational Tutorial*.

__init__(*entities*, *session=None*)

add_column(*column*)

Add a column expression to the list of result columns to be returned.

Pending deprecation: add_column() will be superceded by add_columns().

add_columns(**column*)

Add one or more column expressions 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.

as_scalar()

Return the full SELECT statement represented by this Query, converted to a scalar subquery.

Analagous to sqlalchemy.sql._SelectBaseMixin.as_scalar().

New in 0.6.5.

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.

column_descriptions

Return metadata about the columns which would be returned by this Query.

Format is a list of dictionaries:

```
user_alias = aliased(User, name='user2')
q = sess.query(User, User.id, user_alias)

# this expression:
q.columns

# would return:
[
```

```
        {
            'name':'User',
            'type':User,
            'aliased':False,
            'expr':User,
        },
        {
            'name':'id',
            'type':Integer(),
            'aliased':False,
            'expr':User.id,
        },
        {
            'name':'user2',
            'type':User,
            'aliased':True,
            'expr':user_alias
        }
    ]
```

correlate(**args*)

Return a `Query` construct which will correlate the given FROM clauses to that of an enclosing `Query` or `select()`.

The method here accepts mapped classes, `aliased()` constructs, and `mapper()` constructs as arguments, which are resolved into expression constructs, in addition to appropriate expression constructs.

The correlation arguments are ultimately passed to `Select.correlate()` after coercion to expression constructs.

The correlation arguments take effect in such cases as when `Query.from_self()` is used, or when a subquery as returned by `Query.subquery()` is embedded in another `select()` construct.

count()

Return a count of rows this Query would return.

For simple entity queries, count() issues a SELECT COUNT, and will specifically count the primary key column of the first entity only. If the query uses LIMIT, OFFSET, or DISTINCT, count() will wrap the statement generated by this Query in a subquery, from which a SELECT COUNT is issued, so that the contract of "how many rows would be returned?" is honored.

For queries that request specific columns or expressions, count() again makes no assumptions about those expressions and will wrap everything in a subquery. Therefore, `Query.count()` is usually not what you want in this case. To count specific columns, often in conjunction with GROUP BY, use `func.count()` as an individual column expression instead of `Query.count()`. See the ORM tutorial for an example.

delete(*synchronize_session='evaluate'*)

Perform a bulk delete query.

Deletes rows matched by this query from the database.

Parameters

- **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(), or explicitly using expire_all(). 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.

'evaluate' - Evaluate the query's criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, an error is raised. In that case you probably want to use the 'fetch' strategy 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 relationships - 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 `SessionExtension.after_bulk_delete()` event hook.

distinct()
Apply a `DISTINCT` to the query and return the newly resulting `Query`.

enable_assertions(*value*)
Control whether assertions are generated.

When set to False, the returned Query will not assert its state before certain operations, including that LIMIT/OFFSET has not been applied when filter() is called, no criterion exists when get() is called, and no "from_statement()" exists when filter()/order_by()/group_by() etc. is called. This more permissive mode is used by custom Query subclasses to specify criterion or other modifiers outside of the usual usage patterns.

Care should be taken to ensure that the usage pattern is even possible. A statement applied by from_statement() will override any criterion set by filter() or order_by(), for example.

enable_eagerloads(*value*)
Control whether or not eager joins and subqueries are rendered.

When set to False, the returned Query will not render eager joins regardless of `joinedload()`, `subqueryload()` options or mapper-level `lazy='joined'/lazy='subquery'` 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.

execution_options(***kwargs*)
Set non-SQL options which take effect during execution.

The options are the same as those accepted by `sqlalchemy.sql.expression.Executable.executio`

Note that the `stream_results` execution option is enabled automatically if the `yield_per()` method is used.

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.

first() applies a limit of one within the generated SQL, so that only one primary entity row is generated on the server side (note this may consist of multiple result rows if join-loaded collections are present).

Calling `first()` 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.

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 relationship 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" relationship.
- 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" relationship
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" relationship 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), 'emplo
            Engineer.computers)

# join from Articles to Keywords, using the "keywords" attribute.
# assume this is a many-to-many relationship.
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.

label (*name*)
Return the full SELECT statement represented by this `Query`, converted to a scalar subquery with a label of the given name.

Analagous to `sqlalchemy.sql._SelectBaseMixin.label()`.

New in 0.6.5.

limit (*limit*)
Apply a `LIMIT` to the query and return the newly resulting

`Query`.

merge_result (*iterator*, *load=True*)
Merge a result into this Query's Session.

Given an iterator returned by a Query of the same structure as this one, return an identical iterator of results, with all mapped instances merged into the session using Session.merge(). This is an optimized method which will merge all mapped instances, preserving the structure of the result rows and unmapped columns with less method overhead than that of calling Session.merge() explicitly for each value.

The structure of the results is determined based on the column list of this Query - if these do not correspond, unchecked errors will occur.

The 'load' argument is the same as that of Session.merge().

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 object identities are returned, or if multiple rows are returned for a query that does not return object identities.

Note that an entity query, that is, one which selects one or more mapped classes as opposed to individual column attributes, may ultimately represent many rows but only one row of unique entity or entities - this is a successful result for one().

Calling `one()` results in an execution of the underlying query. As of 0.6, `one()` fully fetches all results instead of applying any kind of limit, so that the "unique"-ing of entities does not conceal multiple object identities.

options (**args*)
Return a new Query object, applying the given list of mapper options.

Most supplied options regard changing how column- and relationship-mapped attributes are loaded. See the sections *Deferred Column Loading* and *Relationship Loading Techniques* for reference documentation.

order_by (**criterion*)
apply one or more ORDER BY criterion to the query and return the newly resulting `Query`

All existing ORDER BY settings can be suppressed by passing `None` - this will suppress any ORDER BY configured on mappers as well.

Alternatively, an existing ORDER BY setting on the Query object can be entirely cancelled by passing `False` as the value - use this before calling methods where an ORDER BY is invalid.

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 expire and refresh all instances as they are loaded, or reused from the current `Session`.

`populate_existing()` does not improve behavior when the ORM is used normally - the `Session` object's usual behavior of maintaining a transaction and expiring all attributes after rollback or commit handles object state automatically. This method is not intended for general use.

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 if no rows present. If multiple rows are returned, raises MultipleResultsFound.

```
>>> 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 clause of this `Query` explicitly.

Sending a mapped class or entity here effectively replaces the "left edge" of any calls to `Query.join()`, when no joinpoint is otherwise established - usually, the default "join point" is the leftmost entity in the `Query` object's list of entities to be selected.

Mapped entities or plain `Table` or other selectables can be sent here which will form the default FROM clause.

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.

The statement by default will not have disambiguating labels applied to the construct unless with_labels(True) is called first.

subquery()
return the full SELECT statement represented by this `Query`, embedded within an `Alias`.

Eager JOIN generation within the query is disabled.

The statement by default will not have disambiguating labels applied to the construct unless with_labels(True) is called first.

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 * FROM Z)
```

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='evaluate'*)
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. This option is the most efficient and is reliable once the session is expired, which typically occurs after a commit(), or explicitly using expire_all(). Before the expiration, updated objects may still remain in the session with stale values on their attributes, which can lead to confusing results.

'fetch' - 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' - Evaluate the Query's criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, an exception is raised.

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 relationships - 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 `SessionExtension.after_bulk_update()` event hook.

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
A readonly attribute which returns the current WHERE criterion for this Query.

This returned value is a SQL expression construct, or `None` if no criterion has been established.

with_entities(**entities*)
Return a new `Query` replacing the SELECT list with the given entities.

e.g.:

```
# Users, filtered on some arbitrary criterion
# and then ordered by related email address
q = session.query(User).\
            join(User.address).\
            filter(User.name.like('%ed%')).\
            order_by(Address.email)

# given *only* User.id==5, Address.email, and 'q', what
# would the *next* User in the result be ?
subq = q.with_entities(Address.email).\
            order_by(None).\
            filter(User.id==5).\
            subquery()
q = q.join((subq, subq.c.email < Address.email)).\
            limit(1)
```

New in 0.6.5.

with_hint(*selectable*, *text*, *dialect_name='*'*)
Add an indexing hint for the given entity or selectable to this `Query`.

Functionality is passed straight through to `with_hint()`, with the addition that `selectable` can be a `Table`, `Alias`, or ORM entity / mapped class /etc.

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 filtering criterion that relates the given instance to a child object or collection, using its attribute state as well as an established `relationship()` configuration.

The method uses the `with_parent()` function to generate the clause, the result of which is passed to `Query.filter()`.

Parameters are the same as `with_parent()`, with the exception that the given property can be None, in which case a search is performed against this `Query` object's target mapper.

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='joined' or 'subquery') since those collections will be cleared for a new load when encountered in a subsequent result batch. In the case of 'subquery' loading, the full result for all rows is fetched which generally defeats the purpose of `yield_per()`.

Also note that many DBAPIs do not "stream" results, pre-buffering all rows before making them available, including mysql-python and psycopg2. `yield_per()` will also set the `stream_results` execution option to `True`, which currently is only understood by psycopg2 and causes server side cursors to be used.

2.7.2 ORM-Specific Query Constructs

class `sqlalchemy.orm.`**`aliased`**

The public name of the `AliasedClass` class.

class `sqlalchemy.orm.util.`**`AliasedClass`**(*cls*, *alias=None*, *name=None*)

Represents an "aliased" form of a mapped class for usage with Query.

The ORM equivalent of a `sqlalchemy.sql.expression.alias()` construct, this object mimics the mapped class using a __getattr__ scheme and maintains a reference to a real `Alias` object.

Usage is via the `aliased()` synonym:

```
# find all pairs of users with the same name
user_alias = aliased(User)
session.query(User, user_alias).\
                join((user_alias, User.id > user_alias.id)).\
                filter(User.name==user_alias.name)
```

`sqlalchemy.orm.`**`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 relationship(), or a class-bound descriptor representing a relationship.

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.

`sqlalchemy.orm.`**`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 relationship(), or a class-bound descriptor representing a relationship.

`sqlalchemy.orm.`**`with_parent`**(*instance*, *prop*)

Create filtering criterion that relates this query's primary entity to the given related instance, using established `relationship()` configuration.

The SQL rendered is the same as that rendered when a lazy loader would fire off from the given parent on that attribute, meaning that the appropriate state is taken from the parent object in Python without the need to render joins to the parent table in the rendered statement.

As of 0.6.4, this method accepts parent instances in all persistence states, including transient, persistent, and detached. Only the requisite primary key/foreign key attributes need to be populated. Previous versions didn't work with transient instances.

Parameters

- **instance** – An instance which has some `relationship()`.
- **property** – String property name, or class-bound attribute, which indicates what relationship from the instance should be used to reconcile the parent/child relationship.

2.8 Relationship Loading Techniques

A big part of SQLAlchemy is providing a wide range of control over how related objects get loaded when querying. This behavior can be configured at mapper construction time using the `lazy` parameter to the `relationship()` function, as well as by using options with the `Query` object.

2.8.1 Using Loader Strategies: Lazy Loading, Eager Loading

By default, all inter-object relationships are **lazy loading**. The scalar or collection attribute associated with a `relationship()` contains a trigger which fires the first time the attribute is accessed. This trigger, in all but one case, issues a SQL call at the point of access in order to load the related object or objects:

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

The one case where SQL is not emitted is for a simple many-to-one relationship, when the related object can be identified by its primary key alone and that object is already present in the current `Session`.

This default behavior of "load upon attribute access" is known as "lazy" or "select" loading - the name "select" because a "SELECT" statement is typically emitted when the attribute is first accessed.

In the *Object Relational Tutorial*, we introduced the concept of **Eager Loading**. We used an `option` in conjunction with the `Query` object in order to indicate that a relationship should be loaded at the same time as the parent, within a single SQL query. This option, known as `joinedload()`

```
>>> jack = session.query(User).options(joinedload('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']
```

In addition to "joined eager loading", a second option for eager loading exists, called "subquery eager loading". This kind of eager loading emits an additional SQL statement for each collection requested, aggregated across all parent objects:

```
>>>jack = session.query(User).options(subqueryload('addresses')).filter_by(name='jack').all
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname,
users.password AS users_password
FROM users
WHERE users.name = ?
('jack',)
SELECT addresses.id AS addresses_id, addresses.email_address AS addresses_email_address,
addresses.user_id AS addresses_user_id, anon_1.users_id AS anon_1_users_id
FROM (SELECT users.id AS users_id
FROM users
WHERE users.name = ?) AS anon_1 JOIN addresses ON anon_1.users_id = addresses.user_id
ORDER BY anon_1.users_id, addresses.id
('jack',)
```

The default **loader strategy** for any `relationship()` is configured by the `lazy` keyword argument, which defaults to `select` - this indicates a "select" statement . Below we set it as `joined` so that the `children` relationship is eager loading, using a join:

```
# load the 'children' collection using LEFT OUTER JOIN
mapper(Parent, parent_table, properties={
    'children': relationship(Child, lazy='joined')
})
```

We can also set it to eagerly load using a second query for all collections, using `subquery`:

```
# load the 'children' attribute using a join to a subquery
mapper(Parent, parent_table, properties={
    'children': relationship(Child, lazy='subquery')
})
```

When querying, all three choices of loader strategy are available on a per-query basis, using the `joinedload()`, `subqueryload()` and `lazyload()` query options:

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

# set children to load eagerly with a join
session.query(Parent).options(joinedload('children')).all()

# set children to load eagerly with a second statement
session.query(Parent).options(subqueryload('children')).all()
```

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

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

When using dot-separated names with `joinedload()` or `subqueryload()`, 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 relationships, named `atob` and `btoc`, are both lazy-loading. A statement like the following:

```
session.query(A).options(joinedload('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 joinedloads for both:

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

or more simply just use `joinedload_all()` or `subqueryload_all()`:

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

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

2.8.2 The Zen of Eager Loading

The philosophy behind loader strategies is that any set of loading schemes can be applied to a particular query, and *the results don't change* - only the number of SQL statements required to fully load related objects and collections changes. A particular query might start out using all lazy loads. After using it in context, it might be revealed that particular attributes or collections are always accessed, and that it would be more efficient to change the loader strategy for these. The strategy can be changed with no other modifications to the query, the results will remain identical, but fewer SQL statements would be emitted. In theory (and pretty much in practice), nothing you can do to the `Query` would make it load a different set of primary or related objects based on a change in loader strategy.

The way eagerloading does this, and in particular how `joinedload()` works, is that it creates an anonymous alias of all the joins it adds to your query, so that they can't be referenced by other parts of the query. If the query contains a DISTINCT, or a limit or offset, the statement is first wrapped inside a subquery, and joins are applied to that. As the user, you don't have access to these aliases or subqueries, and you cannot affect what data they will load at query time - a typical beginner misunderstanding is that adding a `Query.order_by()`, naming the joined relationship, would change the order of the collection, or that the entries in the collection as it is loaded could be affected by `Query.filter()`. Not the case ! If you'd like to join from one table to another, filtering or ordering on the joined result, you'd use `Query.join()`. If you then wanted that joined result to populate itself into a related collection, this is also available, via `contains_eager()` option - see *Routing Explicit Joins/Statements into Eagerly Loaded Collections*.

2.8.3 What Kind of Loading to Use ?

Which type of loading to use typically comes down to optimizing the tradeoff between number of SQL executions, complexity of SQL emitted, and amount of data fetched. Lets take two examples, a `relationship()` which references a collection, and a `relationship()` that references a scalar many-to-one reference.

- One to Many Collection
- When using the default lazy loading, if you load 100 objects, and then access a collection on each of them, a total of 101 SQL statements will be emitted, although each statement will typically be a simple SELECT without any joins.
- When using joined loading, the load of 100 objects and their collections will emit only one SQL statement. However, the total number of rows fetched will be equal to the sum of the size of all the collections, plus one extra row for each parent object that has an empty collection. Each row will also contain the full set of columns represented by the parents, repeated for each collection item - SQLAlchemy does not re-fetch these columns other than those of the primary key, however most DBAPIs (with some exceptions) will transmit the full data of each parent over the wire to the client connection in any case. Therefore joined eager loading only makes sense when the size of the collections are relatively small. The LEFT OUTER JOIN can also be performance intensive compared to an INNER join.
- When using subquery loading, the load of 100 objects will emit two SQL statements. The second statement will fetch a total number of rows equal to the sum of the size of all collections. An INNER JOIN is used, and a minimum of parent columns are requested, only the primary keys. So a subquery load makes sense when the collections are larger.
- When multiple levels of depth are used with joined or subquery loading, loading collections-within- collections will multiply the total number of rows fetched in a cartesian fashion. Both forms of eager loading always join from the original parent class.
- Many to One Reference
- When using the default lazy loading, a load of 100 objects will like in the case of the collection emit as many as 101 SQL statements. However - there is a significant exception to this, in that if the many-to-one reference is a simple foreign key reference to the target's primary key, each reference will be checked first in the current identity map using `query.get()`. So here, if the collection of objects references a relatively small set of target objects, or the full set of possible target objects have already been loaded into the session and are strongly referenced, using the default of *lazy='select'* is by far the most efficient way to go.
- When using joined loading, the load of 100 objects will emit only one SQL statement. The join will be a LEFT OUTER JOIN, and the total number of rows will be equal to 100 in all cases. If you know that each parent definitely has a child (i.e. the foreign key reference is NOT NULL), the joined load can be configured with `innerjoin=True`, which is usually specified within the `relationship()`. For a load of objects where there are many possible target references which may have not been loaded already, joined loading with an INNER JOIN is extremely efficient.

- Subquery loading will issue a second load for all the child objects, so for a load of 100 objects there would be two SQL statements emitted. There's probably not much advantage here over joined loading, however, except perhaps that subquery loading can use an INNER JOIN in all cases whereas joined loading requires that the foreign key is NOT NULL.

2.8.4 Routing Explicit Joins/Statements into Eagerly Loaded Collections

The behavior of `joinedload()` 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 joinedload 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 `joinedload()` 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': relationship(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, (...oth
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)
```

2.8.5 Relation Loader API

`sqlalchemy.orm.`**`contains_alias`**`(`*alias*`)`
Return a `MapperOption` that will indicate to the query that the main table has been aliased.

alias is the string name or `Alias` object representing the alias.

`sqlalchemy.orm.`**`contains_eager`**`(`**keys*, ***kwargs*`)`
Return a `MapperOption` that will indicate to the query that the given attribute should be eagerly loaded from columns currently in the query.

Used with `options()`.

The option is used in conjunction with an explicit join that loads the desired rows, i.e.:

```
sess.query(Order).\
        join(Order.user).\
        options(contains_eager(Order.user))
```

The above query would join from the `Order` entity to its related `User` entity, and the returned `Order` objects would have the `Order.user` attribute pre-populated.

contains_eager() also accepts an *alias* argument, which is the string name of an alias, an alias() construct, or an aliased() construct. Use this when the eagerly-loaded rows are to come from an aliased table:

```
user_alias = aliased(User)
sess.query(Order).\
        join((user_alias, Order.user)).\
        options(contains_eager(Order.user, alias=user_alias))
```

See also eagerload() for the "automatic" version of this functionality.

For additional examples of contains_eager() see *Routing Explicit Joins/Statements into Eagerly Loaded Collections*.

sqlalchemy.orm.**eagerload**(**args*, ***kwargs*)
A synonym for joinedload().

sqlalchemy.orm.**eagerload_all**(**args*, ***kwargs*)
A synonym for joinedload_all()

sqlalchemy.orm.**joinedload**(**keys*, ***kw*)
Return a MapperOption that will convert the property of the given name into an joined eager load.

Note: This function is known as eagerload() in all versions of SQLAlchemy prior to version 0.6beta3, including the 0.5 and 0.4 series. eagerload() will remain available for the foreseeable future in order to enable cross-compatibility.

Used with options().

examples:

```
# joined-load the "orders" colleciton on "User"
query(User).options(joinedload(User.orders))

# joined-load the "keywords" collection on each "Item",
# but not the "items" collection on "Order" - those
# remain lazily loaded.
query(Order).options(joinedload(Order.items, Item.keywords))

# to joined-load across both, use joinedload_all()
query(Order).options(joinedload_all(Order.items, Item.keywords))
```

joinedload() also accepts a keyword argument *innerjoin=True* which indicates using an inner join instead of an outer:

```
query(Order).options(joinedload(Order.user, innerjoin=True))
```

Note that the join created by joinedload() is aliased such that no other aspects of the query will affect what it loads. To use joined eager loading with a join that is constructed manually using join() or join(), see contains_eager().

See also: subqueryload(), lazyload()

sqlalchemy.orm.**joinedload_all**(**keys*, ***kw*)
Return a MapperOption that will convert all properties along the given dot-separated path into an joined eager load.

Note: This function is known as `eagerload_all()` in all versions of SQLAlchemy prior to version 0.6beta3, including the 0.5 and 0.4 series. `eagerload_all()` will remain available for the foreseeable future in order to enable cross-compatibility.

Used with `options()`.

For example:

```
query.options(joinedload_all('orders.items.keywords'))...
```

will set all of 'orders', 'orders.items', and 'orders.items.keywords' to load in one joined eager load.

Individual descriptors are accepted as arguments as well:

```
query.options(joinedload_all(User.orders, Order.items, Item.keywords))
```

The keyword arguments accept a flag *innerjoin=True|False* which will override the value of the *innerjoin* flag specified on the relationship().

See also: `subqueryload_all()`, `lazyload()`

`sqlalchemy.orm.`**`lazyload`**`(`*`*keys`*`)`

Return a `MapperOption` that will convert the property of the given name into a lazy load.

Used with `options()`.

See also: `eagerload()`, `subqueryload()`, `immediateload()`

`sqlalchemy.orm.`**`subqueryload`**`(`*`*keys`*`)`

Return a `MapperOption` that will convert the property of the given name into an subquery eager load.

Used with `options()`.

examples:

```
# subquery-load the "orders" colleciton on "User"
query(User).options(subqueryload(User.orders))

# subquery-load the "keywords" collection on each "Item",
# but not the "items" collection on "Order" - those
# remain lazily loaded.
query(Order).options(subqueryload(Order.items, Item.keywords))

# to subquery-load across both, use subqueryload_all()
query(Order).options(subqueryload_all(Order.items, Item.keywords))
```

See also: `joinedload()`, `lazyload()`

`sqlalchemy.orm.`**`subqueryload_all`**`(`*`*keys`*`)`

Return a `MapperOption` that will convert all properties along the given dot-separated path into a subquery eager load.

Used with `options()`.

For example:

```
query.options(subqueryload_all('orders.items.keywords'))...
```

will set all of 'orders', 'orders.items', and 'orders.items.keywords' to load in one subquery eager load.

Individual descriptors are accepted as arguments as well:

```
query.options(subqueryload_all(User.orders, Order.items,
Item.keywords))
```

See also: `joinedload_all()`, `lazyload()`, `immediateload()`

2.9 ORM Event Interfaces

This section describes the various categories of events which can be intercepted within the SQLAlchemy ORM.

For non-ORM event documentation, see *Core Event Interfaces*.

A new version of this API with a significantly more flexible and consistent interface will be available in version 0.7.

2.9.1 Mapper Events

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

```
from sqlalchemy.orm.interfaces import MapperExtension

class MyExtension(MapperExtension):
    def before_insert(self, mapper, connection, instance):
        print "instance %s before insert !" % instance

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])
```

class `sqlalchemy.orm.interfaces.`**`MapperExtension`**

Base implementation for customizing `Mapper` behavior.

New extension classes subclass `MapperExtension` and are specified using the `extension` mapper() argument, which is a single `MapperExtension` or a list of such. A single mapper can maintain a chain of `MapperExtension` objects. When a particular mapping event occurs, the corresponding method on each `MapperExtension` is invoked serially, and each method has the ability to halt the chain from proceeding further.

Each `MapperExtension` method returns the symbol EXT_CONTINUE by default. This symbol generally means "move to the next `MapperExtension` for processing". For methods that return objects like translated rows or new object instances, EXT_CONTINUE means the result of the method should be ignored. In some cases it's required for a default mapper activity to be performed, such as adding a new instance to a result list.

The symbol EXT_STOP has significance within a chain of `MapperExtension` objects that the chain will be stopped when this symbol is returned. Like EXT_CONTINUE, it also has additional significance in some cases that a default mapper activity will not be performed.

`after_delete`(*mapper*, *connection*, *instance*)

Receive an object instance after that instance is deleted.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

`after_insert`(*mapper*, *connection*, *instance*)

Receive an object instance after that instance is inserted.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

after_update (*mapper*, *connection*, *instance*)
Receive an object instance after that instance is updated.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

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 The QueryContext generated from the Query.

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 deleted.

Note that *no* changes to the overall flush plan can be made here; and manipulation of the `Session` will not have the desired effect. To manipulate the `Session` within an extension, use `SessionExtension`.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

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, and manipulation of the `Session` will not have the desired effect. To manipulate the `Session` within an extension, use `SessionExtension`.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

before_update (*mapper*, *connection*, *instance*)
Receive an object instance before that instance is updated.

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 the new value being updated. However *no* changes to the overall flush plan can be made, and manipulation of the `Session` will not have the desired effect. To manipulate the `Session` within an extension, use `SessionExtension`.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

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 The QueryContext generated from the Query.

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*)
Receive an instance when it's constructor has been called, and raised an exception.

This method is only called during a userland construction of an object. It is not called when an object is loaded from the database.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

init_instance (*mapper*, *class_*, *oldinit*, *instance*, *args*, *kwargs*)
Receive an instance when it's constructor is called.

This method is only called during a userland construction of an object. It is not called when an object is loaded from the database.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

instrument_class (*mapper*, *class_*)
Receive a class when the mapper is first constructed, and has applied instrumentation to the mapped class.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

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. Note that some attributes and collections may or may not be loaded or even initialized, depending on what's present in the result rows.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

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. The given row may or may not be a `RowProxy` object - it will always be a dictionary-like object which contains mapped columns as keys. The returned object should also be a dictionary-like object which recognizes mapped columns as keys.

If the ultimate return value is EXT_CONTINUE, the row is not translated.

2.9.2 Session Events

The `SessionExtension` applies plugin points for `Session` objects:

```
from sqlalchemy.orm.interfaces import SessionExtension

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

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

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

class `sqlalchemy.orm.interfaces.`**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.

2.9.3 Attribute Events

`AttributeExtension` is used to listen for set, remove, and append events on individual mapped attributes. It is established on an individual mapped attribute using the *extension* argument, available on `column_property()`, `relationship()`, and others:

```
from sqlalchemy.orm.interfaces import AttributeExtension
from sqlalchemy.orm import mapper, relationship, column_property

class MyAttrExt(AttributeExtension):
    def append(self, state, value, initiator):
        print "append event !"
        return value

    def set(self, state, value, oldvalue, initiator):
        print "set event !"
        return value

mapper(SomeClass, sometable, properties={
    'foo':column_property(sometable.c.foo, extension=MyAttrExt()),
    'bar':relationship(Bar, extension=MyAttrExt())
})
```

Note that the `AttributeExtension` methods `append()` and `set()` need to return the `value` parameter. The returned value is used as the effective value, and allows the extension to change what is ultimately persisted.

class `sqlalchemy.orm.interfaces.`**`AttributeExtension`**

An event handler for individual attribute change events.

AttributeExtension is assembled within the descriptors associated with a mapped class.

`active_history`

indicates that the set() method would like to receive the 'old' value, even if it means firing lazy callables.

Note that `active_history` can also be set directly via `column_property()` and `relationship()`.

`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.

2.9.4 Instrumentation Events and Re-implementation

`InstrumentationManager` can be subclassed in order to receive class instrumentation events as well as to change how class instrumentation proceeds. This class exists for the purposes of integration with other object management frameworks which would like to entirely modify the instrumentation methodology of the ORM, and is not intended for regular usage. One possible exception is the `InstrumentationManager.post_configure_attribute()` method, which can be useful for adding extensions to all mapped attributes, though a much better way to do this will be available in a future release of SQLAlchemy.

For an example of `InstrumentationManager`, see the example *Attribute Instrumentation*.

class `sqlalchemy.orm.interfaces.`**`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*)

2.10 ORM Exceptions

SQLAlchemy ORM exceptions.

sqlalchemy.orm.exc.**ConcurrentModificationError**
alias of StaleDataError

exception sqlalchemy.orm.exc.**DetachedInstanceError**
Bases: sqlalchemy.exc.SQLAlchemyError

An attempt to access unloaded attributes on a mapped instance that is detached.

exception sqlalchemy.orm.exc.**FlushError**
Bases: sqlalchemy.exc.SQLAlchemyError

A invalid condition was detected during flush().

exception sqlalchemy.orm.exc.**MultipleResultsFound**
Bases: sqlalchemy.exc.InvalidRequestError

A single database result was required but more than one were found.

sqlalchemy.orm.exc.**NO_STATE**
Exception types that may be raised by instrumentation implementations.

exception sqlalchemy.orm.exc.**NoResultFound**
Bases: sqlalchemy.exc.InvalidRequestError

A database result was required but none was found.

exception sqlalchemy.orm.exc.**ObjectDeletedError**
Bases: sqlalchemy.exc.InvalidRequestError

An refresh() operation failed to re-retrieve an object's row.

exception sqlalchemy.orm.exc.**StaleDataError**
Bases: sqlalchemy.exc.SQLAlchemyError

An operation encountered database state that is unaccounted for.

Two conditions cause this to happen:

- A flush may have attempted to update or delete rows and an unexpected number of rows were matched during the UPDATE or DELETE statement. Note that when version_id_col is used, rows in UPDATE or DELETE statements are also matched against the current known version identifier.
- A mapped object with version_id_col was refreshed, and the version number coming back from the database does not match that of the object itself.

exception sqlalchemy.orm.exc.**UnmappedClassError**(*cls*, *msg=None*)
Bases: sqlalchemy.orm.exc.UnmappedError

An mapping operation was requested for an unknown class.

exception `sqlalchemy.orm.exc.`**`UnmappedColumnError`**
Bases: `sqlalchemy.exc.InvalidRequestError`

Mapping operation was requested on an unknown column.

exception `sqlalchemy.orm.exc.`**`UnmappedError`**
Bases: `sqlalchemy.exc.InvalidRequestError`

Base for exceptions that involve expected mappings not present.

exception `sqlalchemy.orm.exc.`**`UnmappedInstanceError`**(*obj*, *msg=None*)
Bases: `sqlalchemy.orm.exc.UnmappedError`

An mapping operation was requested for an unknown instance.

2.11 ORM Extensions

SQLAlchemy has a variety of ORM extensions available, which add additional functionality to the core behavior.

2.11.1 Association Proxy

`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 Relationships

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': relationship(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 relationship 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 relationship that contains just the `.keyword` of the related objects. The proxy is a Python property, and unlike the mapper relationship, 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' relationship
    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 relationship, either a collection or scalar.
- You can access and modify both the proxy and the backing relationship. 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 relationship are fine.
- Proxies are lazy, and won't trigger a load of the backing relationship until they are accessed.
- The relationship 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 relationship 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 Relationships

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 relationship.

```
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': relationship(User, backref='user_keywords'),
    'keyword': relationship(Keyword),
})

user = User('log')
kw1  = Keyword('new_from_blammo')

# Creating a UserKeyword association object will add a Keyword.
# the "user" reference assignment in the UserKeyword() constructor
# populates "user_keywords" via backref.
UserKeyword(user, kw1)

# 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 relationship*, 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': relationship(Holding,
        collection_class=attribute_mapped_collection('stock'))
})
mapper(Holding, holdings_table, properties={
    'stock': relationship(Stock),
    'broker': relationship(Broker)
})
```

Above, we've set up the `by_stock` relationship 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

`sqlalchemy.ext.associationproxy.`**`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 relationship 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 relationship, a simple scalar value.

Parameters

- **target_collection** – Name of the relationship attribute we'll proxy to, usually created with `relationship()`.
- **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 relationship 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 relationships, 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 relationship on-demand in this situation.
- ****kw** – Passes along any other keyword arguments to `AssociationProxy`.

class `sqlalchemy.ext.associationproxy.`**`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 'relationship()' 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 relationships.

proxy_bulk_set Optional, use with proxy_factory. See the _set() method for details.

`any`(*criterion=None*, ***kwargs*)

`contains`(*obj*)

`has`(*criterion=None*, ***kwargs*)

`target_class`

The class the proxy is attached to.

2.11.2 Declarative

Synopsis

SQLAlchemy object-relational configuration involves the combination of `Table`, `mapper()`, and class objects to define a mapped class. `declarative` allows all three to be expressed at once within the class declaration. As much as possible, regular SQLAlchemy schema and ORM constructs are used directly, so that configuration between "classical" ORM usage and declarative remain highly similar.

As a simple example:

```
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 returns a new base class from which all mapped classes should inherit. When the class definition is completed, a new `Table` and `mapper()` will have been generated.

The resulting table and mapper are accessible via `__table__` and `__mapper__` attributes on the `SomeClass` class:

```
# access the mapped Table
SomeClass.__table__

# access the Mapper
SomeClass.__mapper__
```

Defining Attributes

In the previous example, the `Column` objects are automatically named with the name of the attribute to which they are assigned.

To name columns explicitly with a name distinct from their mapped attribute, just give the column a name. Below, column "some_table_id" is mapped to the "id" attribute of *SomeClass*, but in SQL will be represented as "some_table_id":

```
class SomeClass(Base):
    __tablename__ = 'some_table'
    id = Column("some_table_id", Integer, primary_key=True)
```

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 = relationship(RelatedInfo)
```

Classes which are constructed using declarative can interact freely with classes that are mapped explicitly with `mapper()`.

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.

Accessing the MetaData

The `declarative_base()` base class contains a `MetaData` object where newly defined `Table` objects are collected. This object is intended to be accessed directly for `MetaData`-specific operations. Such as, to issue CREATE statements for all tables:

```
engine = create_engine('sqlite://')
Base.metadata.create_all(engine)
```

The usual techniques of associating `MetaData:` with `Engine` apply, such as assigning to the `bind` attribute:

```
Base.metadata.bind = create_engine('sqlite://')
```

To associate the engine with the `declarative_base()` at time of construction, the `bind` argument is accepted:

```
Base = declarative_base(bind=create_engine('sqlite://'))
```

declarative_base() can also receive a pre-existing 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 Relationships

Relationships to other classes are done in the usual way, with the added feature that the class specified to relationship() 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 = relationship("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 = relationship(User, primaryjoin=user_id == User.id)
```

In addition to the main argument for relationship(), 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 = relationship("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 = relationship(Address,
                          primaryjoin=Address.user_id==User.id)
```

Configuring Many-to-Many Relationships

Many-to-many relationships are also declared in the same way with declarative as with traditional mappings. The `secondary` argument to `relationship()` is as usual passed a `Table` object, which is typically declared in the traditional way. The `Table` usually shares the `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 = relationship("Keyword", secondary=keywords)
```

As with traditional mapping, its generally not a good idea to use a `Table` as the "secondary" argument which is also mapped to a class, unless the `relationship` is declared with `viewonly=True`. Otherwise, the unit-of-work system may attempt duplicate INSERT and DELETE statements against the underlying table.

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. Here we present using Python 2.6 style properties:

```
class MyClass(Base):
    __tablename__ = 'sometable'

    id = Column(Integer, primary_key=True)

    _attr = Column('attr', String)

    @property
    def attr(self):
        return self._attr

    @attr.setter
    def attr(self, attr):
        self._attr = attr

    attr = synonym('_attr', descriptor=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'

    id = Column(Integer, primary_key=True)
    _attr = Column('attr', String)
```

```
    @synonym_for('_attr')
    @property
    def attr(self):
        return self._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()
```

Defining SQL Expressions

The usage of `column_property()` with Declarative to define load-time, mapped SQL expressions is pretty much the same as that described in *SQL Expressions as Mapped Attributes*. Local columns within the same class declaration can be referenced directly:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String)
    lastname = Column(String)
    fullname = column_property(
        firstname + " " + lastname
    )
```

Correlated subqueries reference the `Column` objects they need either from the local class definition or from remote classes:

```
from sqlalchemy.sql import func

class Address(Base):
    __tablename__ = 'address'

    id = Column('id', Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'))

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    address_count = column_property(
        select([func.count(Address.id)]).\
            where(Address.user_id==id)
    )
```

In the case that the `address_count` attribute above doesn't have access to `Address` when `User` is defined, the `address_count` attribute should be added to `User` when both `User` and `Address` are available (i.e. there is no string based "late compilation" feature like there is with `relationship()` at this time). Note we reference the `id` column attribute of `User` with its class when we are no longer in the declaration of the `User` class:

```
User.address_count = column_property(
    select([func.count(Address.id)]).\
        where(Address.user_id==User.id)
)
```

Table Configuration

Table arguments other than the name, metadata, and mapped Column arguments are specified using the `__table_args__` class attribute. This attribute accommodates both positional as well as keyword arguments that are normally sent to the `Table` constructor. The attribute can be specified in one of two forms. One is as a dictionary:

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = {'mysql_engine':'InnoDB'}
```

The other, a tuple of the form `(arg1, arg2, ..., {kwarg1:value, ...})`, which allows positional arguments to be specified as well (usually constraints):

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = (
            ForeignKeyConstraint(['id'], ['remote_table.id']),
            UniqueConstraint('foo'),
            {'autoload':True}
            )
```

Note that the keyword parameters dictionary is required in the tuple form even if empty.

Using a Hybrid Approach with __table__

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))
    )
```

`__table__` provides a more focused point of control for establishing table metadata, while still getting most of the benefits of using declarative. An application that uses reflection might want to load table metadata elsewhere and simply pass it to declarative classes:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
Base.metadata.reflect(some_engine)

class User(Base):
    __table__ = metadata.tables['user']

class Address(Base):
    __table__ = metadata.tables['address']
```

Some configuration schemes may find it more appropriate to use `__table__`, such as those which already take advantage of the data-driven nature of `Table` to customize and/or automate schema definition. See the wiki example NamingConventions for one such example.

Mapper Configuration

Declarative makes use of the `mapper()` function internally when it creates the mapping to the declared table. The options for `mapper()` are passed directly through via the `__mapper_args__` class attribute. As always, arguments which reference locally mapped columns can reference them directly from within the class declaration:

```
from datetime import datetime

class Widget(Base):
    __tablename__ = 'widgets'

    id = Column(Integer, primary_key=True)
    timestamp = Column(DateTime, nullable=False)

    __mapper_args__ = {
                    'version_id_col': timestamp,
                    'version_id_generator': lambda v:datetime.now()
                }
```

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}
```

Mixin Classes

A common need when using `declarative` is to share some functionality, often a set of columns, across many classes. The normal Python idiom would be to put this common code into a base class and have all the other classes subclass this class.

When using `declarative`, this need is met by using a "mixin class". A mixin class is one that isn't mapped to a table and doesn't subclass the declarative `Base`. For example:

```
class MyMixin(object):

    __table_args__ = {'mysql_engine': 'InnoDB'}
    __mapper_args__= {'always_refresh': True}

    id =  Column(Integer, primary_key=True)


class MyModel(Base,MyMixin):
    __tablename__ = 'test'

    name = Column(String(1000))
```

Where above, the class `MyModel` will contain an "id" column as well as `__table_args__` and `__mapper_args__` defined by the `MyMixin` mixin class.

Mixing in Columns

The most basic way to specify a column on a mixin is by simple declaration:

```
class TimestampMixin(object):
    created_at = Column(DateTime, default=func.now())

class MyModel(Base, TimestampMixin):
    __tablename__ = 'test'

    id =  Column(Integer, primary_key=True)
    name = Column(String(1000))
```

Where above, all declarative classes that include `TimestampMixin` will also have a column `created_at` that applies a timestamp to all row insertions.

Those familiar with the SQLAlchemy expression language know that the object identity of clause elements defines their role in a schema. Two `Table` objects `a` and `b` may both have a column called `id`, but the way these are differentiated is that `a.c.id` and `b.c.id` are two distinct Python objects, referencing their parent tables `a` and `b` respectively.

In the case of the mixin column, it seems that only one `Column` object is explicitly created, yet the ultimate `created_at` column above must exist as a distinct Python object for each separate destination class. To accomplish this, the declarative extension creates a **copy** of each `Column` object encountered on a class that is detected as a mixin.

This copy mechanism is limited to simple columns that have no foreign keys, as a `ForeignKey` itself contains references to columns which can't be properly recreated at this level. For columns that have foreign keys, as well as for the variety of mapper-level constructs that require destination-explicit context, the `declared_attr()` decorator (renamed from `sqlalchemy.util.classproperty` in 0.6.5) is provided so that patterns common to many classes can be defined as callables:

```
from sqlalchemy.ext.declarative import declared_attr

class ReferenceAddressMixin(object):
    @declared_attr
    def address_id(cls):
        return Column(Integer, ForeignKey('address.id'))

class User(Base, ReferenceAddressMixin):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
```

Where above, the `address_id` class-level callable is executed at the point at which the `User` class is constructed, and the declarative extension can use the resulting `Column` object as returned by the method without the need to copy it.

Columns generated by `declared_attr()` can also be referenced by `__mapper_args__` to a limited degree, currently by `polymorphic_on` and `version_id_col`, by specifying the classdecorator itself into the dictionary - the declarative extension will resolve them at class construction time:

```
class MyMixin:
    @declared_attr
    def type_(cls):
        return Column(String(50))

    __mapper_args__= {'polymorphic_on':type_}

class MyModel(Base,MyMixin):
    __tablename__='test'
    id =  Column(Integer, primary_key=True)
```

Mixing in Relationships

Relationships created by `relationship()` are provided with declarative mixin classes exclusively using the `declared_attr()` approach, eliminating any ambiguity which could arise when copying a relationship and its possibly column-bound contents. Below is an example which combines a foreign key column and a relationship so that two classes `Foo` and `Bar` can both be configured to reference a common target class via many-to-one:

```
class RefTargetMixin(object):
    @declared_attr
    def target_id(cls):
        return Column('target_id', ForeignKey('target.id'))

    @declared_attr
    def target(cls):
        return relationship("Target")

class Foo(Base, RefTargetMixin):
    __tablename__ = 'foo'
    id = Column(Integer, primary_key=True)

class Bar(Base, RefTargetMixin):
    __tablename__ = 'bar'
    id = Column(Integer, primary_key=True)

class Target(Base):
    __tablename__ = 'target'
    id = Column(Integer, primary_key=True)
```

`relationship()` definitions which require explicit primaryjoin, order_by etc. expressions should use the string forms for these arguments, so that they are evaluated as late as possible. To reference the mixin class in these expressions, use the given `cls` to get it's name:

```
class RefTargetMixin(object):
    @declared_attr
    def target_id(cls):
        return Column('target_id', ForeignKey('target.id'))

    @declared_attr
    def target(cls):
        return relationship("Target",
            primaryjoin="Target.id==%s.target_id" % cls.__name__
        )
```

Mixing in deferred(), column_property(), etc.

Like `relationship()`, all `MapperProperty` subclasses such as `deferred()`, `column_property()`, etc. ultimately involve references to columns, and therefore, when used with declarative mixins, have the `declared_attr()` requirement so that no reliance on copying is needed:

```
class SomethingMixin(object):

    @declared_attr
    def dprop(cls):
        return deferred(Column(Integer))
```

```
class Something(Base, SomethingMixin):
    __tablename__ = "something"
```

Controlling table inheritance with mixins

The `__tablename__` attribute in conjunction with the hierarchy of classes involved in a declarative mixin scenario controls what type of table inheritance, if any, is configured by the declarative extension.

If the `__tablename__` is computed by a mixin, you may need to control which classes get the computed attribute in order to get the type of table inheritance you require.

For example, if you had a mixin that computes `__tablename__` but where you wanted to use that mixin in a single table inheritance hierarchy, you can explicitly specify `__tablename__` as `None` to indicate that the class should not have a table mapped:

```
from sqlalchemy.ext.declarative import declared_attr

class Tablename:
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

class Person(Base,Tablename):
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = None
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    primary_language = Column(String(50))
```

Alternatively, you can make the mixin intelligent enough to only return a `__tablename__` in the event that no table is already mapped in the inheritance hierarchy. To help with this, a `has_inherited_table()` helper function is provided that returns `True` if a parent class already has a mapped table.

As an example, here's a mixin that will only allow single table inheritance:

```
from sqlalchemy.ext.declarative import declared_attr
from sqlalchemy.ext.declarative import has_inherited_table

class Tablename:
    @declared_attr
    def __tablename__(cls):
        if has_inherited_table(cls):
            return None
        return cls.__name__.lower()

class Person(Base,Tablename):
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
```

```
    primary_language = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

If you want to use a similar pattern with a mix of single and joined table inheritance, you would need a slightly different mixin and use it on any joined table child classes in addition to their parent classes:

```
from sqlalchemy.ext.declarative import declared_attr
from sqlalchemy.ext.declarative import has_inherited_table

class Tablename:
    @declared_attr
    def __tablename__(cls):
        if (has_inherited_table(cls) and
            Tablename not in cls.__bases__):
            return None
        return cls.__name__.lower()

class Person(Base,Tablename):
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

# This is single table inheritance
class Engineer(Person):
    primary_language = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}

# This is joined table inheritance
class Manager(Person,Tablename):
    id = Column(Integer, ForeignKey('person.id'), primary_key=True)
    preferred_recreation = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

Combining Table/Mapper Arguments from Multiple Mixins

In the case of `__table_args__` or `__mapper_args__` specified with declarative mixins, you may want to combine some parameters from several mixins with those you wish to define on the class iteself. The `declared_attr()` decorator can be used here to create user-defined collation routines that pull from multiple collections:

```
from sqlalchemy.ext.declarative import declared_attr

class MySQLSettings:
    __table_args__ = {'mysql_engine':'InnoDB'}

class MyOtherMixin:
    __table_args__ = {'info':'foo'}

class MyModel(Base,MySQLSettings,MyOtherMixin):
    __tablename__='my_model'

    @declared_attr
    def __table_args__(self):
        args = dict()
        args.update(MySQLSettings.__table_args__)
```

```
        args.update(MyOtherMixin.__table_args__)
        return args

    id =  Column(Integer, primary_key=True)
```

Defining Indexes in Mixins

If you need to define a multi-column index that applies to all tables that make use of a particular mixin, you will need to do this in a metaclass as shown in the following example:

```
from sqlalchemy.ext.declarative import DeclarativeMeta

class MyMixinMeta(DeclarativeMeta):

    def __init__(cls,*args,**kw):
        if getattr(cls,'_decl_class_registry',None) is None:
            return
        super(MyMeta,cls).__init__(*args,**kw)
        # Index creation done here
        Index('test',cls.a,cls.b)

class MyMixin(object):
    __metaclass__=MyMixinMeta
    a =  Column(Integer)
    b =  Column(Integer)

class MyModel(Base,MyMixin):
    __tablename__ = 'atable'
    c =  Column(Integer,primary_key=True)
```

Using multiple Mixins that require Metaclasses

If you end up in a situation where you need to use multiple mixins and more than one of them uses a metaclass to, for example, create a multi-column index, then you will need to create a metaclass that correctly combines the actions of the other metaclasses. For example:

```
class MyMeta1(DeclarativeMeta):

    def __init__(cls,*args,**kw):
        if getattr(cls,'_decl_class_registry',None) is None:
            return
        super(MyMeta1,cls).__init__(*args,**kw)
        Index('ab',cls.a,cls.b)

class MyMixin1(object):
    __metaclass__=MyMeta1
    a =  Column(Integer)
    b =  Column(Integer)

class MyMeta2(DeclarativeMeta):

    def __init__(cls,*args,**kw):
        if getattr(cls,'_decl_class_registry',None) is None:
```

```
            return
        super(MyMeta2,cls).__init__(*args,**kw)
        Index('cd',cls.c,cls.d)

class MyMixin2(object):
    __metaclass__=MyMeta2
    c =  Column(Integer)
    d =  Column(Integer)

class CombinedMeta(MyMeta1,MyMeta2):
    # This is needed to successfully combine
    # two mixins which both have metaclasses
    pass

class MyModel(Base,MyMixin1,MyMixin2):
    __tablename__ = 'awooooga'
    __metaclass__ = CombinedMeta
    z =  Column(Integer,primary_key=True)
```

For this reason, if a mixin requires a custom metaclass, this should be mentioned in any documentation of that mixin to avoid confusion later down the line.

Class Constructor

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')
```

Sessions

Note that `declarative` does nothing special with sessions, and is only intended as an easier way to configure mappers and `Table` objects. A typical application setup using `scoped_session()` might look like:

```
engine = create_engine('postgresql://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.

API Reference

`sqlalchemy.ext.declarative.`**`declarative_base`**(*bind=None, metadata=None, mapper=None, cls=<type 'object'>, name='Base', constructor=<function __init__ at 0x8166758>, metaclass=<class 'sqlalchemy.ext.declarative.DeclarativeMeta'>*)

Construct a base class for declarative class definitions.

The new base class will be given a metaclass that produces appropriate `Table` objects and makes the appropriate `mapper()` calls based on the information provided declaratively in the class and any subclasses of the class.

Parameters

- **bind** – An optional Connectable, will be assigned the bind attribute on the MetaData instance.
- **metadata** – An optional MetaData instance. All Table objects implicitly declared by subclasses of the base will share this MetaData. A MetaData instance will be created 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 class or tuple of classes.
- **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_constructor(), an __init__ implementation that assigns **kwargs for declared fields and relationships to an instance. If None is supplied, no __init__ will be provided and construction will fall back to cls.__init__ by way of the 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.

class sqlalchemy.ext.declarative.**declared_attr**(*fget*, **arg*, ***kw*)

Mark a class-level method as representing the definition of a mapped property or special declarative member name.

Note: @declared_attr is available as sqlalchemy.util.classproperty for SQLAlchemy versions 0.6.2, 0.6.3, 0.6.4.

@declared_attr turns the attribute into a scalar-like property that can be invoked from the uninstantiated class. Declarative treats attributes specifically marked with @declared_attr as returning a construct that is specific to mapping or declarative table configuration. The name of the attribute is that of what the non-dynamic version of the attribute would be.

@declared_attr is more often than not applicable to mixins, to define relationships that are to be applied to different implementors of the class:

```
class ProvidesUser(object):
    "A mixin that adds a 'user' relationship to classes."

    @declared_attr
    def user(self):
        return relationship("User")
```

It also can be applied to mapped classes, such as to provide a "polymorphic" scheme for inheritance:

```
class Employee(Base):
    id = Column(Integer, primary_key=True)
    type = Column(String(50), nullable=False)

    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

    @declared_attr
    def __mapper_args__(cls):
```

```
        if cls.__name__ == 'Employee':
            return {
                    "polymorphic_on":cls.type,
                    "polymorphic_identity":"Employee"
            }
        else:
            return {"polymorphic_identity":cls.__name__}
```

`sqlalchemy.ext.declarative.`**`_declarative_constructor`**`(`*self*, ***kwargs*`)`

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

`sqlalchemy.ext.declarative.`**`has_inherited_table`**`(`*cls*`)`

Given a class, return True if any of the classes it inherits from has a mapped table, otherwise return False.

`sqlalchemy.ext.declarative.`**`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))
```

`sqlalchemy.ext.declarative.`**`comparable_using`**`(`*comparator_factory*`)`

Decorator, allow a Python @property to be used in query criteria.

This is a decorator front end to `comparable_property()` that 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)
```

`sqlalchemy.ext.declarative.`**`instrument_declarative`**`(`*cls*, *registry*, *metadata*`)`

Given a class, configure the class declaratively, using the given registry, which can be any dictionary, and MetaData object.

2.11.3 Ordering List

A custom list that manages index/position information for its children.

author Jason Kirtland

`orderinglist` is a helper for mutable ordered relationships. It will intercept list operations performed on a relationship collection and automatically synchronize changes in list position with an attribute on the related objects. (See *advdatamapping_entitycollections* 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': relationship(Bullet, order_by=[bullets_table.c.position])
 })
 mapper(Bullet, bullets_table)
```

The standard relationship 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': relationship(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 relationships (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 relationship.

> **Warning:** `ordering_list` only provides limited functionality when a primary key column or unique column is the target of the sort. Since changing the order of entries often means that two rows must trade values, this is not possible when the value is constrained by a primary key or unique constraint, since one of the rows would temporarily have to point to a third available value so that the other row could take its old value. `ordering_list` doesn't do any of this for you, nor does SQLAlchemy itself.

`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.

API Reference

`sqlalchemy.ext.orderinglist.`**`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 relationship'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.

2.11.4 Horizontal Sharding

Horizontal sharding support.

Defines a rudimental 'horizontal sharding' system which allows a Session to distribute queries and persistence operations across multiple databases.

For a usage example, see the *Horizontal Sharding* example included in the source distrbution.

API Documentation

class `sqlalchemy.ext.horizontal_shard.`**`ShardedSession`**`(`*shard_chooser*, *id_chooser*, *query_chooser*, *shards=None*, ***kwargs*`)`

`__init__``(`*shard_chooser*, *id_chooser*, *query_chooser*, *shards=None*, ***kwargs*`)`

Construct a ShardedSession.

Parameters

- **shard_chooser** – A callable which, passed a Mapper, a mapped instance, and possibly a SQL clause, returns a shard ID. This id may be based off of the attributes present within the object, or on some round-robin scheme. If the scheme is based on a selection, it should set whatever state on the instance to mark it in the future as participating in that shard.

- **id_chooser** – A callable, passed a query and a tuple of identity values, which should return a list of shard ids where the ID might reside. The databases will be queried in the order of this listing.
- **query_chooser** – For a given Query, returns the list of shard_ids where the query should be issued. Results from all shards returned will be combined together into a single listing.
- **shards** – A dictionary of string shard names to `Engine` objects.

class `sqlalchemy.ext.horizontal_shard.`**`ShardedQuery`**`(`**args*, ***kwargs*`)`

`__init__``(`**args*, ***kwargs*`)`

`set_shard``(`*shard_id*`)`

return a new query, limited to a single shard ID.

all subsequent operations with the returned query will be against the single shard regardless of other state.

2.11.5 SqlSoup

Introduction

SqlSoup provides a convenient way to access existing database tables without having to declare table or mapper classes ahead of time. It is built on top of the SQLAlchemy ORM and provides a super-minimalistic interface to an existing database.

SqlSoup effectively provides a coarse grained, alternative interface to working with the SQLAlchemy ORM, providing a "self configuring" interface for extremely rudimental operations. It's somewhat akin to a "super novice mode" version of the ORM. While SqlSoup can be very handy, users are strongly encouraged to use the full ORM for non-trivial applications.

Suppose we have a database with users, books, and loans tables (corresponding to the PyWebOff dataset, if you're curious).

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 engine:

```
>>> db = SqlSoup(engine)
```

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=None,admin=0),
    MappedUsers(name=u'Bhargan Basepair',email=u'basepair@example.edu',
            password=u'basepair',classname=None,admin=1)
]
```

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',classname=None,admin=1),
    MappedUsers(name=u'Joe Student',email=u'student@example.edu',
        password=u'student',classname=None,admin=0)
]
```

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=None,admin=0),
    MappedUsers(name=u'Bhargan Basepair',email=u'basepair@example.edu',
        password=u'basepair',classname=None,admin=1)
]
```

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',classname=None,admin=1)
```

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',classname=None,admin=1)
```

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',classname=None,admin=1)
```

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 *Querying*.

Modifying objects

Modifying objects is intuitive:

```
>>> user = _
>>> user.email = 'basepair+nospam@example.edu'
>>> db.commit()
```

(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 commit.)

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)

>>> loan = db.loans.filter_by(book_id=2, user_name='Bhargan Basepair').one()
>>> db.delete(loan)
>>> db.commit()
```

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.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, 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=None,admin=0,book_id=1,
        user_name=u'Joe Student',loan_date=datetime.datetime(2006, 7, 12, 0, 0))
]
```

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=None,admin=0,book_id=1,
        user_name=u'Joe Student',loan_date=datetime.datetime(2006, 7, 12, 0, 0),
        id=1,title=u'Mustards I Have Known',published_year=u'1989',
        authors=u'Jones')
]
```

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'loans_book_id',
    u'loans_user_name', u'loans_loan_date']
```

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_name', u'loans_loan_date']
```

Relationships

You can define relationships 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, 12, 0, 0))]
```

```
>>> db.users.filter(~db.users.loans.any()).all()
[MappedUsers(name=u'Bhargan Basepair',
        email='basepair+nospam@example.edu',
        password=u'basepair',classname=None,admin=1)]
```

relate can take any options that the relationship 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

Sessions, Transations and Application Integration

Note: please read and understand this section thoroughly before using SqlSoup in any web application.

SqlSoup uses a ScopedSession to provide thread-local sessions. You can get a reference to the current one like this:

```
>>> session = db.session
```

The default session is available at the module level in SQLSoup, via:

```
>>> from sqlalchemy.ext.sqlsoup import Session
```

The configuration of this session is `autoflush=True`, `autocommit=False`. This means when you work with the SqlSoup object, you need to call `db.commit()` in order to have changes persisted. You may also call `db.rollback()` to roll things back.

Since the SqlSoup object's Session automatically enters into a transaction as soon as it's used, it is *essential* that you call `commit()` or `rollback()` on it when the work within a thread completes. This means all the guidelines for web application integration at *Lifespan of a Contextual Session* must be followed.

The SqlSoup object can have any session or scoped session configured onto it. This is of key importance when integrating with existing code or frameworks such as Pylons. If your application already has a `Session` configured, pass it to your SqlSoup object:

```
>>> from myapplication import Session
>>> db = SqlSoup(session=Session)
```

If the `Session` is configured with `autocommit=True`, use `flush()` instead of `commit()` to persist changes - in this case, the `Session` closes out its transaction immediately and no external management is needed. `rollback()` is also not available. Configuring a new SQLSoup object in "autocommit" mode looks like:

```
>>> from sqlalchemy.orm import scoped_session, sessionmaker
>>> db = SqlSoup('sqlite://', session=scoped_session(sessionmaker(autoflush=False, expire_o
```

Mapping arbitrary Selectables

SqlSoup can map any SQLAlchemy `Selectable` with the map method. Let's map an `expression.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 construct, described in *Using Text*. You can also execute textual SQL directly using the *execute()* method, which corresponds to the *execute()* method on the underlying *Session*. Expressions here are expressed like `text()` constructs, using named parameters with colons:

```
>>> rp = db.execute('select name, email from users where name like :name order by name', na
>>> for name, email in rp.fetchall(): print name, email
Bhargan Basepair basepair+nospam@example.edu
```

Or you can get at the current transaction's connection using *connection()*. This is the raw connection object which can accept any sort of SQL expression or raw SQL string passed to the database:

```
>>> conn = db.connection()
>>> conn.execute("'select name, email from users where name like ? order by name'", '%Bharg
```

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.

SqlSoup API

class `sqlalchemy.ext.sqlsoup.`**`SqlSoup`**(*engine_or_metadata*, *base=<type 'object'>*, *session=None*)

Represent an ORM-wrapped database resource.

`__init__`(*engine_or_metadata*, *base=<type 'object'>*, *session=None*)

Initialize a new `SqlSoup`.

Parameters

- **engine_or_metadata** – a string database URL, `Engine` or `MetaData` object to associate with. If the argument is a `MetaData`, it should be *bound* to an `Engine`.
- **base** – a class which will serve as the default class for returned mapped classes. Defaults to `object`.
- **session** – a `ScopedSession` or `Session` with which to associate ORM operations for this `SqlSoup` instance. If `None`, a `ScopedSession` that's local to this module is used.

`bind`

The `Engine` associated with this `SqlSoup`.

`clear`()

Synonym for `SqlSoup.expunge_all()`.

`commit`()

Commit the current transaction.

See `Session.commit()`.

`connection`()

Return the current `Connection` in use by the current transaction.

`delete`(*instance*)

Mark an instance as deleted.

`engine`

The `Engine` associated with this `SqlSoup`.

entity (*attr*, *schema=None*)
Return the named entity from this `SqlSoup`, or create if not present.

For more generalized mapping, see `map_to()`.

execute (*stmt*, ***params*)
Execute a SQL statement.

The statement may be a string SQL string, an `expression.select()` construct, or an `expression.text()` construct.

expunge (*instance*)
Remove an instance from the `Session`.

See `Session.expunge()`.

expunge_all ()
Clear all objects from the current `Session`.

See `Session.expunge_all()`.

flush ()
Flush pending changes to the database.

See `Session.flush()`.

join (*left*, *right*, *onclause=None*, *isouter=False*, *base=None*, ***mapper_args*)
Create an `expression.join()` and map to it.

The class and its mapping are not cached and will be discarded once dereferenced (as of 0.6.6).

Parameters

- **left** – a mapped class or table object.
- **right** – a mapped class or table object.
- **onclause** – optional "ON" clause construct..
- **isouter** – if True, the join will be an OUTER join.
- **base** – a Python class which will be used as the base for the mapped class. If `None`, the "base" argument specified by this `SqlSoup` instance's constructor will be used, which defaults to `object`.
- **mapper_args** – Dictionary of arguments which will be passed directly to `orm.mapper()`.

map (*selectable*, *base=None*, ***mapper_args*)
Map a selectable directly.

The class and its mapping are not cached and will be discarded once dereferenced (as of 0.6.6).

Parameters

- **selectable** – an `expression.select()` construct.
- **base** – a Python class which will be used as the base for the mapped class. If `None`, the "base" argument specified by this `SqlSoup` instance's constructor will be used, which defaults to `object`.
- **mapper_args** – Dictionary of arguments which will be passed directly to `orm.mapper()`.

map_to (*attrname*, *tablename=None*, *selectable=None*, *schema=None*, *base=None*, *mapper_args=frozendict({})*)
Configure a mapping to the given attrname.

This is the "master" method that can be used to create any configuration.

(new in 0.6.6)

Parameters

- **attrname** – String attribute name which will be established as an attribute on this :class:.'.SqlSoup' instance.
- **base** – a Python class which will be used as the base for the mapped class. If `None`, the "base" argument specified by this `SqlSoup` instance's constructor will be used, which defaults to `object`.
- **mapper_args** – Dictionary of arguments which will be passed directly to `orm.mapper()`.
- **tablename** – String name of a `Table` to be reflected. If a `Table` is already available, use the `selectable` argument. This argument is mutually exclusive versus the `selectable` argument.
- **selectable** – a `Table`, `Join`, or `Select` object which will be mapped. This argument is mutually exclusive versus the `tablename` argument.
- **schema** – String schema name to use if the `tablename` argument is present.

rollback()

Rollback the current transction.

See `Session.rollback()`.

with_labels(*selectable*, *base=None*, ***mapper_args*)

Map a selectable directly, wrapping the selectable in a subquery with labels.

The class and its mapping are not cached and will be discarded once dereferenced (as of 0.6.6).

Parameters

- **selectable** – an `expression.select()` construct.
- **base** – a Python class which will be used as the base for the mapped class. If `None`, the "base" argument specified by this `SqlSoup` instance's constructor will be used, which defaults to `object`.
- **mapper_args** – Dictionary of arguments which will be passed directly to `orm.mapper()`.

2.12 Examples

The SQLAlchemy distribution includes a variety of code examples illustrating a select set of patterns, some typical and some not so typical. All are runnable and can be found in the `/examples` directory of the distribution. Each example contains a README in its `__init__.py` file, each of which are listed below.

Additional SQLAlchemy examples, some user contributed, are available on the wiki at http://www.sqlalchemy.org/trac/wiki/UsageRecipes.

2.12.1 Adjacency List

Location: /examples/adjacency_list/ An example of a dictionary-of-dictionaries structure mapped using an adjacency list model.

E.g.:

```
node = TreeNode('rootnode')
node.append('node1')
node.append('node3')
session.add(node)
session.commit()

dump_tree(node)
```

2.12.2 Associations

Location: /examples/association/ Examples illustrating the usage of the "association object" pattern, where an intermediary object associates two endpoint objects together.

The first example illustrates a basic association from a User object to a collection or Order objects, each which references a collection of Item objects.

The second example builds upon the first to add the Association Proxy extension.

E.g.:

```
# create an order
order = Order('john smith')

# append an OrderItem association via the "itemassociations"
# collection with a custom price.
order.itemassociations.append(OrderItem(item('MySQL Crowbar'), 10.99))

# append two more Items via the transparent "items" proxy, which
# will create OrderItems automatically using the default price.
order.items.append(item('SA Mug'))
order.items.append(item('SA Hat'))
```

2.12.3 Attribute Instrumentation

Location: /examples/custom_attributes/ Two examples illustrating modifications to SQLAlchemy's attribute management system.

`listen_for_events.py` illustrates the usage of `AttributeExtension` to intercept attribute events. It additionally illustrates a way to automatically attach these listeners to all class attributes using a `InstrumentationManager`.

`custom_management.py` illustrates much deeper usage of `InstrumentationManager` as well as collection adaptation, to completely change the underlying method used to store state on an object. This example was developed to illustrate techniques which would be used by other third party object instrumentation systems to interact with SQLAlchemy's event system and is only intended for very intricate framework integrations.

2.12.4 Beaker Caching

Location: /examples/beaker_caching/ Illustrates how to embed Beaker cache functionality within the Query object, allowing full cache control as well as the ability to pull "lazy loaded" attributes from long term cache as well.

In this demo, the following techniques are illustrated:

- Using custom subclasses of Query

- Basic technique of circumventing Query to pull from a custom cache source instead of the database.
- Rudimental caching with Beaker, using "regions" which allow global control over a fixed set of configurations.
- Using custom MapperOption objects to configure options on a Query, including the ability to invoke the options deep within an object graph when lazy loads occur.

E.g.:

```
# query for Person objects, specifying cache
q = Session.query(Person).options(FromCache("default", "all_people"))

# specify that each Person's "addresses" collection comes from
# cache too
q = q.options(RelationshipCache("default", "by_person", Person.addresses))

# query
print q.all()
```

To run, both SQLAlchemy and Beaker (1.4 or greater) must be installed or on the current PYTHONPATH. The demo will create a local directory for datafiles, insert initial data, and run. Running the demo a second time will utilize the cache files already present, and exactly one SQL statement against two tables will be emitted - the displayed result however will utilize dozens of lazyloads that all pull from cache.

The demo scripts themselves, in order of complexity, are run as follows:

```
python examples/beaker_caching/helloworld.py

python examples/beaker_caching/relationship_caching.py

python examples/beaker_caching/advanced.py

python examples/beaker_caching/local_session_caching.py
```

Listing of files:

environment.py - Establish the Session, the Beaker cache manager, data / cache file paths, and configurations, bootstrap fixture data if necessary.

caching_query.py - Represent functions and classes which allow the usage of Beaker caching with SQLAlchemy. Introduces a query option called FromCache.

model.py - The datamodel, which represents Person that has multiple Address objects, each with PostalCode, City, Country

fixture_data.py - creates demo PostalCode, Address, Person objects in the database.

helloworld.py - the basic idea.

relationship_caching.py - Illustrates how to add cache options on relationship endpoints, so that lazyloads load from cache.

advanced.py - Further examples of how to use FromCache. Combines techniques from the first two scripts.

local_session_caching.py - Grok everything so far ? This example creates a new Beaker container that will persist data in a dictionary which is local to the current session. remove() the session and the cache is gone.

2.12.5 Derived Attributes

Location: /examples/derived_attributes/ Illustrates a clever technique using Python descriptors to create custom attributes representing SQL expressions when used at the class level, and Python expressions when used at the instance level. In some cases this technique replaces the need to configure the attribute in the mapping, instead relying upon ordinary Python behavior to create custom expression components.

E.g.:

```
class BaseInterval(object):
    @hybrid
    def contains(self,point):
        return (self.start <= point) & (point < self.end)
```

2.12.6 Directed Graphs

Location: /examples/graphs/ An example of persistence for a directed graph structure. The graph is stored as a collection of edges, each referencing both a "lower" and an "upper" node in a table of nodes. Basic persistence and querying for lower- and upper- neighbors are illustrated:

```
n2 = Node(2)
n5 = Node(5)
n2.add_neighbor(n5)
print n2.higher_neighbors()
```

2.12.7 Dynamic Relations as Dictionaries

Location: /examples/dynamic_dict/ Illustrates how to place a dictionary-like facade on top of a "dynamic" relation, so that dictionary operations (assuming simple string keys) can operate upon a large collection without loading the full collection at once.

2.12.8 Horizontal Sharding

Location: /examples/sharding a basic example of using the SQLAlchemy Sharding API. Sharding refers to horizontally scaling data across multiple databases.

The basic components of a "sharded" mapping are:

- multiple databases, each assigned a 'shard id'
- a function which can return a single shard id, given an instance to be saved; this is called "shard_chooser"
- a function which can return a list of shard ids which apply to a particular instance identifier; this is called "id_chooser". If it returns all shard ids, all shards will be searched.
- a function which can return a list of shard ids to try, given a particular Query ("query_chooser"). If it returns all shard ids, all shards will be queried and the results joined together.

In this example, four sqlite databases will store information about weather data on a database-per-continent basis. We provide example shard_chooser, id_chooser and query_chooser functions. The query_chooser illustrates inspection of the SQL expression element in order to attempt to determine a single shard being requested.

2.12.9 Inheritance Mappings

Location: /examples/inheritance/ Working examples of single-table, joined-table, and concrete-table inheritance as described in *datamapping_inheritance*.

2.12.10 Large Collections

Location: /examples/large_collection/ Large collection example.

Illustrates the options to use with `relationship()` when the list of related objects is very large, including:

- "dynamic" relationships which query slices of data as accessed
- how to use ON DELETE CASCADE in conjunction with `passive_deletes=True` to greatly improve the performance of related collection deletion.

2.12.11 Nested Sets

Location: /examples/nested_sets/ Illustrates a rudimentary way to implement the "nested sets" pattern for hierarchical data using the SQLAlchemy ORM.

2.12.12 Polymorphic Associations

Location: /examples/poly_assoc/ Illustrates polymorphic associations, a method of associating a particular child object with many different types of parent object.

This example is based off the original blog post at http://techspot.zzzeek.org/?p=13 and illustrates three techniques:

- `poly_assoc.py` - imitates the non-foreign-key schema used by Ruby on Rails' Active Record.
- `poly_assoc_fk.py` - Adds a polymorphic association table so that referential integrity can be maintained.
- `poly_assoc_generic.py` - further automates the approach of `poly_assoc_fk.py` to also generate the association table definitions automatically.

2.12.13 PostGIS Integration

Location: /examples/postgis A naive example illustrating techniques to help embed PostGIS functionality.

This example was originally developed in the hopes that it would be extrapolated into a comprehensive PostGIS integration layer. We are pleased to announce that this has come to fruition as GeoAlchemy.

The example illustrates:

- a DDL extension which allows CREATE/DROP to work in conjunction with AddGeometryColumn/DropGeometryColumn
- a Geometry type, as well as a few subtypes, which convert result row values to a GIS-aware object, and also integrates with the DDL extension.
- a GIS-aware object which stores a raw geometry value and provides a factory for functions such as AsText().
- an ORM comparator which can override standard column methods on mapped objects to produce GIS operators.
- an attribute event listener that intercepts strings and converts to GeomFromText().
- a standalone operator example.

The implementation is limited to only public, well known and simple to use extension points.

E.g.:

```
print session.query(Road).filter(Road.road_geom.intersects(r1.road_geom)).all()
```

2.12.14 Versioned Objects

Location: /examples/versioning Illustrates an extension which creates version tables for entities and stores records for each change. The same idea as Elixir's versioned extension, but more efficient (uses attribute API to get history) and handles class inheritance. The given extensions generate an anonymous "history" class which represents historical versions of the target object.

Usage is illustrated via a unit test module `test_versioning.py`, which can be run via nose:

```
nosetests -w examples/versioning/
```

A fragment of example usage, using declarative:

```
from history_meta import VersionedMeta, VersionedListener

Base = declarative_base(metaclass=VersionedMeta, bind=engine)
Session = sessionmaker(extension=VersionedListener())

class SomeClass(Base):
    __tablename__ = 'sometable'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    def __eq__(self, other):
        assert type(other) is SomeClass and other.id == self.id

sess = Session()
sc = SomeClass(name='sc1')
sess.add(sc)
sess.commit()

sc.name = 'sc1modified'
sess.commit()

assert sc.version == 2

SomeClassHistory = SomeClass.__history_mapper__.class_

assert sess.query(SomeClassHistory).\
            filter(SomeClassHistory.version == 1).\
            all() \
            == [SomeClassHistory(version=1, name='sc1')]
```

To apply `VersionedMeta` to a subset of classes (probably more typical), the metaclass can be applied on a per-class basis:

```
from history_meta import VersionedMeta, VersionedListener

Base = declarative_base(bind=engine)
```

```
class SomeClass(Base):
    __tablename__ = 'sometable'

    # ...

class SomeVersionedClass(Base):
    __metaclass__ = VersionedMeta
    __tablename__ = 'someothertable'

    # ...
```

The `VersionedMeta` is a declarative metaclass - to use the extension with plain mappers, the `_history_mapper` function can be applied:

```
from history_meta import _history_mapper

m = mapper(SomeClass, sometable)
_history_mapper(m)

SomeHistoryClass = SomeClass.__history_mapper__.class_
```

2.12.15 Vertical Attribute Mapping

Location: /examples/vertical Illustrates "vertical table" mappings.

A "vertical table" refers to a technique where individual attributes of an object are stored as distinct rows in a table. The "vertical table" technique is used to persist objects which can have a varied set of attributes, at the expense of simple query control and brevity. It is commonly found in content/document management systems in order to represent user-created structures flexibly.

Two variants on the approach are given. In the second, each row references a "datatype" which contains information about the type of information stored in the attribute, such as integer, string, or date.

Example:

```
shrew = Animal(u'shrew')
shrew[u'cuteness'] = 5
shrew[u'weasel-like'] = False
shrew[u'poisonous'] = True

session.add(shrew)
session.flush()

q = (session.query(Animal).
     filter(Animal.facts.any(
       and_(AnimalFact.key == u'weasel-like',
            AnimalFact.value == True))))
print 'weasel-like animals', q.all()
```

2.12.16 XML Persistence

Location: /examples/elementtree/ Illustrates three strategies for persisting and querying XML documents as represented by ElementTree in a relational database. The techniques do not apply any mappings to the ElementTree objects directly, so are compatible with the native cElementTree as well as lxml, and can be adapted to suit any kind of DOM representation system. Querying along xpath-like strings is illustrated as well.

In order of complexity:

- `pickle.py` - Quick and dirty, serialize the whole DOM into a BLOB column. While the example is very brief, it has very limited functionality.
- `adjacency_list.py` - Each DOM node is stored in an individual table row, with attributes represented in a separate table. The nodes are associated in a hierarchy using an adjacency list structure. A query function is introduced which can search for nodes along any path with a given structure of attributes, basically a (very narrow) subset of xpath.
- `optimized_al.py` - Uses the same strategy as `adjacency_list.py`, but associates each DOM row with its owning document row, so that a full document of DOM nodes can be loaded using O(1) queries - the construction of the "hierarchy" is performed after the load in a non-recursive fashion and is much more efficient.

E.g.:

```
# parse an XML file and persist in the database
doc = ElementTree.parse("test.xml")
session.add(Document(file, doc))
session.commit()

# locate documents with a certain path/attribute structure
for document in find_document('/somefile/header/field2[@attr=foo]'):
    # dump the XML
    print document
```

CHAPTER

THREE

SQLALCHEMY CORE

3.1 SQL Expression Language Tutorial

3.1.1 Introduction

The SQLAlchemy Expression Language presents a system of representing relational database structures and expressions using Python constructs. These constructs are modeled to resemble those of the underlying database as closely as possible, while providing a modicum of abstraction of the various implementation differences between database backends. While the constructs attempt to represent equivalent concepts between backends with consistent structures, they do not conceal useful concepts that are unique to particular subsets of backends. The Expression Language therefore presents a method of writing backend-neutral SQL expressions, but does not attempt to enforce that expressions are backend-neutral.

The Expression Language is in contrast to the Object Relational Mapper, which is a distinct API that builds on top of the Expression Language. Whereas the ORM, introduced in *Object Relational Tutorial*, presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language, the Expression Language presents a system of representing the primitive constructs of the relational database directly without opinion.

While there is overlap among the usage patterns of the ORM and the Expression Language, the similarities are more superficial than they may at first appear. One approaches the structure and content of data from the perspective of a user-defined domain model which is transparently persisted and refreshed from its underlying storage model. The other approaches it from the perspective of literal schema and SQL expression representations which are explicitly composed into messages consumed individually by the database.

A successful application may be constructed using the Expression Language exclusively, though the application will need to define its own system of translating application concepts into individual database messages and from individual database result sets. Alternatively, an application constructed with the ORM may, in advanced scenarios, make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

The following 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.

3.1.2 Version Check

A quick check to verify that we are on at least **version 0.6** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
0.6.0
```

3.1.3 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.

3.1.4 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 *Schema Definition Language*.

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
```

Note: Users familiar with the syntax of CREATE TABLE may notice that the VARCHAR columns were generated without a length; on SQLite and Postgresql, this is a valid datatype, but on others, it's not allowed. So if running this tutorial on one of those databases, and you wish to use SQLAlchemy to issue CREATE TABLE, 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.

Additionally, Firebird and Oracle require sequences to generate new primary key identifiers, and SQLAlchemy doesn't generate or assume these without being instructed. For that, you use the `Sequence` construct:

```
from sqlalchemy import Sequence
Column('id', Integer, Sequence('user_id_seq'), primary_key=True)
```

A full, foolproof `Table` is therefore:

```
users = Table('users', metadata,
   Column('id', Integer, Sequence('user_id_seq'), primary_key=True),
   Column('name', String(50)),
   Column('fullname', String(50)),
   Column('password', String(12))
)
```

3.1.5 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'}
```

3.1.6 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.inserted_primary_key
[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 database's `Dialect` knows the specific steps needed to determine the correct value (or values; note that `inserted_primary_key` returns a list so that it supports composite primary keys).

3.1.7 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.

3.1.8 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 checks out from the connection pool a connection for you, performs the execute operation with that connection, and then checks the connection back into the pool upon completion of the operation:

```
>>> 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: *Connectionless Execution, Implicit Execution*.

3.1.9 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 cursor and connection resources referenced by the `ResultProxy` will be respectively closed and returned to the connection pool 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 see 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`.

3.1.10 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
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in *somecolumn*.

3.1.11 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.

3.1.12 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, the *select()* construct accepts strings for most of its arguments. Below we combine the usage of strings with our constructed `select()` object, by using the `select()` object to structure the statement, and 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(["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.

3.1.13 Using Aliases

The alias corresponds to a "renamed" version of a table or arbitrary relationship, 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',)]
```

3.1.14 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.dialects.oracle import dialect as 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 ;).

3.1.15 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.

3.1.16 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.

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
```

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)
```

By "generates", we mean that **any** SQL function is created based on the word you choose:

```
>>> print func.xyz_my_goofy_function()
xyz_my_goofy_function()
```

Certain function names are known by SQLAlchemy, allowing special behavioral rules to be applied. Some for example are "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`.

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')]
```

A common issue with so-called "compound" selectables arises due to the fact that they nest with parenthesis. SQLite in particular doesn't like a statement that starts with parenthesis. So when nesting a "compound" inside a "compound", it's often necessary to apply `.alias().select()` to the first element of the outermost compound, if that element is also a compound. For example, to nest a "union" and a "select" inside of "except_", SQLite will want the "union" to be stated as a subquery:

```
>>> u = except_(
...     union(
...          addresses.select(addresses.c.email_address.like('%@yahoo.com')),
...          addresses.select(addresses.c.email_address.like('%@msn.com'))
...      ).alias().select(),    # apply subquery here
...     addresses.select(addresses.c.email_address.like('%@msn.com'))
... )
>>> print conn.execute(u).fetchall()
SELECT anon_1.id, anon_1.user_id, anon_1.email_address
FROM (SELECT addresses.id AS id, addresses.user_id AS user_id,
addresses.email_address AS email_address FROM addresses
WHERE addresses.email_address LIKE ? UNION SELECT addresses.id AS id,
addresses.user_id AS user_id, addresses.email_address AS email_address
FROM addresses WHERE addresses.email_address LIKE ?) AS anon_1 EXCEPT
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address LIKE ?
('%@yahoo.com', '%@msn.com', '%@msn.com')[(1, 1, u'jack@yahoo.com')]
```

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)]
```

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)
```

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')]
```

3.1.17 Inserts and Updates

Finally, we're back to INSERT for some more detail. The `insert()` construct provides a `values()` method which can be used to send any value or clause expression to the VALUES portion of the INSERT:

```
# insert from a function
users.insert().values(id=12, name=func.upper('jack'))

# insert from a concatenation expression
addresses.insert().values(email_address = name + '@' + host)
```

`values()` can be mixed with per-execution values:

```
conn.execute(
    users.insert().values(name=func.upper('jack')),
    fullname='Jack Jones'
)
```

`bindparam()` constructs can be passed, however the names of the table's columns are reserved for the "automatic" generation of bind names:

```
users.insert().values(id=bindparam('_id'), name=bindaparam('_name'))

# insert many rows at once:
conn.execute(
    users.insert().values(id=bindparam('_id'), name=bindaparam('_name')),
    [
        {'_id':1, '_name':'name1'},
        {'_id':2, '_name':'name2'},
        {'_id':3, '_name':'name3'},
    ]
)
```

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('newname'))
>>> conn.execute(u, oldname='jack', newname='ed')
UPDATE users SET name=? WHERE users.name = ?
('ed', 'jack')
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>

>>> # with binds, you can also update many rows at once
>>> conn.execute(u,
...     {'oldname':'jack', 'newname':'ed'},
...     {'oldname':'wendy', 'newname':'mary'},
...     {'oldname':'jim', 'newname':'jake'},
...     )
UPDATE users SET name=? WHERE users.name = ?
[('ed', 'jack'), ('mary', 'wendy'), ('jake', 'jim')]
COMMIT<sqlalchemy.engine.base.ResultProxy object at 0x...>

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

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(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...>
```

3.1.18 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...>
```

3.1.19 Further Reference

Expression Language Reference: *SQL Statements and Expressions*

Database Metadata Reference: *Schema Definition Language*

Engine Reference: *Engine Configuration*

Connection Reference: *Working with Engines and Connections*

Types Reference: *Column and Data Types*

3.2 SQL Statements and Expressions

This section presents the API reference for the SQL Expression Language. For a full introduction to its usage, see *SQL Expression Language Tutorial*.

3.2.1 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`.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`asc`**`(`*column*`)`
Return an ascending `ORDER BY` clause element.

e.g.:

```
order_by = [asc(table1.mycol)]
```

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`bindparam`**`(`*key*, *value=None*, *type_=None*, *unique=False*, *required=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

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.

required A value is required at execution time.

`sqlalchemy.sql.expression.`**`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 `literal_column()` 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,
    })
```

Using `literal_column()`, to allow for databases that do not support bind parameters in the `then` clause. The type can be specified which determines the type of the `case()` construct overall:

```
case([(orderline.c.qty > 100,
        literal_column("'greaterthan100'", String)),
      (orderline.c.qty > 10, literal_column("'greaterthan10'",
        String))
    ], else_=literal_column("'lethan10'", String))
```

`sqlalchemy.sql.expression.`**`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)
```

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`collate`**`(`*expression*, *collation*`)`
Return the clause `expression COLLATE collation`.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`desc`**`(`*column*`)`
Return a descending `ORDER BY` clause element.

e.g.:

```
order_by = [desc(table1.mycol)]
```

`sqlalchemy.sql.expression.`**`distinct`**`(`*expr*`)`
Return a `DISTINCT` clause.

`sqlalchemy.sql.expression.`**`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()`.

`sqlalchemy.sql.expression.`**`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()`.

`sqlalchemy.sql.expression.`**`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)
```

sqlalchemy.sql.expression.**extract**(*field*, *expr*)
Return the clause `extract(field FROM expr)`.

sqlalchemy.sql.expression.**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*.

sqlalchemy.sql.expression.**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.

`sqlalchemy.sql.expression.`**`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()`.

`sqlalchemy.sql.expression.`**`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()`.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`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`.

`sqlalchemy.sql.expression.`**`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.

Parameters

- **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.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`null`**`()`

Return a `_Null` object, which compiles to `NULL` in a sql statement.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`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.

Parameters

- **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.
- **autocommit** – Deprecated. Use .execution_options(autocommit=<True|False>) to set the autocommit option.
- **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.

`sqlalchemy.sql.expression.`**`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.

`sqlalchemy.sql.expression.`**`table`**`(`*name*, **columns*`)`
Return a `TableClause` object.

This is a primitive version of the `Table` object, which is a subclass of this object.

`sqlalchemy.sql.expression.`**`text`**`(`*text*, *bind=None*, **args*, ***kwargs*`)`
Create a SQL construct that is represented by a literal string.

E.g.:

```
t = text("SELECT * FROM users")
result = connection.execute(t)
```

The advantages `text()` provides over a plain string are backend-neutral support for bind parameters, per-statement execution options, as well as bind parameter and result-column typing behavior, allowing SQLAlchemy type constructs to play a role when executing a statement that is specified literally.

Bind parameters are specified by name, using the format `:name`. E.g.:

```
t = text("SELECT * FROM users WHERE id=:user_id")
result = connection.execute(t, user_id=12)
```

To invoke SQLAlchemy typing logic for bind parameters, the `bindparams` list allows specification of `bindparam()` constructs which specify the type for a given name:

```
t = text("SELECT id FROM users WHERE updated_at>:updated",
            bindparams=[bindparam('updated', DateTime())]
        )
```

Typing during result row processing is also an important concern. Result column types are specified using the `typemap` dictionary, where the keys match the names of columns. These names are taken from what the DBAPI returns as `cursor.description`:

```
t = text("SELECT id, name FROM users",
        typemap={
            'id':Integer,
            'name':Unicode
```

```
        }
)
```

The `text()` construct is used internally for most cases when a literal string is specified for part of a larger query, such as within `select()`, `update()`, `insert()` or `delete()`. In those cases, the same bind parameter syntax is applied:

```
s = select([users.c.id, users.c.name]).where("id=:user_id")
result = connection.execute(s, user_id=12)
```

Using `text()` explicitly usually implies the construction of a full, standalone statement. As such, SQLAlchemy refers to it as an `Executable` object, and it supports the `Executable.execution_options()` method. For example, a `text()` construct that should be subject to "autocommit" can be set explicitly so using the `autocommit` option:

```
t = text("EXEC my_procedural_thing()").\
        execution_options(autocommit=True)
```

Note that SQLAlchemy's usual "autocommit" behavior applies to `text()` constructs - that is, statements which begin with a phrase such as `INSERT`, `UPDATE`, `DELETE`, or a variety of other phrases specific to certain backends, will be eligible for autocommit if no transaction is in progress.

Parameters

- **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.
- **autocommit** – Deprecated. Use .execution_options(autocommit=<True|False>) to set the autocommit option.
- **bind** – an optional connection or engine to be used for this text query.
- **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 columns clause of a `SELECT` statement to type objects, which will be used to perform post-processing on columns within the result set. This argument applies to any expression that returns result sets.

`sqlalchemy.sql.expression.`**`tuple_`**`(*expr)`

Return a SQL tuple.

Main usage is to produce a composite IN construct:

```
tuple_(table.c.col1, table.c.col2).in_(
    [(1, 2), (5, 12), (10, 19)]
)
```

`sqlalchemy.sql.expression.`**`type_coerce`**`(expr, type_)`

Coerce the given expression into the given type, on the Python side only.

`type_coerce()` is roughly similar to :func:.'cast', except no "CAST" expression is rendered - the given type is only applied towards expression typing and against received result values.

e.g.:

```
from sqlalchemy.types import TypeDecorator
import uuid

class AsGuid(TypeDecorator):
    impl = String

    def process_bind_param(self, value, dialect):
        if value is not None:
            return str(value)
        else:
            return None

    def process_result_value(self, value, dialect):
        if value is not None:
            return uuid.UUID(value)
        else:
            return None

conn.execute(
    select([type_coerce(mytable.c.ident, AsGuid)]).\
            where(
                type_coerce(mytable.c.ident, AsGuid) ==
                uuid.uuid3(uuid.NAMESPACE_URL, 'bar')
            )
)
```

`sqlalchemy.sql.expression.`**`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()`.

`sqlalchemy.sql.expression.`**`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()`.

`sqlalchemy.sql.expression.`**`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.

3.2.2 Classes

class `sqlalchemy.sql.expression.`**`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 `sqlalchemy.sql.expression.`**`_BindParamClause`**(*key*, *value*, *type_=None*, *unique=False*, *isoutparam=False*, *required=False*, *_compared_to_operator=None*, *_compared_to_type=None*)

Bases: `sqlalchemy.sql.expression.ColumnElement`

Represent a bind parameter.

Public constructor is the `bindparam()` function.

`__init__`(*key*, *value*, *type_=None*, *unique=False*, *isoutparam=False*, *required=False*, *_compared_to_operator=None*, *_compared_to_type=None*)

Construct a _BindParamClause.

Parameters

- **key** – the key for this bind param. Will be used in the generated SQL statement for dialects that use named parameters. This value may be modified when part of a compilation operation, if other `_BindParamClause` objects exist with the same key, or if its length is too long and truncation is required.
- **value** – Initial value for this bind param. This value may be overridden by the dictionary of parameters sent to statement compilation/execution.

- **type_** – A `TypeEngine` object that will be used to pre-process the value corresponding to this `_BindParamClause` at execution time.
- **unique** – if True, the key name of this BindParamClause will be modified if another `_BindParamClause` of the same name already has been located within the containing `ClauseElement`.
- **required** – a value is required at execution time.
- **isoutparam** – if True, the parameter should be treated like a stored procedure "OUT" parameter.

compare(*other*, ***kw*)
Compare this `_BindParamClause` to the given clause.

class `sqlalchemy.sql.expression.`**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*, ***kw*)
Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)
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.
- **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`. Deprecated since version 0.7: (pending) Only SQL expressions which subclass `Executable` may provide the `execute()` method.

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 Deprecated since version 0.7: (pending) Only SQL expressions which subclass `Executable` may provide the `scalar()` method. the result's scalar representation.

self_group (*against=None*)
Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

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 `sqlalchemy.sql.expression.`**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 `sqlalchemy.sql.expression.`**`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 `sqlalchemy.sql.expression.`**`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.

`anon_label`

provides a constant 'anonymous label' for this ColumnElement.

This is a label() expression which will be named at compile time. The same label() is returned each time anon_label is called so that expressions can reference anon_label multiple times, producing the same label name at compile time.

the compiler uses this function automatically at compile time for expressions that are known to be 'unnamed' like binary expressions and function calls.

`compare`(*other*, *use_proxies=False*, *equivalents=None*, ***kw*)

Compare this ColumnElement to another.

Special arguments understood:

Parameters

- **use_proxies** – when True, consider two columns that share a common base column as equivalent (i.e. shares_lineage())
- **equivalents** – a dictionary of columns as keys mapped to sets of columns. If the given "other" column is present in this dictionary, if any of the columns in the correponding set() pass the comparison test, the result is True. This is used to expand the comparison to other columns that may be known to be equivalent to this one via foreign key or other criterion.

shares_lineage(*othercolumn*)

Return True if the given `ColumnElement` has a common ancestor to this `ColumnElement`.

class `sqlalchemy.sql.expression.`**_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
```

Parameters

- **operator** – a string which will be output as the infix operator between this `ClauseElement` and the expression passed to the generated function.

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in somecolumn.

operate(*op*, **other*, ***kwargs*)

reverse_operate(*op*, *other*, ***kwargs*)

startswith(*other*, *escape=None*)
Produce the clause `LIKE '<other>%'`

class `sqlalchemy.sql.expression.`**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 `sqlalchemy.sql.expression.`**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 `sqlalchemy.sql.expression.`**`Delete`**`(`*table*, *whereclause*, *bind=None*, *returning=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 `sqlalchemy.sql.expression.`**`Executable`**

Bases: `sqlalchemy.sql.expression._Generative`

Mark a ClauseElement as supporting execution.

`Executable` is a superclass for all "statement" types of objects, including `select()`, `delete()`, `update()`, `insert()`, `text()`.

`execute``(`**multiparams*, ***params*`)`

Compile and execute this `Executable`.

`execution_options``(`***kw*`)`

Set non-SQL options for the statement which take effect during execution.

Current options include:

- autocommit - when True, a COMMIT will be invoked after execution when executed in 'autocommit' mode, i.e. when an explicit transaction is not begun on the connection. Note that DBAPI connections by default are always in a transaction - SQLAlchemy uses rules applied to different kinds of statements to determine if COMMIT will be invoked in order to provide its "autocommit" feature. Typically, all INSERT/UPDATE/DELETE statements as well as CREATE/DROP statements have autocommit behavior enabled; SELECT constructs do not. Use this option when invoking a SELECT or other specific SQL construct where COMMIT is desired (typically when calling stored procedures and such).

- stream_results - indicate to the dialect that results should be "streamed" and not pre-buffered, if possible. This is a limitation of many DBAPIs. The flag is currently understood only by the psycopg2 dialect.

- compiled_cache - a dictionary where `Compiled` objects will be cached when the `Connection` compiles a clause expression into a dialect- and parameter-specific `Compiled` object. It is the user's responsibility to manage the size of this dictionary, which will have keys corresponding to the dialect, clause element, the column names within the VALUES or SET clause of an INSERT or UPDATE, as well as the "batch" mode for an INSERT or UPDATE statement. The format of this dictionary is not guaranteed to stay the same in future releases.

  This option is usually more appropriate to use via the `sqlalchemy.engine.base.Connection.execution_c` method of `Connection`, rather than upon individual statement objects, though the effect is the same.

See also:

`sqlalchemy.engine.base.Connection.execution_options()`

`sqlalchemy.orm.query.Query.execution_options()`

`scalar``(`**multiparams*, ***params*`)`

Compile and execute this `Executable`, returning the result's scalar representation.

class `sqlalchemy.sql.expression.`**`FunctionElement`**`(`**clauses*, ***kwargs*`)`

Bases: `sqlalchemy.sql.expression.Executable`, `sqlalchemy.sql.expression.ColumnElement`, `sqlalchemy.sql.expression.FromClause`

Base for SQL function-oriented constructs.

__init__(**clauses*, ***kwargs*)

class sqlalchemy.sql.expression.**Function**(*name*, **clauses*, ***kw*)
Bases: `sqlalchemy.sql.expression.FunctionElement`

Describe a named SQL function.

__init__(*name*, **clauses*, ***kw*)

class sqlalchemy.sql.expression.**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 `sqlalchemy.sql.expression.`**Insert** (*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=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 `sqlalchemy.sql.expression.`**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.
- ****kwargs** – all other kwargs are sent to the underlying `select()` function.

class `sqlalchemy.sql.expression.`**`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.

locate_all_froms
return a Set of all FromClause elements referenced by this Select.

> This set is a superset of that returned by the `froms` property,

which is specifically for those FromClause elements that would actually be rendered.

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_hint (*selectable*, *text*, *dialect_name='*'*)
Add an indexing hint for the given selectable to this `Select`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the given `Table` or `Alias` passed as the *selectable* argument. The dialect implementation typically uses Python string substitution syntax with the token `%(name)s` to render the name of the table or alias. E.g. when using Oracle, the following:

```
select([mytable]).\
    with_hint(mytable, "+ index(%(name)s ix_mytable)")
```

Would render SQL as:

```
select /*+ index(mytable ix_mytable) */ ... from mytable
```

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add hints for both Oracle and Sybase simultaneously:

```
select([mytable]).\
    with_hint(mytable, "+ index(%(name)s ix_mytable)", 'oracle').\
    with_hint(mytable, "WITH INDEX ix_mytable", 'sybase')
```

with_only_columns(*columns*)
return a new select() construct with its columns clause replaced with the given columns.

class `sqlalchemy.sql.expression.`**Selectable**
Bases: `sqlalchemy.sql.expression.ClauseElement`

mark a class as being selectable

class `sqlalchemy.sql.expression.`**_SelectBaseMixin**(*use_labels=False*, *for_update=False*, *limit=None*, *offset=None*, *order_by=None*, *group_by=None*, *bind=None*, *autocommit=None*)
Bases: `sqlalchemy.sql.expression.Executable`

Base class for `Select` and `CompoundSelects`.

__init__(*use_labels=False*, *for_update=False*, *limit=None*, *offset=None*, *order_by=None*, *group_by=None*, *bind=None*, *autocommit=None*)

append_group_by(**clauses*)
Append the given GROUP BY criterion applied to this selectable.

The criterion will be appended to any pre-existing GROUP BY criterion.

append_order_by(**clauses*)
Append the given ORDER BY criterion applied to this selectable.

The criterion will be appended to any pre-existing ORDER BY criterion.

apply_labels()
return a new selectable with the 'use_labels' flag set to True.

This will result in column expressions being generated using labels against their table name, such as "SELECT somecolumn AS tablename_somecolumn". This allows selectables which contain multiple FROM clauses to produce a unique set of column names regardless of name conflicts among the individual FROM clauses.

as_scalar()
return a 'scalar' representation of this selectable, which can be used as a column expression.

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

The returned object is an instance of `_ScalarSelect`.

autocommit()
return a new selectable with the 'autocommit' flag set to Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag. True.

group_by(**clauses*)
return a new selectable with the given list of GROUP BY criterion applied.

The criterion will be appended to any pre-existing GROUP BY criterion.

label (*name*)
return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also `as_scalar()`.

limit (*limit*)
return a new selectable with the given LIMIT criterion applied.

offset (*offset*)
return a new selectable with the given OFFSET criterion applied.

order_by (**clauses*)
return a new selectable with the given list of ORDER BY criterion applied.

The criterion will be appended to any pre-existing ORDER BY criterion.

class `sqlalchemy.sql.expression.`**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*)

count (*whereclause=None*, ***params*)
return a SELECT COUNT generated against this `TableClause`.

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 `sqlalchemy.sql.expression.`**Update** (*table*, *whereclause*, *values=None*, *inline=False*, *bind=None*, *returning=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.

3.2.3 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)
```

Note that any name not known to `func` generates the function name as is - there is no restriction on what SQL functions can be called, known or unknown to SQLAlchemy, built-in or user defined. The section here only describes those functions where SQLAlchemy already knows what argument and return types are in use.

class `sqlalchemy.sql.functions.`**`AnsiFunction`**(***kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

`__init__`(***kwargs*)

class `sqlalchemy.sql.functions.`**`GenericFunction`**(*type_=None*, *args=()*, ***kwargs*)
Bases: `sqlalchemy.sql.expression.Function`

`__init__`(*type_=None*, *args=()*, ***kwargs*)

class `sqlalchemy.sql.functions.`**`ReturnTypeFromArgs`**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

Define a function whose return type is the same as its arguments.

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.sql.functions.`**`char_length`**(*arg*, ***kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

`__init__`(*arg*, ***kwargs*)

class `sqlalchemy.sql.functions.`**`coalesce`**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

class `sqlalchemy.sql.functions.`**`concat`**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.sql.functions.`**`count`**(*expression=None*, ***kwargs*)
Bases: `sqlalchemy.sql.functions.GenericFunction`

The ANSI COUNT aggregate function. With no arguments, emits COUNT *.

`__init__`(*expression=None*, ***kwargs*)

class `sqlalchemy.sql.functions.`**`current_date`**(***kwargs*)
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`current_time`**(***kwargs*)
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`current_timestamp`**(***kwargs*)
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`current_user`**(***kwargs*)
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`localtime`**(***kwargs*)
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`localtimestamp`**(***kwargs*)
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`max`**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

class `sqlalchemy.sql.functions.`**`min`**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

class `sqlalchemy.sql.functions.`**`now`**`(`*type_=None, args=(), **kwargs*`)`
Bases: `sqlalchemy.sql.functions.GenericFunction`

class `sqlalchemy.sql.functions.`**`random`**`(`**args, **kwargs*`)`
Bases: `sqlalchemy.sql.functions.GenericFunction`

`__init__``(`**args, **kwargs*`)`

class `sqlalchemy.sql.functions.`**`session_user`**`(`***kwargs*`)`
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`sum`**`(`**args, **kwargs*`)`
Bases: `sqlalchemy.sql.functions.ReturnTypeFromArgs`

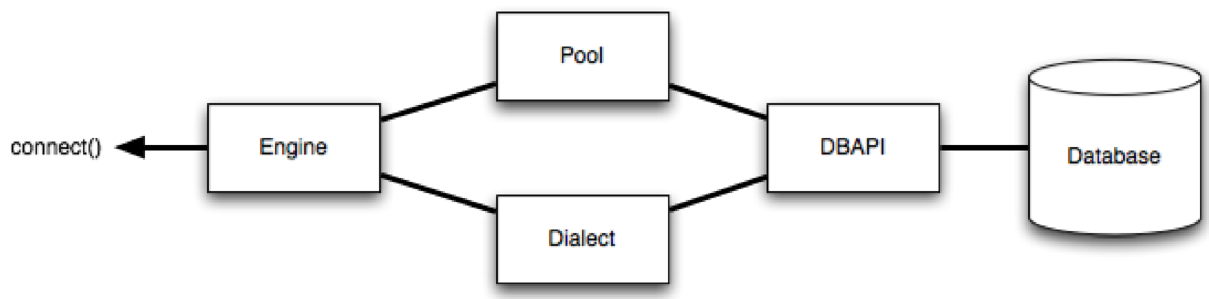
class `sqlalchemy.sql.functions.`**`sysdate`**`(`***kwargs*`)`
Bases: `sqlalchemy.sql.functions.AnsiFunction`

class `sqlalchemy.sql.functions.`**`user`**`(`***kwargs*`)`
Bases: `sqlalchemy.sql.functions.AnsiFunction`

3.3 Engine Configuration

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 can be illustrated as follows:



Where above, an `Engine` references both a `Dialect` and a `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()`:

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')
```

The above engine invokes the `postgresql` dialect and a connection pool which references `localhost:5432`.

The `Engine`, once created, can either be used directly to interact with the database, or can be passed to a `Session` object to work with the ORM. This section covers the details of configuring an `Engine`. The next section, *Working with Engines and Connections*, will detail the usage API of the `Engine` and similar, typically for non-ORM applications.

3.3.1 Supported Databases

SQLAlchemy includes many `Dialect` implementations for various backends; each is described as its own package in the *sqlalchemy.dialects_toplevel* package. A SQLAlchemy dialect always requires that an appropriate DBAPI driver is installed.

The table below summarizes the state of DBAPI support in SQLAlchemy 0.6. The values translate as:

- yes / Python platform - The SQLAlchemy dialect is mostly or fully operational on the target platform.
- yes / OS platform - The DBAPI supports that platform.
- no / Python platform - The DBAPI does not support that platform, or there is no SQLAlchemy dialect support.
- no / OS platform - The DBAPI does not support that platform.
- partial - the DBAPI is partially usable on the target platform but has major unresolved issues.
- development - a development version of the dialect exists, but is not yet usable.
- thirdparty - the dialect itself is maintained by a third party, who should be consulted for information on current support.
- * - indicates the given DBAPI is the "default" for SQLAlchemy, i.e. when just the database name is specified

Driver	Connect string	Py2K	Py3K	Jython	Unix
DB2/Informix IDS					
ibm-db	thirdparty	thirdparty	thirdparty	thirdparty	thirdparty
Firebird					
kinterbasdb	`firebird+kinterbasdb*`	yes	development	no	yes
Informix					
informixdb	`informix+informixdb*`	yes	development	no	unknown
MaxDB					
sapdb	`maxdb+sapdb*`	development	development	no	yes
Microsoft Access					
pyodbc	`access+pyodbc*`	development	development	no	unknown
Microsoft SQL Server					
adodbapi	`mssql+adodbapi`	development	development	no	no
jTDS JDBC Driver	`mssql+zxjdbc`	no	no	development	yes
mxodbc	`mssql+mxodbc`	yes	development	no	yes with FreeTDS
pyodbc	`mssql+pyodbc*`	yes	development	no	yes with FreeTDS
pymssql	`mssql+pymssql`	yes	development	no	yes
MySQL					
MySQL Connector/J	`mysql+zxjdbc`	no	no	yes	yes
MySQL Connector/Python	`mysql+mysqlconnector`	yes	yes	no	yes
mysql-python	`mysql+mysqldb*`	yes	development	no	yes
OurSQL	`mysql+oursql`	yes	yes	no	yes
Oracle					
cx_oracle	`oracle+cx_oracle*`	yes	development	no	yes
Oracle JDBC Driver	`oracle+zxjdbc`	no	no	yes	yes
Postgresql					
pg8000	`postgresql+pg8000`	yes	yes	no	yes
PostgreSQL JDBC Driver	`postgresql+zxjdbc`	no	no	yes	yes
psycopg2	`postgresql+psycopg2*`	yes	development	no	yes
pypostgresql	`postgresql+pypostgresql`	no	yes	no	yes
SQLite					
pysqlite	`sqlite+pysqlite*`	yes	yes	no	yes
sqlite3	`sqlite+pysqlite*`	yes	yes	no	yes
Sybase ASE					
mxodbc	`sybase+mxodbc`	development	development	no	yes
pyodbc	`sybase+pyodbc*`	partial	development	no	unknown

Continued on next page

Table 3.1 – continued from previous page

python-sybase	sybase+pysybase	yes [1]	development	no	yes

Further detail on dialects is available at *Dialects* as well as additional notes on the wiki at Database Notes

3.3.2 Database Engine Options

Keyword options can also be specified to `create_engine()`, following the string URL as follows:

```
db = create_engine('postgresql://...', encoding='latin1', echo=True)
```

`sqlalchemy.`**`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+driver://user:password@host/dbname[?key=value..]`, where `dialect` is a database name such as `mysql`, `oracle`, `postgresql`, etc., and `driver` the name of a DBAPI, such as `psycopg2`, `pyodbc`, `cx_oracle`, 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** – Deprecated. A warning is raised in all cases when a non-Unicode object is passed when SQLAlchemy would coerce into an encoding (note: but **not** when the DBAPI handles unicode objects natively). To suppress or raise this warning to an error, use the Python warnings filter documented at: http://docs.python.org/library/warnings.html
- **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 Python Unicode values to raw byte values sent to the DBAPI as bind parameters, 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 those DBAPIs that do not accept Python Unicode objects as input. For Unicode conversion on a column-by-column level, use the `Unicode` column type instead, described in *Column and Data Types*. Note that many DBAPIs have the ability to return Python Unicode objects in result sets directly - SQLAlchemy will use these modes of operation if possible and will also attempt to detect "Unicode returns" behavior by the DBAPI upon first connect by the `Engine`. When this is detected, string values in result sets are passed through without further processing.
- **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

[1] The Sybase dialect currently lacks the ability to reflect tables.

will be printed to the standard output as well. This flag ultimately controls a Python logger; see *Configuring Logging* 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 *Configuring 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.
- **execution_options** – Dictionary execution options which will be applied to all connections. See `execution_options()`
- **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.
- **listeners** – A list of one or more `PoolListener` objects which will receive connection pool events.
- **logging_name** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.engine" logger. Defaults to a hexstring of the object's id.
- **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`.
- **module=None** – reference to a Python module object (the module itself, not its string name). Specifies an alternate DBAPI module to be used by the engine's dialect. Each sub-dialect references a specific DBAPI which will be imported before first connect. This parameter causes the import to be bypassed, and the given module to be used instead. Can be used for testing of DBAPIs as well as to inject "mock" DBAPI implementations into the `Engine`.
- **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 *Connection 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.
- **pool_logging_name** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.pool" logger. Defaults to a hexstring of the object's id.
- **pool_size=5** – the number of connections to keep open inside the connection pool. This used with `QueuePool` as well as `SingletonThreadPool`. With `QueuePool`, a `pool_size` setting of 0 indicates no limit; to disable pooling, set `poolclass` to `NullPool` instead.
- **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'** – selects alternate engine implementations. Currently available is the `threadlocal` strategy, which is described in *Using the Threadlocal Execution Strategy*.

`sqlalchemy.`**`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.

3.3.3 Database Urls

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:

`dialect+driver://username:password@host:port/database`

Dialect names include the identifying name of the SQLAlchemy dialect which include `sqlite`, `mysql`, `postgresql`, `oracle`, `mssql`, and `firebird`. The drivername is the name of the DBAPI to be used to connect to the database using all lowercase letters. If not specified, a "default" DBAPI will be imported if available - this default is typically the most widely known driver available for that backend (i.e. cx_oracle, pysqlite/sqlite3, psycopg2, mysqldb). For Jython connections, specify the *zxjdbc* driver, which is the JDBC-DBAPI bridge included with Jython.

```
# postgresql - psycopg2 is the default driver.
pg_db = create_engine('postgresql://scott:tiger@localhost/mydatabase')
pg_db = create_engine('postgresql+psycopg2://scott:tiger@localhost/mydatabase')
pg_db = create_engine('postgresql+pg8000://scott:tiger@localhost/mydatabase')
pg_db = create_engine('postgresql+pypostgresql://scott:tiger@localhost/mydatabase')

# postgresql on Jython
pg_db = create_engine('postgresql+zxjdbc://scott:tiger@localhost/mydatabase')

# mysql - MySQLdb (mysql-python) is the default driver
mysql_db = create_engine('mysql://scott:tiger@localhost/foo')
mysql_db = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

# mysql on Jython
mysql_db = create_engine('mysql+zxjdbc://localhost/foo')

# mysql with pyodbc (buggy)
mysql_db = create_engine('mysql+pyodbc://scott:tiger@some_dsn')

# oracle - cx_oracle is the default driver
oracle_db = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

# oracle via TNS name
oracle_db = create_engine('oracle+cx_oracle://scott:tiger@tnsname')

# mssql using ODBC datasource names.  PyODBC is the default driver.
mssql_db = create_engine('mssql://mydsn')
mssql_db = create_engine('mssql+pyodbc://mydsn')
mssql_db = create_engine('mssql+adodbapi://mydsn')
mssql_db = create_engine('mssql+pyodbc://username:password@mydsn')
```

SQLite connects to file based databases. The same URL format is used, omitting the hostname, and using the "file" portion as the filename of the database. This has the effect of four slashes being present for an absolute file path:

```
# sqlite://<nohostname>/<path>
# where <path> is relative:
sqlite_db = create_engine('sqlite:///foo.db')

# or absolute, starting with a slash:
sqlite_db = create_engine('sqlite:////absolute/path/to/foo.db')
```

To use a SQLite `:memory:` database, specify an empty URL:

```
sqlite_memory_db = create_engine('sqlite://')
```

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.

class `sqlalchemy.engine.url.`**`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.

3.3.4 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('postgresql://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('postgresql://scott:tiger@localhost/test', connect_args = {'argument1':
```

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('postgresql://', creator=connect)
```

3.3.5 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.dialects` - controls custom logging for SQL dialects. See the documentation of individual dialects for details.
- `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 information on mapper configurations.

For example, to log SQL queries using Python logging instead of the `echo=True` flag:

```
import logging

logging.basicConfig()
logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)
```

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.

The logger name of instance such as an `Engine` or `Pool` defaults to using a truncated hex identifier string. To set this to a specific name, use the "logging_name" and "pool_logging_name" keyword arguments with `sqlalchemy.create_engine()`.

3.4 Working with Engines and Connections

This section details direct usage of the `Engine`, `Connection`, and related objects. Its important to note that when using the SQLAlchemy ORM, these objects are not generally accessed; instead, the `Session` object is used as the interface to the database. However, for applications that are built around direct usage of textual SQL statements and/or SQL expression constructs without involvement by the ORM's higher level management services, the `Engine` and `Connection` are king (and queen?) - read on.

3.4.1 Basic Usage

Recall from *Engine Configuration* that an `Engine` is created via the `create_engine()` call:

```
engine = create_engine('mysql://scott:tiger@localhost/test')
```

The typical usage of `create_engine()` is once per particular database URL, held globally for the lifetime of a single application process. A single `Engine` manages many individual DBAPI connections on behalf of the process and is intended to be called upon in a concurrent fashion. The `Engine` is **not** synonymous to the DBAPI `connect` function, which represents just one connection resource - the `Engine` is most efficient when created just once at the module level of an application, not per-object or per-function call.

For a multiple-process application that uses the `os.fork` system call, or for example the Python `multiprocessing` module, it's usually required that a separate `Engine` be used for each child process. This is because the `Engine` maintains a reference to a connection pool that ultimately references DBAPI connections - these tend to not be portable across process boundaries. An `Engine` that is configured not to use pooling (which is achieved via the usage of `NullPool`) does not have this requirement.

The engine can be used directly to issue SQL to the database. The most generic way is first procure a connection resource, 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 DBAPI connection is retrieved from the connection pool at the point at which `Connection` is created.

The returned result is an instance of `ResultProxy`, which references a DBAPI cursor and provides a largely compatible interface with that of the DBAPI cursor. The DBAPI cursor will be closed by the `ResultProxy` when all of its result rows (if any) are exhausted. A ResultProxy that returns no rows, such as that of an UPDATE statement (without any returned rows), releases cursor resources immediately upon construction.

When the close() method is called, the referenced DBAPI connection is returned to the connection pool. From the perspective of the database itself, nothing is actually "closed", assuming pooling is in use. The pooling mechanism issues a rollback() call on the DBAPI connection so that any transactional state or locks are removed, and the connection is ready for its next usage.

The above procedure can be performed in a shorthand way by using the execute() method of Engine itself:

```
result = engine.execute("select username from users")
for row in result:
    print "username:", row['username']
```

Where above, the execute() method acquires a new Connection on its own, executes the statement with that object, and returns the ResultProxy. In this case, the ResultProxy contains a special flag known as close_with_result, which indicates that when its underlying DBAPI cursor is closed, the Connection object itself is also closed, which again returns the DBAPI connection to the connection pool, releasing transactional resources.

If the ResultProxy potentially has rows remaining, it can be instructed to close out its resources explicitly:

```
result.close()
```

If the ResultProxy has pending rows remaining and is dereferenced by the application without being closed, Python garbage collection will ultimately close out the cursor as well as trigger a return of the pooled DBAPI connection resource to the pool (SQLAlchemy achieves this by the usage of weakref callbacks - *never* the __del__ method) - however it's never a good idea to rely upon Python garbage collection to manage resources.

Our example above illustrated the execution of a textual SQL string. The execute() method can of course accommodate more than that, including the variety of SQL expression constructs described in *SQL Expression Language Tutorial*.

class sqlalchemy.engine.base.**Connection**(*engine*, *connection=None*, *close_with_result=False*, *_branch=False*, *_execution_options=None*)

Bases: sqlalchemy.engine.base.Connectable

Provides high-level functionality for a wrapped DB-API connection.

Provides execution support for string-based SQL statements as well as ClauseElement, Compiled and DefaultGenerator objects. Provides a begin() method to return Transaction objects.

The Connection object is **not** thread-safe. While a Connection can be shared among threads using properly synchronized access, it is still possible that the underlying DBAPI connection may not support shared access between threads. Check the DBAPI documentation for details.

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.

__init__(*engine*, *connection=None*, *close_with_result=False*, *_branch=False*, *_execution_options=None*)

Construct a new Connection.

The constructor here is not public and is only called only by an Engine. See Engine.connect() and Engine.contextual_connect() methods.

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.

Parameters

- **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 the given construct and returns a `ResultProxy`.

The construct can be one of:

- a textual SQL string
- any `ClauseElement` construct that is also a subclass of `Executable`, such as a `select()` construct
- a `FunctionElement`, such as that generated by `func`, will be automatically wrapped in a SELECT statement, which is then executed.
- a `DDLElement` object
- a `DefaultGenerator` object
- a `Compiled` object

execution_options(***opt*)

Set non-SQL options for the connection which take effect during execution.

The method returns a copy of this `Connection` which references the same underlying DBAPI connection, but also defines the given execution options which will take effect for a call to `execute()`. As the new `Connection` references the same underlying resource, it is probably best to ensure that the copies would be discarded immediately, which is implicit if used as in:

```
result = connection.execution_options(stream_results=True).
```

The options are the same as those accepted by `sqlalchemy.sql.expression.Executable.execution_optio`

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.

transaction(*callable_*, **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.

See also transaction() on engine.

class `sqlalchemy.engine.base.`**`Connectable`**

Bases: `object`

Interface for an object which supports execution of SQL constructs.

The two implementations of `Connectable` are `Connection` and `Engine`.

Connectable must also implement the 'dialect' member which references a `Dialect` instance.

`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.

class `sqlalchemy.engine.base.`**`Engine`**`(`*pool*, *dialect*, *url*, *logging_name=None*, *echo=None*, *proxy=None*, *execution_options=None*`)`

Bases: `sqlalchemy.engine.base.Connectable`, `sqlalchemy.log.Identified`

Connects a `Pool` and `Dialect` together to provide a source of database connectivity and behavior.

An `Engine` object is instantiated publically using the `create_engine()` function.

`__init__``(`*pool*, *dialect*, *url*, *logging_name=None*, *echo=None*, *proxy=None*, *execution_options=None*`)`

`connect``(`***kwargs*`)`

Return a new `Connection` object.

The `Connection`, upon construction, will procure a DBAPI connection from the `Pool` referenced by this `Engine`, returning it back to the `Pool` after the `Connection.close()` method is called.

`contextual_connect``(`*close_with_result=False*, ***kwargs*`)`

Return a `Connection` object which may be part of some ongoing context.

By default, this method does the same thing as `Engine.connect()`. Subclasses of `Engine` may override this method to provide contextual behavior.

Parameters

- **close_with_result** – When True, the first `ResultProxy` created by the `Connection` will call the `Connection.close()` method of that connection as soon as any pending result rows are exhausted. This is used to supply the "connectionless execution" behavior provided by the `Engine.execute()` method.

`create``(`*entity*, *connection=None*, ***kwargs*`)`

Create a table or index within this engine's database connection given a schema object.

`dispose``()`

Dispose of the connection pool used by this `Engine`.

A new connection pool is created immediately after the old one has been disposed. This new pool, like all SQLAlchemy connection pools, does not make any actual connections to the database until one is first requested.

This method has two general use cases:

- When a dropped connection is detected, it is assumed that all connections held by the pool are potentially dropped, and the entire pool is replaced.
- An application may want to use `dispose()` within a test suite that is creating multiple engines.

It is critical to note that `dispose()` does **not** guarantee that the application will release all open database connections - only those connections that are checked into the pool are closed. Connections which remain checked out or have been detached from the engine are not affected.

driver
Driver name of the `Dialect` in use by this `Engine`.

drop (*entity*, *connection=None*, ***kwargs*)
Drop a table or index within this engine's database connection given a schema 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`.

execute (*statement*, **multiparams*, ***params*)
Executes the given construct and returns a `ResultProxy`.

The arguments are the same as those used by `Connection.execute()`.

Here, a `Connection` is acquired using the `contextual_connect()` method, and the statement executed with that connection. The returned `ResultProxy` is flagged such that when the `ResultProxy` is exhausted and its underlying cursor is closed, the `Connection` created here will also be closed, which allows its associated DBAPI connection resource to be returned to the connection pool.

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.

Parameters

- **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 `text()` construct, bound to this engine.

This is equivalent to:

```
text("SELECT * FROM table", bind=engine)
```

transaction (*callable_*, **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.

The connection used is that of contextual_connect().

See also the similar method on Connection itself.

update_execution_options (***opt*)
update the execution_options dictionary of this `Engine`.

For details on execution_options, see `Connection.execution_options()` as well as `sqlalchemy.sql.expression.Executable.execution_options()`.

class `sqlalchemy.engine.base.`**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 handles post-processing of result column data using `TypeEngine` objects, which are referenced from the originating SQL statement that produced this result set.

__init__ (*context*)

close (*_autoclose_connection=True*)
Close this ResultProxy.

Closes the underlying DBAPI cursor corresponding to the execution.

Note that any data cached within this ResultProxy is still available. For some types of results, this may include buffered rows.

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)`.

If rows are present, the cursor remains open after this is called. Else the cursor is automatically closed and an empty list is returned.

fetchone ()
Fetch one row, just like DB-API `cursor.fetchone()`.

If a row is present, the cursor remains open after this is called. Else the cursor is automatically closed and None is returned.

first ()
Fetch the first row and then close the result set unconditionally.

Returns None if no row is present.

inserted_primary_key
Return the primary key for the row just inserted.

This only applies to single row insert() constructs which did not explicitly specify returning().

keys()
Return the current set of string keys for rows.

last_inserted_ids()
Return the primary key for the row just inserted. Deprecated since version 0.6: Use `ResultProxy.inserted_primary_key`

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.

lastrowid
return the 'lastrowid' accessor on the DBAPI cursor.

This is a DBAPI specific method and is only functional for those backends which support it, for statements where it is appropriate. It's behavior is not consistent across backends.

Usage of this method is normally unnecessary; the `inserted_primary_key` attribute provides a tuple of primary key values for a newly inserted row, regardless of database backend.

postfetch_cols()
Return `postfetch_cols()` from the underlying ExecutionContext.

See ExecutionContext for details.

rowcount
Return the 'rowcount' for this result.

The 'rowcount' reports the number of rows affected by an UPDATE or DELETE statement. It has *no* other uses and is not intended to provide the number of rows present from a SELECT.

Note that this row count may not be properly implemented in some dialects; this is indicated by `supports_sane_rowcount()` and `supports_sane_multi_rowcount()`. `rowcount()` also may not work at this time for a statement that uses `returning()`.

scalar()
Fetch the first column of the first row, and close the result set.

Returns None if no row is present.

supports_sane_multi_rowcount()
Return `supports_sane_multi_rowcount` from the dialect.

supports_sane_rowcount()
Return `supports_sane_rowcount` from the dialect.

class `sqlalchemy.engine.base.`**RowProxy**(*parent, row, processors, keymap*)
Proxy values from a single cursor row.

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).

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.

3.4.2 Using Transactions

Note: This section describes how to use transactions when working directly with `Engine` and `Connection` objects. When using the SQLAlchemy ORM, the public API for transaction control is via the `Session` object, which makes usage of the `Transaction` object internally. See *Managing Transactions* for further information.

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
```

Nesting of Transaction Blocks

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.

class `sqlalchemy.engine.base.`**`Transaction`**`(`*connection*, *parent*`)`

Represent a Transaction in progress.

The object provides `rollback()` and `commit()` methods in order to control transaction boundaries. It also implements a context manager interface so that the Python `with` statement can be used with the `Connection.begin()` method.

The Transaction object is **not** threadsafe.

`__init__``(`*connection*, *parent*`)`

The constructor for `Transaction` is private and is called from within the `Connection.begin` implementation.

`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``()`

Commit this `Transaction`.

`rollback``()`

Roll back this `Transaction`.

3.4.3 Understanding Autocommit

The previous 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 DBAPI implementation provide various special "non-transactional" modes, the current SQLAlchemy behavior is such that it implements its own "autocommit" which works completely consistently across all backends. This is achieved by detecting statements which represent data-changing operations, i.e. INSERT, UPDATE, DELETE, as well as data definition language (DDL) statements such as CREATE TABLE, ALTER TABLE, 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
```

Full control of the "autocommit" behavior is available using the generative `Connection.execution_options()` method provided on `Connection`, `Engine`, `Executable`, using the "autocommit" flag which will turn on or off the autocommit for the selected scope. For example, a `text()`

construct representing a stored procedure that commits might use it so that a SELECT statement will issue a COMMIT:

```
engine.execute(text("SELECT my_mutating_procedure()").execution_options(autocommit=True))
```

3.4.4 Connectionless Execution, Implicit Execution

Recall from the first section we mentioned executing with and without explicit usage of `Connection`. "Connectionless" execution refers to the usage of the `execute()` method on an object which is not a `Connection`. This was illustrated using the `execute()` method of `Engine`.

In addition to "connectionless" execution, it is also possible to use the `execute()` method of any `Executable` construct, which is a marker for SQL expression objects that support execution. The SQL expression object itself references an `Engine` or `Connection` known as the **bind**, which it uses in order to provide so-called "implicit" execution services.

Given a table as below:

```
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 *Schema Definition Language*):

```
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 the `ResultProxy` is closed, the underlying `Connection` is closed for us, resulting in the DBAPI connection being returned to the pool with transactional resources removed.

3.4.5 Using the Threadlocal Execution Strategy

The "threadlocal" engine strategy is an optional feature which can be used by non-ORM applications to associate transactions with 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')
```

The above `Engine` will now acquire a `Connection` using connection resources derived from a thread-local variable whenever `Engine.execute()` or `Engine.contextual_connect()` is called. This connection resource is maintained as long as it is referenced, which allows multiple points of an application to share a transaction while using connectionless execution:

```
def call_operation1():
    engine.execute("insert into users values (?, ?)", 1, "john")

def call_operation2():
    users.update(users.c.user_id==5).execute(name='ed')

db.begin()
try:
    call_operation1()
    call_operation2()
    db.commit()
except:
    db.rollback()
```

Explicit execution can be mixed with connectionless execution by using the `Engine.connect` method to acquire a `Connection` that is not part of the threadlocal scope:

```
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()
```

To access the `Connection` that is bound to the threadlocal scope, call `Engine.contextual_connect()`:

```
conn = db.contextual_connect()
call_operation3(conn)
conn.close()
```

Calling `close()` on the "contextual" connection does not release its resources until all other usages of that resource are closed as well, including that any ongoing transactions are rolled back or committed.

3.5 Connection Pooling

The establishment of a database connection is typically a somewhat expensive operation, and applications need a way to get at database connections repeatedly with minimal overhead. Particularly for server-side web applications, a connection pool is the standard way to maintain a "pool" of active database connections in memory which are reused across requests.

SQLAlchemy includes several connection pool implementations which integrate with the `Engine`. They can also be used directly for applications that want to add pooling to an otherwise plain DBAPI approach.

3.5.1 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('postgresql://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, as well as to provide a reasonable default behavior to SQLite "memory" databases, which maintain their entire dataset within the scope of a single connection.

All SQLAlchemy pool implementations have in common that none of them "pre create" connections - all implementations wait until first use before creating a connection. At that point, if no additional concurrent checkout requests for more connections are made, no additional connections are created. This is why it's perfectly fine for `create_engine()` to default to using a `QueuePool` of size five without regard to whether or not the application really needs five connections queued up - the pool would only grow to that size if the application actually used five connections concurrently, in which case the usage of a small pool is an entirely appropriate default behavior.

3.5.2 Switching Pool Implementations

The usual way to use a different kind of pool with `create_engine()` is to use the `poolclass` argument. This argument accepts a class imported from the `sqlalchemy.pool` module, and handles the details of building the pool for you. Common options include specifying `QueuePool` with SQLite:

```
from sqlalchemy.pool import QueuePool
engine = create_engine('sqlite:///file.db', poolclass=QueuePool)
```

Disabling pooling using `NullPool`:

```
from sqlalchemy.pool import NullPool
engine = create_engine(
          'postgresql+psycopg2://scott:tiger@localhost/test',
          poolclass=NullPool)
```

3.5.3 Using a Custom Connection Function

All `Pool` classes accept an argument `creator` which is a callable that creates a new connection. `create_engine()` accepts this function to pass onto the pool via an argument of the same name:

```
import sqlalchemy.pool as pool
import psycopg2

def getconn():
    c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')
    # do things with 'c' to set up
    return c

engine = create_engine('postgresql+psycopg2://', creator=getconn)
```

For most "initialize on connection" routines, it's more convenient to use a `PoolListener`, so that the usual URL argument to `create_engine()` is still usable. `creator` is there as a total last resort for when a DBAPI has some form of `connect` that is not at all supported by SQLAlchemy.

3.5.4 Constructing a Pool

To use a `Pool` by itself, the `creator` function is the only argument that's required and is passed first, followed by any additional options:

```
import sqlalchemy.pool as pool
import psycopg2

def getconn():
    c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')
    return c

mypool = pool.QueuePool(getconn, max_overflow=10, pool_size=5)
```

DBAPI connections can then be procured from the pool using the `Pool.connect()` function. The return value of this method is a DBAPI connection that's contained within a transparent proxy:

```
# get a connection
conn = mypool.connect()

# use it
cursor = conn.cursor()
cursor.execute("select foo")
```

The purpose of the transparent proxy is to intercept the `close()` call, such that instead of the DBAPI connection being closed, its returned to the pool:

```
# "close" the connection.  Returns
# it to the pool.
conn.close()
```

The proxy also returns its contained DBAPI connection to the pool when it is garbage collected, though it's not deterministic in Python that this occurs immediately (though it is typical with cPython).

A particular pre-created `Pool` can be shared with one or more engines by passing it to the `pool` argument of `create_engine()`:

```
e = create_engine('postgresql://', pool=mypool)
```

3.5.5 Pool Event Listeners

Connection pools support an event interface that allows hooks to execute upon first connect, upon each new connection, and upon checkout and checkin of connections. See `PoolListener` for details.

3.5.6 Builtin Pool Implementations

class `sqlalchemy.pool.`**`Pool`**(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*)

Abstract base class for connection pools.

`__init__`(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *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.
- **logging_name** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.pool" logger. Defaults to a hexstring of the object's id.
- **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.

`connect`()

Return a DBAPI connection from the pool.

The connection is instrumented such that when its `close()` method is called, the connection will be returned to the pool.

`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.

`recreate`()

Return a new `Pool`, of the same class as this one and configured with identical creation arguments.

This method is used in conjunection with `dispose()` to close out an entire `Pool` and create a new one in its place.

class `sqlalchemy.pool.`**`QueuePool`**(*creator*, *pool_size=5*, *max_overflow=10*, *timeout=30*, ***kw*)

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*, ***kw*)

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, defaults to 5. 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. `pool_size` can be set to 0 to indicate no size limit; to disable pooling, use a `NullPool` instead.
- **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 `sqlalchemy.pool.`**`SingletonThreadPool`**(*creator*, *pool_size=5*, ***kw*)

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:

Parameters

- **pool_size** – The number of threads in which to maintain connections at once. Defaults to five.

__init__(*creator*, *pool_size=5*, ***kw*)

class sqlalchemy.pool.**AssertionPool**(**args*, ***kw*)
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.

class sqlalchemy.pool.**NullPool**(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *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 sqlalchemy.pool.**StaticPool**(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*)
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.

3.5.7 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 weakref callbacks (`__del__` is not used).

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.

`sqlalchemy.pool.`**`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*

`sqlalchemy.pool.`**`clear_managers`**`()`

Remove all current DB-API 2.0 managers.

All pools and connections are disposed.

3.6 Schema Definition Language

3.6.1 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 are at the core of three major types of operations - issuing CREATE and DROP statements (known as *DDL*), constructing SQL queries, and expressing information about structures that already exist within the database.

Database metadata can be expressed by explicitly naming the various components and their properties, using constructs such as `Table`, `Column`, `ForeignKey` and `Sequence`, all of which are imported from the `sqlalchemy.schema` package. It can also be generated by SQLAlchemy using a process called *reflection*, which means you start with a single object such as `Table`, assign it a name, and then instruct SQLAlchemy to load all the additional information related to that name from a particular engine source.

A key feature of SQLAlchemy's database metadata constructs is that they are designed to be used in a *declarative* style which closely resembles that of real DDL. They are therefore most intuitive to those who have some background in creating real schema generation scripts.

A collection of metadata entities is stored in an object aptly named `MetaData`:

```
from sqlalchemy import *

metadata = MetaData()
```

`MetaData` is a container object that keeps together many different features of a database (or multiple databases) being described.

To represent a table, use the `Table` class. Its two primary arguments are the table name, then the `MetaData` object which it will be associated with. The remaining positional arguments are mostly `Column` objects describing each column:

```
user = Table('user', metadata,
    Column('user_id', Integer, primary_key = True),
    Column('user_name', String(16), nullable = False),
```

```
    Column('email_address', String(60)),
    Column('password', String(20), nullable = False)
)
```

Above, a table called `user` is described, which contains four columns. The primary key of the table consists of the `user_id` column. Multiple columns may be assigned the `primary_key=True` flag which denotes a multi-column primary key, known as a *composite* primary key.

Note also that each column describes its datatype using objects corresponding to genericized types, such as `Integer` and `String`. SQLAlchemy features dozens of types of varying levels of specificity as well as the ability to create custom types. Documentation on the type system can be found at *types*.

Accessing Tables and Columns

The `MetaData` object contains all of the schema constructs we've associated with it. It supports a few methods of accessing these table objects, such as the `sorted_tables` accessor which returns a list of each `Table` object in order of foreign key dependency (that is, each table is preceded by all tables which it references):

```
>>> for t in metadata.sorted_tables:
...     print t.name
user
user_preference
invoice
invoice_item
```

In most cases, individual `Table` objects have been explicitly declared, and these objects are typically accessed directly as module-level variables in an application. Once a `Table` has been defined, it has a full set of accessors which allow inspection of its properties. Given the following `Table` definition:

```
employees = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('employee_name', String(60), nullable=False),
    Column('employee_dept', Integer, ForeignKey("departments.department_id"))
)
```

Note the `ForeignKey` object used in this table - this construct defines a reference to a remote table, and is fully described in *Defining Foreign Keys*. Methods of accessing information about this table include:

```
# 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_keys

# get the "key" of a column, which defaults to its name, but can
# be any user-defined string:
employees.c.employee_name.key

# access a column's table:
employees.c.employee_id.table is employees

# get the table related by a foreign key
list(employees.c.employee_dept.foreign_keys)[0].column.table
```

Creating and Dropping Database Tables

Once you've defined some `Table` objects, assuming you're working with a brand new database one thing you might want to do is issue CREATE statements for those tables and their related constructs (as an aside, it's also quite possible that you *don't* want to do this, if you already have some preferred methodology such as tools included with your database or an existing scripting system - if that's the case, feel free to skip this section - SQLAlchemy has no requirement that it be used to create your tables).

The usual way to issue CREATE is to use `create_all()` on the `MetaData` object. This method will issue queries that first check for the existence of each individual table, and if not found will issue the CREATE statements:

```
engine = create_engine('sqlite:///:memory:')

metadata = MetaData()

user = Table('user', 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("user.user_id"), nullable=False),
    Column('pref_name', String(40), nullable=False),
    Column('pref_value', String(100))
)
```

```
metadata.create_all(engine)
PRAGMA table_info(user){}
CREATE TABLE user(
        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 user(user_id),
        pref_name VARCHAR(40) NOT NULL,
        pref_value VARCHAR(100)
)
```

`create_all()` creates foreign key constraints between tables usually inline with the table definition itself, and for this reason it also generates the tables in order of their dependency. There are options to change this behavior such that `ALTER TABLE` is used instead.

Dropping all tables is similarly achieved using the `drop_all()` method. This method does the exact opposite of `create_all()` - the presence of each table is checked first, and tables are dropped in reverse order of dependency.

Creating and dropping individual tables can be done via the `create()` and `drop()` methods of `Table`. These methods by default issue the CREATE or DROP regardless of the table being present:

```
engine = create_engine('sqlite:///:memory:')

meta = MetaData()

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(engine)
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(engine)
DROP TABLE employee
```

To enable the "check first for the table existing" logic, add the `checkfirst=True` argument to `create()` or `drop()`:

```
employees.create(engine, checkfirst=True)
employees.drop(engine, checkfirst=False)
```

Binding MetaData to an Engine or Connection

Notice in the previous section the creator/dropper methods accept an argument for the database engine in use. When a schema construct is combined with an `Engine` object, or an individual `Connection` object, we call this the *bind*. In the above examples the bind is associated with the schema construct only for the duration of the operation. However, the option exists to persistently associate a bind with a set of schema constructs via the `MetaData` object's `bind` attribute:

```
engine = create_engine('sqlite://')

# create MetaData
meta = MetaData()

# bind to an engine
meta.bind = engine
```

We can now call methods like `create_all()` without needing to pass the `Engine`:

```
meta.create_all()
```

The MetaData's bind is used for anything that requires an active connection, such as loading the definition of a table from the database automatically (called *reflection*):

```
# describe a table called 'users', query the database for its columns
users_table = Table('users', meta, autoload=True)
```

As well as for executing SQL constructs that are derived from that MetaData's table objects:

```
# generate a SELECT statement and execute
result = users_table.select().execute()
```

Binding the MetaData to the Engine is a **completely optional** feature. The above operations can be achieved without the persistent bind using parameters:

```
# 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())
```

Should you use bind ? It's probably best to start without it, and wait for a specific need to arise. Bind is useful if:

- You aren't using the ORM, are usually using "connectionless" execution, and find yourself constantly needing to specify the same `Engine` object throughout the entire application. Bind can be used here to provide "implicit" execution.
- Your application has multiple schemas that correspond to different engines. Using one `MetaData` for each schema, bound to each engine, provides a decent place to delineate between the schemas. The ORM will also integrate with this approach, where the `Session` will naturally use the engine that is bound to each table via its metadata (provided the `Session` itself has no `bind` configured.).

Alternatively, the `bind` attribute of `MetaData` is *confusing* if:

- Your application talks to multiple database engines at different times, which use the *same* set of `Table` objects. It's usually confusing and unnecessary to begin to create "copies" of `Table` objects just so that different engines can be used for different operations. An example is an application that writes data to a "master" database while performing read-only operations from a "read slave". A global `MetaData` object is *not* appropriate for per-request switching like this, although a `ThreadLocalMetaData` object is.
- You are using the ORM `Session` to handle which class/table is bound to which engine, or you are using the `Session` to manage switching between engines. Its a good idea to keep the "binding of tables to engines" in one place - either using `MetaData` only (the `Session` can of course be present, it just has no `bind` configured), or using `Session` only (the `bind` attribute of `MetaData` is left empty).

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 specify references to columns in this table using the form `remote_banks.financial_info.id`.

The `schema` argument should be used for any name qualifiers required, including Oracle's "owner" attribute and similar. It also can accommodate a dotted name for longer schemes:

```
schema="dbo.scott"
```

Backend-Specific Options

`Table` supports database-specific options. For example, MySQL has different table backend types, including "MyISAM" and "InnoDB". This can be expressed with `Table` using `mysql_engine`:

```
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'
)
```

Other backends may support table-level options as well - these would be described in the individual documentation sections for each dialect.

Schema API Constructs

class `sqlalchemy.schema.`**`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.

  There is partial support for automatic detection of the type based on that of a ForeignKey associated with this column, if the type is specified as None. However, this feature is not fully implemented and may not function in all cases.

- ***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 indicate an integer primary key column that should not be considered to be the "autoincrement" column, that is the integer primary key column which generates values implicitly upon INSERT and whose value is usually returned via the DBAPI cursor.lastrowid attribute. It defaults to True to satisfy the common use case of a table with a single integer primary key column. If the table has a composite primary key consisting of more than one integer column, set this flag to True only on the column that should be considered "autoincrement".

  The setting *only* has an effect for columns which are:

  – Integer derived (i.e. INT, SMALLINT, BIGINT)

  – Part of the primary key

  – Are not referenced by any foreign keys

  – have no server side or client side defaults (with the exception of Postgresql SERIAL).

  The setting has these two effects on columns that meet the above criteria:

  – DDL issued for the column will include database-specific keywords intended to signify this column as an "autoincrement" column, such as AUTO INCREMENT on MySQL, SERIAL on Postgresql, and IDENTITY on MS-SQL. It does *not* issue AUTOINCREMENT for SQLite since this is a special SQLite flag that is not required for autoincrementing behavior. See the SQLite dialect documentation for information on SQLite's AUTOINCREMENT.

  – The column will be considered to be available as cursor.lastrowid or equivalent, for those dialects which "post fetch" newly inserted identifiers after a row has been inserted (SQLite, MySQL, MS-SQL). It does not have any effect in this regard for databases that use sequences to generate primary key identifiers (i.e. Firebird, Postgresql, Oracle).

- **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.

- **doc** – optional String that can be used by the ORM or similar to document attributes. This attribute does not render SQL comments (a future attribute 'comment' will achieve that).
- **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*)

copy (***kw*)
Create a copy of this `Column`, unitialized.

This is used in `Table.tometadata`.

get_children (*schema_visitor=False*, ***kwargs*)

references (*column*)
Return True if this Column references the given column via foreign key.

class `sqlalchemy.schema.`**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.

Parameters

- **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.

Parameters

- **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

  target 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.

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.

Parameters

- **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.

Parameters

- **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*, *views=False*, *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.

Parameters

- **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.
- **views** – If True, also reflect views.
- **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.

class `sqlalchemy.schema.`**Table**(**args*, ***kw*)
Bases: `sqlalchemy.schema.SchemaItem`, `sqlalchemy.sql.expression.TableClause`

Represent a table in a database.

e.g.:

```
mytable = Table("mytable", metadata,
                Column('mytable_id', Integer, primary_key=True),
                Column('value', String(50))
           )
```

The Table object constructs a unique instance of itself based on its name within the given MetaData object. Constructor arguments are as follows:

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.
- **implicit_returning** – True by default - indicates that RETURNING can be used by default to fetch newly inserted primary key values, for backends which support this. Note that create_engine() also provides an implicit_returning flag.
- **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.

`__init__`(**args*, ***kw*)

`add_is_dependent_on`(*table*)

Add a 'dependency' for this Table.

This is another Table object which must be created first before this one can, or dropped after this one.

Usually, dependencies between tables are determined via ForeignKey objects. However, for other situations that create dependencies outside of foreign keys (rules, inheriting), this method can manually establish such a link.

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.

Parameters

- **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

  target 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 Table.

create (*bind=None, checkfirst=False*)
Issue a `CREATE` statement for this table.

See also `metadata.create_all()`.

drop (*bind=None, checkfirst=False*)
Issue a `DROP` statement for this table.

See also `metadata.drop_all()`.

exists (*bind=None*)
Return True if this table exists.

get_children (*column_collections=True, schema_visitor=False, **kw*)

key

primary_key

tometadata (*metadata, schema=<symbol 'retain_schema>*)
Return a copy of this `Table` associated with a different `MetaData`.

E.g.:

```
# 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)
```

class sqlalchemy.schema.**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().

dispose()

Dispose all bound engines, in all thread contexts.

is_bound()

True if there is a bind for this thread.

3.6.2 Reflecting Database Objects

A Table object can be instructed to load information about itself from the corresponding database schema object already existing within the database. This process is called *reflection*. Most simply you need only specify the table name, a MetaData object, and the autoload=True flag. If the MetaData is not persistently bound, also add the autoload_with argument:

```
>>> messages = Table('messages', meta, autoload=True, autoload_with=engine)
>>> [c.name for c in messages.columns]
['message_id', 'message_name', 'date']
```

The above operation will use the given engine to query the database for information about the messages table, and will then generate Column, ForeignKey, and other objects corresponding to this information as though the Table object were hand-constructed in Python.

When tables are reflected, if a given table references another one via foreign key, a second Table object is created within the MetaData object representing the connection. Below, assume the table shopping_cart_items references a table named shopping_carts. Reflecting the shopping_cart_items table has the effect such that the shopping_carts table will also be loaded:

```
>>> shopping_cart_items = Table('shopping_cart_items', meta, autoload=True, autoload_with=e
>>> 'shopping_carts' in meta.tables:
True
```

The MetaData has an interesting "singleton-like" behavior such that if you requested both tables individually, MetaData will ensure that exactly one Table object is created for each distinct table name. The Table constructor actually returns to you the already-existing Table object if one already exists with the given name. Such as below, we can access the already generated shopping_carts table just by naming it:

```
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 the table's attributes will be loaded if they have not been already. The autoload operation only occurs for the table if it hasn't already been loaded; once loaded, new calls to Table with the same name 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 Views

The reflection system can also reflect views. Basic usage is the same as that of a table:

```
my_view = Table("some_view", metadata, autoload=True)
```

Above, `my_view` is a `Table` object with `Column` objects representing the names and types of each column within the view "some_view".

Usually, it's desired to have at least a primary key constraint when reflecting a view, if not foreign keys as well. View reflection doesn't extrapolate these constraints.

Use the "override" technique for this, specifying explicitly those columns which are part of the primary key or have foreign key constraints:

```
my_view = Table("some_view", metadata,
                Column("view_id", Integer, primary_key=True),
                Column("related_thing", Integer, ForeignKey("othertable.thing_id")),
                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()` also provides a handy way to clear or delete all the rows in a database:

```
meta = MetaData()
meta.reflect(bind=someengine)
for table in reversed(meta.sorted_tables):
    someengine.execute(table.delete())
```

Fine Grained Reflection with Inspector

A low level interface which provides a backend-agnostic system of loading lists of schema, table, column, and constraint descriptions from a given database is also available. This is known as the "Inspector":

```
from sqlalchemy import create_engine
from sqlalchemy.engine import reflection
engine = create_engine('...')
```

```
insp = reflection.Inspector.from_engine(engine)
print insp.get_table_names()
```

class `sqlalchemy.engine.reflection.`**`Inspector`**(*bind*)

Bases: `object`

Performs database schema inspection.

The Inspector acts as a proxy to the reflection methods of the `Dialect`, providing a consistent interface as well as caching support for previously fetched metadata.

The preferred method to construct an `Inspector` is via the `Inspector.from_engine()` method. I.e.:

```
engine = create_engine('...')
insp = Inspector.from_engine(engine)
```

Where above, the `Dialect` may opt to return an `Inspector` subclass that provides additional methods specific to the dialect's target database.

`__init__`(*bind*)

Initialize a new `Inspector`.

Parameters

- **bind** – a `Connectable`, which is typically an instance of `Engine` or `Connection`.

For a dialect-specific instance of `Inspector`, see `Inspector.from_engine()`

`default_schema_name`

Return the default schema name presented by the dialect for the current engine's database user.

E.g. this is typically `public` for Postgresql and `dbo` for SQL Server.

classmethod **`from_engine`**(*bind*)

Construct a new dialect-specific Inspector object from the given engine or connection.

Parameters

- **bind** – a `Connectable`, which is typically an instance of `Engine` or `Connection`.

This method differs from direct a direct constructor call of `Inspector` in that the `Dialect` is given a chance to provide a dialect-specific `Inspector` instance, which may provide additional methods.

See the example at `Inspector`.

`get_columns`(*table_name*, *schema=None*, ***kw*)

Return information about columns in *table_name*.

Given a string *table_name* and an optional string *schema*, return column information as a list of dicts with these keys:

name the column's name

type `TypeEngine`

nullable boolean

default the column's default value

attrs dict containing optional column attributes

get_foreign_keys(*table_name*, *schema=None*, ***kw*)
Return information about foreign_keys in *table_name*.

Given a string *table_name*, and an optional string *schema*, return foreign key information as a list of dicts with these keys:

constrained_columns a list of column names that make up the foreign key

referred_schema the name of the referred schema

referred_table the name of the referred table

referred_columns a list of column names in the referred table that correspond to constrained_columns

name optional name of the foreign key constraint.

****kw** other options passed to the dialect's get_foreign_keys() method.

get_indexes(*table_name*, *schema=None*, ***kw*)
Return information about indexes in *table_name*.

Given a string *table_name* and an optional string *schema*, return index information as a list of dicts with these keys:

name the index's name

column_names list of column names in order

unique boolean

****kw** other options passed to the dialect's get_indexes() method.

get_pk_constraint(*table_name*, *schema=None*, ***kw*)
Return information about primary key constraint on *table_name*.

Given a string *table_name*, and an optional string *schema*, return primary key information as a dictionary with these keys:

constrained_columns a list of column names that make up the primary key

name optional name of the primary key constraint.

get_primary_keys(*table_name*, *schema=None*, ***kw*)
Return information about primary keys in *table_name*.

Given a string *table_name*, and an optional string *schema*, return primary key information as a list of column names.

get_schema_names()
Return all schema names.

get_table_names(*schema=None*, *order_by=None*)
Return all table names in *schema*.

Parameters

- **schema** – Optional, retrieve names from a non-default schema.
- **order_by** – Optional, may be the string "foreign_key" to sort the result on foreign key dependencies.

This should probably not return view names or maybe it should return them with an indicator t or v.

get_table_options(*table_name*, *schema=None*, ***kw*)
Return a dictionary of options specified when the table of the given name was created.

This currently includes some options that apply to MySQL tables.

get_view_definition (*view_name*, *schema=None*)
Return definition for *view_name*.

Parameters

- **schema** – Optional, retrieve names from a non-default schema.

get_view_names (*schema=None*)
Return all view names in *schema*.

Parameters

- **schema** – Optional, retrieve names from a non-default schema.

reflecttable (*table*, *include_columns*)
Given a Table object, load its internal constructs based on introspection.

This is the underlying method used by most dialects to produce table reflection. Direct usage is like:

```
from sqlalchemy import create_engine, MetaData, Table
from sqlalchemy.engine import reflection

engine = create_engine('...')
meta = MetaData()
user_table = Table('user', meta)
insp = Inspector.from_engine(engine)
insp.reflecttable(user_table, None)
```

Parameters

- **table** – a `Table` instance.
- **include_columns** – a list of string column names to include in the reflection process. If `None`, all columns are reflected.

3.6.3 Column Insert/Update Defaults

SQLAlchemy provides a very rich featureset regarding column level events which take place during INSERT and UPDATE statements. Options include:

- Scalar values used as defaults during INSERT and UPDATE operations
- Python functions which execute upon INSERT and UPDATE operations
- SQL expressions which are embedded in INSERT statements (or in some cases execute beforehand)
- SQL expressions which are embedded in UPDATE statements
- Server side default values used during INSERT
- Markers for server-side triggers used during UPDATE

The general rule for all insert/update defaults is that they only take effect if no value for a particular column is passed as an `execute()` parameter; otherwise, the given value is used.

Scalar Defaults

The simplest kind of default is a scalar value used as the default value of a column:

```
Table("mytable", meta,
    Column("somecolumn", Integer, default=12)
)
```

Above, the value "12" will be bound as the column value during an INSERT if no other value is supplied.

A scalar value may also be associated with an UPDATE statement, though this is not very common (as UPDATE statements are usually looking for dynamic defaults):

```
Table("mytable", meta,
    Column("somecolumn", Integer, onupdate=25)
)
```

Python-Executed Functions

The `default` and `onupdate` keyword arguments also accept Python functions. These functions are invoked at the time of insert or update if no other value for that column is supplied, and the value returned is used for the column's value. Below illustrates a crude "sequence" that assigns an incrementing counter to a primary key column:

```
# a function which counts upwards
i = 0
def mydefault():
    global i
    i += 1
    return i

t = Table("mytable", meta,
    Column('id', Integer, primary_key=True, default=mydefault),
)
```

It should be noted that for real "incrementing sequence" behavior, the built-in capabilities of the database should normally be used, which may include sequence objects or other autoincrementing capabilities. For primary key columns, SQLAlchemy will in most cases use these capabilities automatically. See the API documentation for `Column` including the `autoincrement` flag, as well as the section on `Sequence` later in this chapter for background on standard primary key generation techniques.

To illustrate onupdate, we assign the Python `datetime` function `now` to the `onupdate` attribute:

```
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),
)
```

When an update statement executes and no value is passed for `last_updated`, the `datetime.datetime.now()` Python function is executed and its return value used as the value for `last_updated`. Notice that we provide `now` as the function itself without calling it (i.e. there are no parenthesis following) - SQLAlchemy will execute the function at the time the statement executes.

Context-Sensitive Default Functions

The Python functions used by `default` and `onupdate` may also make use of the current statement's context in order to determine a value. The *context* of a statement is an internal SQLAlchemy object which contains all information about the statement being executed, including its source expression, the parameters associated with it and the cursor. The typical use case for this context with regards to default generation is to have access to the other values being inserted or updated on the row. To access the context, provide a function that accepts a single `context` argument:

```
def mydefault(context):
    return context.current_parameters['counter'] + 12

t = Table('mytable', meta,
    Column('counter', Integer),
    Column('counter_plus_twelve', Integer, default=mydefault, onupdate=mydefault)
)
```

Above we illustrate a default function which will execute for all INSERT and UPDATE statements where a value for `counter_plus_twelve` was otherwise not provided, and the value will be that of whatever value is present in the execution for the `counter` column, plus the number 12.

While the context object passed to the default function has many attributes, the `current_parameters` member is a special member provided only during the execution of a default function for the purposes of deriving defaults from its existing values. For a single statement that is executing many sets of bind parameters, the user-defined function is called for each set of parameters, and `current_parameters` will be provided with each individual parameter set for each execution.

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.utc_timestamp())
    )
```

Above, the `create_date` column will be populated with the result of the `now()` SQL function (which, depending on backend, compiles into `NOW()` or `CURRENT_TIMESTAMP` in most cases) during an INSERT statement, and the `key` column with the result of a SELECT subquery from another table. The `last_modified` column will be populated with the value of `UTC_TIMESTAMP()`, a function specific to MySQL, when an UPDATE statement is emitted for this table.

Note that when using `func` functions, unlike when using Python *datetime* functions we *do* call the function, i.e. with parenthesis "()" - this is because what we want in this case is the return value of the function, which is the SQL expression construct that will be rendered into the INSERT or UPDATE statement.

The above SQL functions are usually executed "inline" with the INSERT or UPDATE statement being executed, meaning, a single statement is executed which embeds the given expressions or subqueries within the VALUES or SET clause of the statement. Although in some cases, the function is "pre-executed" in a SELECT statement of its own beforehand. This happens when all of the following is true:

- the column is a primary key column

- the database dialect does not support a usable `cursor.lastrowid` accessor (or equivalent); this currently includes PostgreSQL, Oracle, and Firebird, as well as some MySQL dialects.
- the dialect does not support the "RETURNING" clause or similar, or the `implicit_returning` flag is set to `False` for the dialect. Dialects which support RETURNING currently include Postgresql, Oracle, Firebird, and MS-SQL.
- 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, and the statement has not defined an explicit *returning()* clause.

Whether or not the default generation clause "pre-executes" is not something that normally needs to be considered, unless it is being addressed for performance reasons.

When the statement is executed with a single set of parameters (that is, it is not an "executemany" style execution), 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 `inserted_primary_key` collection contains a list of primary key values for the row inserted (a list so that single-column and composite-column primary keys are represented in the same format).

Server Side Defaults

A variant on the 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.

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.

Defining Sequences

SQLAlchemy represents database sequences using the `Sequence` object, which is considered to be a special case of "column default". It only has an effect on databases which have explicit support for sequences, which currently includes Postgresql, Oracle, and Firebird. The `Sequence` object is otherwise ignored.

The `Sequence` may be placed on any column as a "default" generator to be used during INSERT operations, and can also be configured to fire off during UPDATE operations if desired. It is most commonly used in conjunction with a single integer primary key column:

```
table = Table("cartitems", meta,
    Column("cart_id", Integer, Sequence('cart_id_seq'), primary_key=True),
    Column("description", String(40)),
    Column("createdate", DateTime())
)
```

Where above, the table "cartitems" is associated with a sequence named "cart_id_seq". When INSERT statements take place for "cartitems", and no value is passed for the "cart_id" column, the "cart_id_seq" sequence will be used to generate a value.

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 `Sequence` object also implements special functionality to accommodate Postgresql's SERIAL datatype. The SERIAL type in PG automatically generates a sequence that is used implicitly during inserts. This means that if a `Table` object defines a `Sequence` on its primary key column so that it works with Oracle and Firebird, the `Sequence` would get in the way of the "implicit" sequence that PG would normally use. For this use case, add the flag `optional=True` to the `Sequence` object - this indicates that the `Sequence` should only be used if the database provides no other option for generating primary key identifiers.

The `Sequence` object also has the ability to be executed standalone like a SQL expression, which has the effect of calling its "next value" function:

```
seq = Sequence('some_sequence')
nextid = connection.execute(seq)
```

Default Geneation API Constructs

class `sqlalchemy.schema.`**`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.

`ColumnDefault` is generated automatically whenever the `default`, `onupdate` arguments of `Column` are used. A `ColumnDefault` can be passed positionally as well.

For example, the following:

```
Column('foo', Integer, default=50)
```

Is equivalent to:

```
Column('foo', Integer, ColumnDefault(50))
```

class `sqlalchemy.schema.`**`DefaultClause`**(*arg*, *for_update=False*)
Bases: `sqlalchemy.schema.FetchedValue`

A DDL-specified DEFAULT column value.

`DefaultClause` is a `FetchedValue` that also generates a "DEFAULT" clause when "CREATE TABLE" is emitted.

`DefaultClause` is generated automatically whenever the `server_default`, `server_onupdate` arguments of `Column` are used. A `DefaultClause` can be passed positionally as well.

For example, the following:

```
Column('foo', Integer, server_default="50")
```

Is equivalent to:

```
Column('foo', Integer, DefaultClause("50"))
```

class `sqlalchemy.schema.`**`DefaultGenerator`**(*for_update=False*)
Bases: `sqlalchemy.schema.SchemaItem`

Base class for column *default* values.

class `sqlalchemy.schema.`**`FetchedValue`**(*for_update=False*)
Bases: `object`

A marker for a transparent database-side default.

Use `FetchedValue` when the database is configured to provide some automatic default for a column.

E.g.:

```
Column('foo', Integer, FetchedValue())
```

Would indicate that some trigger or default generator will create a new value for the `foo` column during an INSERT.

class `sqlalchemy.schema.`**`PassiveDefault`**(**arg*, ***kw*)
Bases: `sqlalchemy.schema.DefaultClause`

A DDL-specified DEFAULT column value. Deprecated since version 0.6: `PassiveDefault` is deprecated. Use `DefaultClause`.

class `sqlalchemy.schema.`**`Sequence`**(*name*, *start=None*, *increment=None*, *schema=None*, *optional=False*, *quote=None*, *metadata=None*, *for_update=False*)
Bases: `sqlalchemy.schema.DefaultGenerator`

Represents a named database sequence.

3.6.4 Defining Constraints and Indexes

Defining Foreign Keys

A *foreign key* in SQL is a table-level construct that constrains one or more columns in that table to only allow values that are present in a different set of columns, typically but not always located on a different table. We call the columns which are constrained the *foreign key* columns and the columns which they are constrained towards the *referenced* columns. The referenced columns almost always define the primary key for their owning table, though there are exceptions to this. The foreign key is the "joint" that connects together pairs of rows which have a relationship with each other, and SQLAlchemy assigns very deep importance to this concept in virtually every area of its operation.

In SQLAlchemy as well as in DDL, foreign key constraints can be defined as additional attributes within the table clause, or for single-column foreign keys they may optionally be specified within the definition of a single column.

The single column foreign key is more common, and at the column level is specified by constructing a `ForeignKey` object as an argument to a `Column` object:

```
user_preference = Table('user_preference', metadata,
    Column('pref_id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey("user.user_id"), nullable=False),
    Column('pref_name', String(40), nullable=False),
    Column('pref_value', String(100))
)
```

Above, we define a new table `user_preference` for which each row must contain a value in the `user_id` column that also exists in the `user` table's `user_id` column.

The argument to `ForeignKey` is most commonly a string of the form <*tablename*>.<*columnname*>, or for a table in a remote schema or "owner" of the form <*schemaname*>.<*tablename*>.<*columnname*>. It may also be an actual `Column` object, which as we'll see later is accessed from an existing `Table` object via its `c` collection:

```
ForeignKey(user.c.user_id)
```

The advantage to using a string is that the in-python linkage between `user` and `user_preference` is resolved only when first needed, so that table objects can be easily spread across multiple modules and defined in any order.

Foreign keys may also be defined at the table level, using the `ForeignKeyConstraint` object. This object can describe a single- or multi-column foreign key. A multi-column foreign key is known as a *composite* foreign key, and almost always references a table that has a composite primary key. Below we define a table `invoice` which has a composite primary key:

```
invoice = Table('invoice', metadata,
    Column('invoice_id', Integer, primary_key=True),
    Column('ref_num', Integer, primary_key=True),
    Column('description', String(60), nullable=False)
)
```

And then a table `invoice_item` with a composite foreign key referencing `invoice`:

```
invoice_item = Table('invoice_item', 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'], ['invoice.invoice_id', 'invoice.ref_num
)
```

It's important to note that the `ForeignKeyConstraint` is the only way to define a composite foreign key. While we could also have placed individual `ForeignKey` objects on both the `invoice_item.invoice_id` and `invoice_item.ref_num` columns, SQLAlchemy would not be aware that these two values should be paired together - it would be two individual foreign key constraints instead of a single composite foreign key referencing two columns.

Creating/Dropping Foreign Key Constraints via ALTER

In all the above examples, the `ForeignKey` object causes the "REFERENCES" keyword to be added inline to a column definition within a "CREATE TABLE" statement when `create_all()` is issued, and `ForeignKeyConstraint` invokes the "CONSTRAINT" keyword inline with "CREATE TABLE". There are some cases where this is undesireable, particularly when two tables reference each other mutually, each with a foreign key referencing the other. In such a situation at least one of the foreign key constraints must be generated after both tables have been built. To support such a scheme, `ForeignKey` and `ForeignKeyConstraint` offer the flag `use_alter=True`. When using this flag, the constraint will be generated using a definition similar to "ALTER

TABLE <tablename> ADD CONSTRAINT <name> ...". Since a name is required, the `name` attribute must also be specified. For example:

```
node = Table('node', meta,
    Column('node_id', Integer, primary_key=True),
    Column('primary_element', Integer,
        ForeignKey('element.element_id', use_alter=True, name='fk_node_element_id')
    )
)

element = Table('element', meta,
    Column('element_id', Integer, primary_key=True),
    Column('parent_node_id', Integer),
    ForeignKeyConstraint(
        ['parent_node_id'],
        ['node.node_id'],
        use_alter=True,
        name='fk_element_parent_node_id'
    )
)
```

ON UPDATE and ON DELETE

Most databases support *cascading* of foreign key values, that is the when a parent row is updated the new value is placed in child rows, or when the parent row is deleted all corresponding child rows are set to null or deleted. In data definition language these are specified using phrases like "ON UPDATE CASCADE", "ON DELETE CASCADE", and "ON DELETE SET NULL", corresponding to foreign key constraints. The phrase after "ON UPDATE" or "ON DELETE" may also other allow other phrases that are specific to the database in use. The `ForeignKey` and `ForeignKeyConstraint` objects support the generation of this clause via the `onupdate` and `ondelete` keyword arguments. The value is any string which will be output after the appropriate "ON UPDATE" or "ON DELETE" phrase:

```
child = Table('child', meta,
    Column('id', Integer,
            ForeignKey('parent.id', onupdate="CASCADE", ondelete="CASCADE"),
            primary_key=True
    )
)

composite = Table('composite', meta,
    Column('id', Integer, primary_key=True),
    Column('rev_id', Integer),
    Column('note_id', Integer),
    ForeignKeyConstraint(
                ['rev_id', 'note_id'],
                ['revisions.id', 'revisions.note_id'],
                onupdate="CASCADE", ondelete="SET NULL"
    )
)
```

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.

Foreign Key API Constructs

class `sqlalchemy.schema.`**`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 dependency 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"))
)
```

Note that `ForeignKey` is only a marker object that defines a dependency between two columns. The actual constraint is in all cases represented by the `ForeignKeyConstraint` object. This object will be generated automatically when a `ForeignKey` is associated with a `Column` which in turn is associated with a `Table`. Conversely, when `ForeignKeyConstraint` is applied to a `Table`, `ForeignKey` markers are automatically generated to be present on each associated `Column`, which are also associated with the constraint object.

Note that you cannot define a "composite" foreign key constraint, that is a constraint between a grouping of multiple parent/child columns, using `ForeignKey` objects. To define this grouping, the `ForeignKeyConstraint` object must be used, and applied to the `Table`. The associated `ForeignKey` objects are created automatically.

The `ForeignKey` objects associated with an individual `Column` object are available in the *foreign_keys* collection of that column.

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.

The `ForeignKey` object when constructed generates a `ForeignKeyConstraint` which is associated with the parent `Table` object's collection of constraints.

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.
- **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** – passed to the underlying `ForeignKeyConstraint` to indicate the constraint should be generated/dropped externally from the CREATE TABLE/ DROP TABLE statement. See that classes' constructor for details.

column
Return the target `Column` referenced by this `ForeignKey`.

If this `ForeignKey` was created using a string-based target column specification, this attribute will on first access initiate a resolution process to locate the referenced remote `Column`. The resolution process traverses to the parent `Column`, `Table`, and `MetaData` to proceed - if any of these aren't yet present, an error is raised.

copy (*schema=None*)
Produce a copy of this `ForeignKey` object.

The new `ForeignKey` will not be bound to any `Column`.

This method is usually used by the internal copy procedures of `Column`, `Table`, and `MetaData`.

Parameters

- **schema** – The returned `ForeignKey` will reference the original table and column name, qualified by the given string schema name.

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
Return a string based 'column specification' for this `ForeignKey`.

This is usually the equivalent of the string-based "tablename.colname" argument first passed to the object's constructor.

class `sqlalchemy.schema.`**ForeignKeyConstraint** (*columns, refcolumns, name=None, onupdate=None, ondelete=None, deferrable=None, initially=None, use_alter=False, link_to_name=False, table=None*)

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, deferrable=None, initially=None, use_alter=False, link_to_name=False, table=None*)
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 the DDL for this constraint as part of the CREATE TABLE definition. Instead, generate it via an ALTER TABLE statement issued after the full collection of tables have been created, and drop it via an ALTER TABLE statement before the full collection of tables are dropped. This is shorthand for the usage of `AddConstraint` and `DropConstraint` applied as "after-create" and "before-drop" events on the MetaData object. This is normally used to generate/drop constraints on objects that are mutually dependent on each other.

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')
    )
```

class `sqlalchemy.schema.`**`UniqueConstraint`**`(`**columns*, ***kw*`)`

Bases: `sqlalchemy.schema.ColumnCollectionConstraint`

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.

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.

```
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')
    )

mytable.create(engine)
CREATE TABLE mytable (
    col1 INTEGER  CHECK (col1>5),
    col2 INTEGER,
    col3 INTEGER,
    CONSTRAINT check1  CHECK (col2 > col3 + 5)
)
```

class `sqlalchemy.schema.`**`CheckConstraint`**(*sqltext*, *name=None*, *deferrable=None*, *initially=None*, *table=None*, *_create_rule=None*)

Bases: `sqlalchemy.schema.Constraint`

A table- or column-level CHECK constraint.

Can be included in the definition of a Table or Column.

Other Constraint Classes

class `sqlalchemy.schema.`**`Constraint`**(*name=None*, *deferrable=None*, *initially=None*, *_create_rule=None*)

Bases: `sqlalchemy.schema.SchemaItem`

A table-level SQL constraint.

class `sqlalchemy.schema.`**`ColumnCollectionConstraint`**(**columns*, ***kw*)

Bases: `sqlalchemy.schema.Constraint`

A constraint that proxies a ColumnCollection.

class `sqlalchemy.schema.`**`PrimaryKeyConstraint`**(**columns*, ***kw*)

Bases: `sqlalchemy.schema.ColumnCollectionConstraint`

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.

Indexes

Indexes can be created anonymously (using an auto-generated name `ix_<column label>`) 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.

Below we illustrate a `Table` with several `Index` objects associated. The DDL for "CREATE INDEX" is issued right after the create statements for the table:

```
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)

mytable.create(engine)
CREATE TABLE mytable (
    col1 INTEGER,
    col2 INTEGER,
    col3 INTEGER,
    col4 INTEGER,
    col5 INTEGER,
    col6 INTEGER
)
CREATE INDEX ix_mytable_col1 ON mytable (col1)
CREATE UNIQUE INDEX ix_mytable_col2 ON mytable (col2)
CREATE UNIQUE INDEX myindex ON mytable (col5, col6)
CREATE INDEX idx_col34 ON mytable (col3, col4)
```

The `Index` object also supports its own `create()` method:

```
i = Index('someindex', mytable.c.col5)
i.create(engine)
CREATE INDEX someindex ON mytable (col5)
```

class `sqlalchemy.schema.`**`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.

3.6.5 Customizing DDL

In the preceding sections we've discussed a variety of schema constructs including `Table`, `ForeignKeyConstraint`, `CheckConstraint`, and `Sequence`. Throughout, we've relied upon the `create()` and `create_all()` methods of `Table` and `MetaData` in order to issue data definition language (DDL) for all constructs. When issued, a pre-determined order of operations is invoked, and DDL to create each table is created unconditionally including all constraints and other objects associated with it. For more complex scenarios where database-specific DDL is required, SQLAlchemy offers two techniques which can be used to add any DDL based on any condition, either accompanying the standard generation of tables or by itself.

Controlling DDL Sequences

The `sqlalchemy.schema` package contains SQL expression constructs that provide DDL expressions. For example, to produce a `CREATE TABLE` statement:

```
from sqlalchemy.schema import CreateTable
engine.execute(CreateTable(mytable))
CREATE TABLE mytable (
    col1 INTEGER,
    col2 INTEGER,
    col3 INTEGER,
    col4 INTEGER,
    col5 INTEGER,
    col6 INTEGER
)
```

Above, the `CreateTable` construct works like any other expression construct (such as `select()`, `table.insert()`, etc.). A full reference of available constructs is in *DDL API*.

The DDL constructs all extend a common base class which provides the capability to be associated with an individual `Table` or `MetaData` object, to be invoked upon create/drop events. Consider the example of a table which contains a CHECK constraint:

```
users = Table('users', metadata,
               Column('user_id', Integer, primary_key=True),
               Column('user_name', String(40), nullable=False),
               CheckConstraint('length(user_name) >= 8',name="cst_user_name_length")
               )

users.create(engine)
CREATE TABLE users (
    user_id SERIAL NOT NULL,
    user_name VARCHAR(40) NOT NULL,
    PRIMARY KEY (user_id),
    CONSTRAINT cst_user_name_length  CHECK (length(user_name) >= 8)
)
```

The above table contains a column "user_name" which is subject to a CHECK constraint that validates that the length of the string is at least eight characters. When a `create()` is issued for this table, DDL for the `CheckConstraint` will also be issued inline within the table definition.

The CheckConstraint construct can also be constructed externally and associated with the Table afterwards:

```
constraint = CheckConstraint('length(user_name) >= 8',name="cst_user_name_length")
users.append_constraint(constraint)
```

So far, the effect is the same. However, if we create DDL elements corresponding to the creation and removal of this constraint, and associate them with the Table as events, these new events will take over the job of issuing DDL for the constraint. Additionally, the constraint will be added via ALTER:

```
AddConstraint(constraint).execute_at("after-create", users)
DropConstraint(constraint).execute_at("before-drop", users)

users.create(engine)
CREATE TABLE users (
    user_id SERIAL NOT NULL,
    user_name VARCHAR(40) NOT NULL,
    PRIMARY KEY (user_id)
)

ALTER TABLE users ADD CONSTRAINT cst_user_name_length  CHECK (length(user_name) >= 8)
users.drop(engine)
ALTER TABLE users DROP CONSTRAINT cst_user_name_length
DROP TABLE user
```

The real usefulness of the above becomes clearer once we illustrate the on attribute of a DDL event. The on parameter is part of the constructor, and may be a string name of a database dialect name, a tuple containing dialect names, or a Python callable. This will limit the execution of the item to just those dialects, or when the return value of the callable is True. So if our CheckConstraint was only supported by Postgresql and not other databases, we could limit it to just that dialect:

```
AddConstraint(constraint, on='postgresql').execute_at("after-create", users)
DropConstraint(constraint, on='postgresql').execute_at("before-drop", users)
```

Or to any set of dialects:

```
AddConstraint(constraint, on=('postgresql', 'mysql')).execute_at("after-create", users)
DropConstraint(constraint, on=('postgresql', 'mysql')).execute_at("before-drop", users)
```

When using a callable, the callable is passed the ddl element, event name, the Table or MetaData object whose "create" or "drop" event is in progress, and the Connection object being used for the operation, as well as additional information as keyword arguments. The callable can perform checks, such as whether or not a given item already exists. Below we define should_create() and should_drop() callables that check for the presence of our named constraint:

```
def should_create(ddl, event, target, connection, **kw):
    row = connection.execute("select conname from pg_constraint where conname='%s'" % ddl.e
    return not bool(row)

def should_drop(ddl, event, target, connection, **kw):
    return not should_create(ddl, event, target, connection, **kw)

AddConstraint(constraint, on=should_create).execute_at("after-create", users)
DropConstraint(constraint, on=should_drop).execute_at("before-drop", users)

users.create(engine)
CREATE TABLE users (
    user_id SERIAL NOT NULL,
    user_name VARCHAR(40) NOT NULL,
```

```
    PRIMARY KEY (user_id)
)

select conname from pg_constraint where conname='cst_user_name_length'
ALTER TABLE users ADD CONSTRAINT cst_user_name_length  CHECK (length(user_name) >= 8)
users.drop(engine)
select conname from pg_constraint where conname='cst_user_name_length'
ALTER TABLE users DROP CONSTRAINT cst_user_name_length
DROP TABLE user
```

Custom DDL

Custom DDL phrases are most easily achieved using the `DDL` construct. This construct works like all the other DDL elements except it accepts a string which is the text to be emitted:

```
DDL("ALTER TABLE users ADD CONSTRAINT "
    "cst_user_name_length "
    " CHECK (length(user_name) >= 8)").execute_at("after-create", metadata)
```

A more comprehensive method of creating libraries of DDL constructs is to use custom compilation - see *Custom SQL Constructs and Compilation Extension* for details.

DDL API

class `sqlalchemy.schema.`**`DDLElement`**

Bases: `sqlalchemy.sql.expression.Executable`, `sqlalchemy.sql.expression.ClauseElement`

Base class for DDL expression constructs.

`against`(*target*)

Return a copy of this DDL against a specific schema item.

`bind`

`execute`(*bind=None*, *target=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.

Parameters

- **bind** – Optional, an `Engine` or `Connection`. If not supplied, a valid `Connectable` must be present in the `.bind` property.
- **target** – Optional, defaults to None. The target SchemaItem for the execute call. Will be passed to the `on` callable if any, and may also provide string expansion data for the statement. See `execute_at` for more information.

`execute_at`(*event*, *target*)

Link execution of this DDL to the DDL lifecycle of a SchemaItem.

Links this `DDLElement` 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.

Parameters

- **event** – One of the events defined in the schema item's `.ddl_events`; e.g. 'before-create', 'after-create', 'before-drop' or 'after-drop'
- **target** – The Table or MetaData instance for which this DDLElement will be associated with.

A DDLElement instance can be linked to any number of schema items.

`execute_at` builds on the `append_ddl_listener` interface of `MetaData` 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.

class `sqlalchemy.schema.`**`DDL`**(*statement*, *on=None*, *context=None*, *bind=None*)

Bases: `sqlalchemy.schema.DDLElement`

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)
```

When operating on Table events, the following `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.

`__init__`(*statement*, *on=None*, *context=None*, *bind=None*)

Create a DDL statement.

Parameters

- **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, tuple or a callable predicate. If a string, it will be compared to the name of the executing database dialect:

  ```
  DDL('something', on='postgresql')
  ```

  If a tuple, specifies multiple dialect names:

```
DDL('something', on=('postgresql', 'mysql'))
```

If a callable, it will be invoked with four positional arguments as well as optional keyword arguments:

ddl This DDL element.

event The name of the event that has triggered this DDL, such as 'after-create' Will be None if the DDL is executed explicitly.

target The `Table` or `MetaData` object which is the target of this event. May be None if the DDL is executed explicitly.

connection The `Connection` being used for DDL execution

tables Optional keyword argument - a list of Table objects which are to be created/ dropped within a MetaData.create_all() or drop_all() method call.

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.

class `sqlalchemy.schema.`**`CreateTable`**(*element*, *on=None*, *bind=None*)
Bases: `sqlalchemy.schema._CreateDropBase`

Represent a CREATE TABLE statement.

class `sqlalchemy.schema.`**`DropTable`**(*element*, *on=None*, *bind=None*)
Bases: `sqlalchemy.schema._CreateDropBase`

Represent a DROP TABLE statement.

class `sqlalchemy.schema.`**`CreateSequence`**(*element*, *on=None*, *bind=None*)
Bases: `sqlalchemy.schema._CreateDropBase`

Represent a CREATE SEQUENCE statement.

class `sqlalchemy.schema.`**`DropSequence`**(*element*, *on=None*, *bind=None*)
Bases: `sqlalchemy.schema._CreateDropBase`

Represent a DROP SEQUENCE statement.

class `sqlalchemy.schema.`**`CreateIndex`**(*element*, *on=None*, *bind=None*)
Bases: `sqlalchemy.schema._CreateDropBase`

Represent a CREATE INDEX statement.

class `sqlalchemy.schema.`**`DropIndex`**(*element*, *on=None*, *bind=None*)
Bases: `sqlalchemy.schema._CreateDropBase`

Represent a DROP INDEX statement.

class `sqlalchemy.schema.`**`AddConstraint`**(*element*, **args*, ***kw*)
Bases: `sqlalchemy.schema._CreateDropBase`

Represent an ALTER TABLE ADD CONSTRAINT statement.

`__init__`(*element*, **args*, ***kw*)

class sqlalchemy.schema.**DropConstraint** (*element*, *cascade=False*, ***kw*)
Bases: sqlalchemy.schema._CreateDropBase

Represent an ALTER TABLE DROP CONSTRAINT statement.

__init__ (*element*, *cascade=False*, ***kw*)

3.7 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.

3.7.1 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 sqlalchemy.types.**BigInteger** (**args*, ***kwargs*)
Bases: sqlalchemy.types.Integer

A type for bigger int integers.

Typically generates a BIGINT in DDL, and otherwise acts like a normal Integer on the Python side.

class sqlalchemy.types.**Boolean** (*create_constraint=True*, *name=None*)
Bases: sqlalchemy.types.TypeEngine, sqlalchemy.types.SchemaType

A bool datatype.

Boolean typically uses BOOLEAN or SMALLINT on the DDL side, and on the Python side deals in True or False.

__init__ (*create_constraint=True*, *name=None*)
Construct a Boolean.

Parameters

- **create_constraint** – defaults to True. If the boolean is generated as an int/smallint, also create a CHECK constraint on the table that ensures 1 or 0 as a value.
- **name** – if a CHECK constraint is generated, specify the name of the constraint.

class `sqlalchemy.types.Date(*args, **kwargs)`
Bases: `sqlalchemy.types._DateAffinity`, `sqlalchemy.types.TypeEngine`

A type for `datetime.date()` objects.

class `sqlalchemy.types.DateTime(timezone=False)`
Bases: `sqlalchemy.types._DateAffinity`, `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.

`__init__(timezone=False)`

class `sqlalchemy.types.Enum(*enums, **kw)`
Bases: `sqlalchemy.types.String`, `sqlalchemy.types.SchemaType`

Generic Enum Type.

The Enum type provides a set of possible string values which the column is constrained towards.

By default, uses the backend's native ENUM type if available, else uses VARCHAR + a CHECK constraint.

`__init__(*enums, **kw)`
Construct an enum.

Keyword arguments which don't apply to a specific backend are ignored by that backend.

Parameters

- ***enums** – string or unicode enumeration labels. If unicode labels are present, the *convert_unicode* flag is auto-enabled.
- **convert_unicode** – Enable unicode-aware bind parameter and result-set processing for this Enum's data. This is set automatically based on the presence of unicode label strings.
- **metadata** – Associate this type directly with a `MetaData` object. For types that exist on the target database as an independent schema construct (Postgresql), this type will be created and dropped within `create_all()` and `drop_all()` operations. If the type is not associated with any `MetaData` object, it will associate itself with each `Table` in which it is used, and will be created when any of those individual tables are created, after a check is performed for it's existence. The type is only dropped when `drop_all()` is called for that `Table` object's metadata, however.
- **name** – The name of this type. This is required for Postgresql and any future supported database which requires an explicitly named type, or an explicitly named constraint in order to generate the type and/or a table that uses it.
- **native_enum** – Use the database's native ENUM type when available. Defaults to True. When False, uses VARCHAR + check constraint for all backends.
- **schema** – Schemaname of this type. For types that exist on the target database as an independent schema construct (Postgresql), this parameter specifies the named schema in which the type is present.
- **quote** – Force quoting to be on or off on the type's name. If left as the default of *None*, the usual schema-level "case sensitive"/"reserved name" rules are used to determine if this type's name should be quoted.

class `sqlalchemy.types.`**`Float`**`(`*precision=None*, *asdecimal=False*, ***kwargs*`)`
Bases: `sqlalchemy.types.Numeric`

A type for `float` numbers.

Returns Python `float` objects by default, applying conversion as needed.

`__init__``(`*precision=None*, *asdecimal=False*, ***kwargs*`)`
Construct a Float.

Parameters

- **precision** – the numeric precision for use in DDL `CREATE TABLE`.
- **asdecimal** – the same flag as that of `Numeric`, but defaults to `False`. Note that setting this flag to `True` results in floating point conversion.

class `sqlalchemy.types.`**`Integer`**`(`**args*, ***kwargs*`)`
Bases: `sqlalchemy.types._DateAffinity`, `sqlalchemy.types.TypeEngine`

A type for `int` integers.

class `sqlalchemy.types.`**`Interval`**`(`*native=True*, *second_precision=None*, *day_precision=None*`)`
Bases: `sqlalchemy.types._DateAffinity`, `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).

Note that the `Interval` type does not currently provide date arithmetic operations on platforms which do not support interval types natively. Such operations usually require transformation of both sides of the expression (such as, conversion of both sides into integer epoch values first) which currently is a manual procedure (such as via `func`).

`__init__``(`*native=True*, *second_precision=None*, *day_precision=None*`)`
Construct an Interval object.

Parameters

- **native** – when True, use the actual INTERVAL type provided by the database, if supported (currently Postgresql, Oracle). Otherwise, represent the interval data as an epoch value regardless.
- **second_precision** – For native interval types which support a "fractional seconds precision" parameter, i.e. Oracle and Postgresql
- **day_precision** – for native interval types which support a "day precision" parameter, i.e. Oracle.

`impl`
alias of `DateTime`

class `sqlalchemy.types.`**`LargeBinary`**`(`*length=None*`)`
Bases: `sqlalchemy.types._Binary`

A type for large 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.

`__init__``(`*length=None*`)`
Construct a LargeBinary type.

Parameters

- **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a small BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class `sqlalchemy.types.`**`Numeric`**(*precision=None*, *scale=None*, *asdecimal=True*)
Bases: `sqlalchemy.types._DateAffinity`, `sqlalchemy.types.TypeEngine`

A type for fixed precision numbers.

Typically generates DECIMAL or NUMERIC. Returns `decimal.Decimal` objects by default, applying conversion as needed.

`__init__`(*precision=None*, *scale=None*, *asdecimal=True*)
Construct a Numeric.

Parameters

- **precision** – the numeric precision for use in DDL `CREATE TABLE`.
- **scale** – the numeric scale for use in DDL `CREATE TABLE`.
- **asdecimal** – default True. Return whether or not values should be sent as Python Decimal objects, or as floats. Different DBAPIs send one or the other based on datatypes - the Numeric type will ensure that return values are one or the other across DBAPIs consistently.

When using the `Numeric` type, care should be taken to ensure that the asdecimal setting is apppropriate for the DBAPI in use - when Numeric applies a conversion from Decimal->float or float-> Decimal, this conversion incurs an additional performance overhead for all result columns received.

DBAPIs that return Decimal natively (e.g. psycopg2) will have better accuracy and higher performance with a setting of `True`, as the native translation to Decimal reduces the amount of floating- point issues at play, and the Numeric type itself doesn't need to apply any further conversions. However, another DBAPI which returns floats natively *will* incur an additional conversion overhead, and is still subject to floating point data loss - in which case `asdecimal=False` will at least remove the extra conversion overhead.

class `sqlalchemy.types.`**`PickleType`**(*protocol=2*, *pickler=None*, *mutable=True*, *comparator=None*)
Bases: `sqlalchemy.types.MutableType`, `sqlalchemy.types.TypeDecorator`

Holds Python objects, which are serialized using pickle.

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.

Note: be sure to read the notes for `MutableType` regarding ORM performance implications.

`__init__`(*protocol=2*, *pickler=None*, *mutable=True*, *comparator=None*)
Construct a PickleType.

Parameters

- **protocol** – defaults to `pickle.HIGHEST_PROTOCOL`.
- **pickler** – defaults to cPickle.pickle or pickle.pickle if cPickle is not available. May be any object with pickle-compatible `dumps' and ''loads` methods.
- **mutable** – defaults to True; implements `AbstractType.is_mutable()`. When `True`, incoming objects should provide an `__eq__()` method which per-

forms the desired deep comparison of members, or the `comparator` argument must be present.

- **comparator** – optional. a 2-arg callable predicate used to compare values of this type. Otherwise, the == operator is used to compare values.

impl
alias of `LargeBinary`

is_mutable()
Return True if the target Python type is 'mutable'.

When this method is overridden, `copy_value()` should also be supplied. The `MutableType` mixin is recommended as a helper.

class sqlalchemy.types.**SchemaType**(***kw*)
Bases: `object`

Mark a type as possibly requiring schema-level DDL for usage.

Supports types that must be explicitly created/dropped (i.e. PG ENUM type) as well as types that are complimented by table or schema level constraints, triggers, and other rules.

__init__(***kw*)

bind

create(*bind=None*, *checkfirst=False*)
Issue CREATE ddl for this type, if applicable.

drop(*bind=None*, *checkfirst=False*)
Issue DROP ddl for this type, if applicable.

class sqlalchemy.types.**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 sqlalchemy.types.**String**(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)
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.

__init__(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)
Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL statements. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued. Whether the value is interpreted as bytes or characters is database specific.

- **convert_unicode** – defaults to False. If True, the type will do what is necessary in order to accept Python Unicode objects as bind parameters, and to return Python Unicode objects in result rows. This may require SQLAlchemy to explicitly coerce incoming Python unicodes into an encoding, and from an encoding back to Unicode, or it may not require any interaction from SQLAlchemy at all, depending on the DBAPI in use.

  When SQLAlchemy performs the encoding/decoding, the encoding used is configured via `encoding`, which defaults to *utf-8*.

  The "convert_unicode" behavior can also be turned on for all String types by setting `sqlalchemy.engine.base.Dialect.convert_unicode` on create_engine().

  To instruct SQLAlchemy to perform Unicode encoding/decoding even on a platform that already handles Unicode natively, set convert_unicode='force'. This will incur significant performance overhead when fetching unicode result columns.

- **assert_unicode** – Deprecated. A warning is raised in all cases when a non-Unicode object is passed when SQLAlchemy would coerce into an encoding (note: but **not** when the DBAPI handles unicode objects natively). To suppress or raise this warning to an error, use the Python warnings filter documented at: http://docs.python.org/library/warnings.html

- **unicode_error** – Optional, a method to use to handle Unicode conversion errors. Behaves like the 'errors' keyword argument to the standard library's string.decode() functions. This flag requires that *convert_unicode* is set to *"force"* - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as an absolute last resort for reading strings from a column with varied or corrupted encodings, which only applies to databases that accept invalid encodings in the first place (i.e. MySQL. *not* PG, Sqlite, etc.)

class `sqlalchemy.types.`**`Text`**(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)

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 `sqlalchemy.types.`**`Time`**(*timezone=False*)

Bases: `sqlalchemy.types._DateAffinity`, `sqlalchemy.types.TypeEngine`

A type for `datetime.time()` objects.

`__init__`(*timezone=False*)

class `sqlalchemy.types.`**`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.

It's roughly equivalent to using a `String` object with `convert_unicode=True`, however the type has other significances in that it implies the usage of a unicode-capable type being used on the backend, such as

NVARCHAR. This may affect what type is emitted when issuing CREATE TABLE and also may effect some DBAPI-specific details, such as type information passed along to `setinputsizes()`.

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*.

`__init__`(*length=None*, ***kwargs*)
Create a Unicode-converting String type.

Parameters

- **length** – optional, a length for the column for use in DDL statements. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued. Whether the value is interpreted as bytes or characters is database specific.
- ****kwargs** – passed through to the underlying `String` type.

class `sqlalchemy.types.`**`UnicodeText`**(*length=None*, ***kwargs*)
Bases: `sqlalchemy.types.Text`

An unbounded-length Unicode string.

See `Unicode` for details on the unicode behavior of this object.

Like `Unicode`, usage the `UnicodeText` type implies a unicode-capable type being used on the backend, such as NCLOB.

`__init__`(*length=None*, ***kwargs*)
Create a Unicode-converting Text type.

Parameters

- **length** – optional, a length for the column for use in DDL statements. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued. Whether the value is interpreted as bytes or characters is database specific.

3.7.2 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 `sqlalchemy.types.`**`BIGINT`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.BigInteger`

The SQL BIGINT type.

class `sqlalchemy.types.`**`BINARY`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

The SQL BINARY type.

class `sqlalchemy.types.`**`BLOB`**(*length=None*)
Bases: `sqlalchemy.types.LargeBinary`

The SQL BLOB type.

class `sqlalchemy.types.`**`BOOLEAN`**(*create_constraint=True*, *name=None*)
Bases: `sqlalchemy.types.Boolean`

The SQL BOOLEAN type.

class `sqlalchemy.types.`**`CHAR`**(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: `sqlalchemy.types.String`

The SQL CHAR type.

class `sqlalchemy.types.`**`CLOB`**(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: `sqlalchemy.types.Text`

The CLOB type.

This type is found in Oracle and Informix.

class `sqlalchemy.types.`**`DATE`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.Date`

The SQL DATE type.

class `sqlalchemy.types.`**`DATETIME`**(*timezone=False*)
Bases: `sqlalchemy.types.DateTime`

The SQL DATETIME type.

class `sqlalchemy.types.`**`DECIMAL`**(*precision=None*, *scale=None*, *asdecimal=True*)
Bases: `sqlalchemy.types.Numeric`

The SQL DECIMAL type.

class `sqlalchemy.types.`**`FLOAT`**(*precision=None*, *asdecimal=False*, ***kwargs*)
Bases: `sqlalchemy.types.Float`

The SQL FLOAT type.

`sqlalchemy.types.`**`INT`**
alias of `INTEGER`

class `sqlalchemy.types.`**`INTEGER`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.Integer`

The SQL INT or INTEGER type.

class `sqlalchemy.types.`**`NCHAR`**(*length=None*, ***kwargs*)
Bases: `sqlalchemy.types.Unicode`

The SQL NCHAR type.

class `sqlalchemy.types.`**`NVARCHAR`**(*length=None*, ***kwargs*)
Bases: `sqlalchemy.types.Unicode`

The SQL NVARCHAR type.

class `sqlalchemy.types.`**`NUMERIC`**(*precision=None*, *scale=None*, *asdecimal=True*)
Bases: `sqlalchemy.types.Numeric`

The SQL NUMERIC type.

class `sqlalchemy.types.SMALLINT`(**args*, ***kwargs*)
Bases: `sqlalchemy.types.SmallInteger`

The SQL SMALLINT type.

class `sqlalchemy.types.TEXT`(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: `sqlalchemy.types.Text`

The SQL TEXT type.

class `sqlalchemy.types.TIME`(*timezone=False*)
Bases: `sqlalchemy.types.Time`

The SQL TIME type.

class `sqlalchemy.types.TIMESTAMP`(*timezone=False*)
Bases: `sqlalchemy.types.DateTime`

The SQL TIMESTAMP type.

class `sqlalchemy.types.VARBINARY`(*length=None*)
Bases: `sqlalchemy.types._Binary`

The SQL VARBINARY type.

class `sqlalchemy.types.VARCHAR`(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: `sqlalchemy.types.String`

The SQL VARCHAR type.

3.7.3 Vendor-Specific Types

Database-specific types are also available for import from each database's dialect module. See the *sqlalchemy.dialects_toplevel* 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.dialects import mysql

table = Table('foo', meta,
    Column('id', mysql.BIGINTEGER),
    Column('enumerates', mysql.ENUM('a', 'b', 'c'))
)
```

Or some PostgreSQL types:

```
from sqlalchemy.dialects import postgresql

table = Table('foo', meta,
    Column('ipaddress', postgresql.INET),
    Column('elements', postgresql.ARRAY(str))
    )
```

Each dialect provides the full set of typenames supported by that backend within its *__all__* collection, so that a simple *import* * or similar will import all supported types as implemented for that backend:

```
from sqlalchemy.dialects.postgresql import *

t = Table('mytable', metadata,
```

```
            Column('id', INTEGER, primary_key=True),
            Column('name', VARCHAR(300)),
            Column('inetaddr', INET)
)
```

Where above, the INTEGER and VARCHAR types are ultimately from sqlalchemy.types, and INET is specific to the Postgresql dialect.

Some dialect level types have the same name as the SQL standard type, but also provide additional arguments. For example, MySQL implements the full range of character and string types including additional arguments such as *collation* and *charset*:

```
from sqlalchemy.dialects.mysql import VARCHAR, TEXT

table = Table('foo', meta,
    Column('col1', VARCHAR(200, collation='binary')),
    Column('col2', TEXT(charset='latin1'))
)
```

3.7.4 Custom Types

A variety of methods exist to redefine the behavior of existing types as well as to provide new ones.

Overriding Type Compilation

The string produced by any type object, when rendered in a CREATE TABLE statement or other SQL function like CAST, can be changed. See the section *Changing Compilation of Types*, a subsection of *Custom SQL Constructs and Compilation Extension*, for a short example.

Augmenting Existing Types

The `TypeDecorator` allows the creation of custom types which add bind-parameter and result-processing behavior to an existing type object. It is used when additional in-Python marshalling of data to and from the database is required.

class `sqlalchemy.types.`**`TypeDecorator`**`(*args, **kwargs)`

Bases: `sqlalchemy.types.AbstractType`

Allows the creation of types which add additional functionality to an existing type.

This method is preferred to direct subclassing of SQLAlchemy's built-in types as it ensures that all required functionality of the underlying type is kept in place.

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.

Types that receive a Python type that isn't similar to the ultimate type used may want to define the `TypeDecorator.coerce_compared_value()` method. This is used to give the expression system a hint when coercing Python objects into bind parameters within expressions. Consider this expression:

```
mytable.c.somecol + datetime.date(2009, 5, 15)
```

Above, if "somecol" is an `Integer` variant, it makes sense that we're doing date arithmetic, where above is usually interpreted by databases as adding a number of days to the given date. The expression system does the right thing by not attempting to coerce the "date()" value into an integer-oriented bind parameter.

However, in the case of `TypeDecorator`, we are usually changing an incoming Python type to something new - `TypeDecorator` by default will "coerce" the non-typed side to be the same type as itself. Such as below, we define an "epoch" type that stores a date value as an integer:

```
class MyEpochType(types.TypeDecorator):
    impl = types.Integer

    epoch = datetime.date(1970, 1, 1)

    def process_bind_param(self, value, dialect):
        return (value - self.epoch).days

    def process_result_value(self, value, dialect):
        return self.epoch + timedelta(days=value)
```

Our expression of `somecol + date` with the above type will coerce the "date" on the right side to also be treated as `MyEpochType`.

This behavior can be overridden via the `coerce_compared_value()` method, which returns a type that should be used for the value of the expression. Below we set it such that an integer value will be treated as an `Integer`, and any other value is assumed to be a date and will be treated as a `MyEpochType`:

```
def coerce_compared_value(self, op, value):
    if isinstance(value, int):
        return Integer()
    else:
        return self
```

`__init__`(**args*, ***kwargs*)

`adapt`(*cls*)

`bind_processor`(*dialect*)

`coerce_compared_value`(*op*, *value*)

Suggest a type for a 'coerced' Python value in an expression.

By default, returns self. This method is called by the expression system when an object using this type is on the left or right side of an expression against a plain Python object which does not yet have a SQLAlchemy type assigned:

```
expr = table.c.somecolumn + 35
```

Where above, if `somecolumn` uses this type, this method will be called with the value `operator.add` and `35`. The return value is whatever SQLAlchemy type should be used for `35` for this particular operation.

compare_values(*x, y*)

compile(*dialect=None*)

copy()

copy_value(*value*)

dialect_impl(*dialect*)

get_dbapi_type(*dbapi*)

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. Values such as dicts, lists which are serialized into strings are examples of "mutable" column structures.

When this method is overridden, `copy_value()` should also be supplied. The `MutableType` mixin is recommended as a helper.

load_dialect_impl(*dialect*)
User hook which can be overridden to provide a different 'impl' type per-dialect.

by default returns self.impl.

process_bind_param(*value, dialect*)

process_result_value(*value, dialect*)

result_processor(*dialect, coltype*)

type_engine(*dialect*)
Return a TypeEngine instance for this TypeDecorator.

A few key `TypeDecorator` recipes follow.

Rounding Numerics

Some database connectors like those of SQL Server choke if a Decimal is passed with too many decimal places. Here's a recipe that rounds them down:

```
from sqlalchemy.types import TypeDecorator, Numeric
from decimal import Decimal

class SafeNumeric(TypeDecorator):
    """Adds quantization to Numeric."""

    impl = Numeric

    def __init__(self, *arg, **kw):
        TypeDecorator.__init__(self, *arg, **kw)
        self.quantize_int = -(self.impl.precision - self.impl.scale)
        self.quantize = Decimal(10) ** self.quantize_int
```

```
    def process_bind_param(self, value, dialect):
        if isinstance(value, Decimal) and \
            value.as_tuple()[2] < self.quantize_int:
            value = value.quantize(self.quantize)
        return value
```

Backend-agnostic GUID Type

Receives and returns Python uuid() objects. Uses the PG UUID type when using Postgresql, CHAR(32) on other backends, storing them in stringified hex format. Can be modified to store binary in CHAR(16) if desired:

```
from sqlalchemy.types import TypeDecorator, CHAR
from sqlalchemy.dialects.postgresql import UUID
import uuid

class GUID(TypeDecorator):
    """Platform-independent GUID type.

    Uses Postgresql's UUID type, otherwise uses
    CHAR(32), storing as stringified hex values.

    """
    impl = CHAR

    def load_dialect_impl(self, dialect):
        if dialect.name == 'postgresql':
            return dialect.type_descriptor(UUID())
        else:
            return dialect.type_descriptor(CHAR(32))

    def process_bind_param(self, value, dialect):
        if value is None:
            return value
        elif dialect.name == 'postgresql':
            return str(value)
        else:
            if not isinstance(value, uuid.UUID):
                return "%.32x" % uuid.UUID(value)
            else:
                # hexstring
                return "%.32x" % value

    def process_result_value(self, value, dialect):
        if value is None:
            return value
        else:
            return uuid.UUID(value)
```

Marshal JSON Strings

This type uses `simplejson` to marshal Python data structures to/from JSON. Can be modified to use Python's builtin json encoder.

Note that the base type is not "mutable", meaning in-place changes to the value will not be detected by the ORM - you instead would need to replace the existing value with a new one to detect changes. The subtype `MutableJSONEncodedDict` adds "mutability" to allow this, but note that "mutable" types add a significant performance penalty to the ORM's flush process:

```
from sqlalchemy.types import TypeDecorator, MutableType, VARCHAR
import simplejson

class JSONEncodedDict(TypeDecorator):
    """Represents an immutable structure as a json-encoded string.

    Usage::

        JSONEncodedDict(255)

    """

    impl = VARCHAR

    def process_bind_param(self, value, dialect):
        if value is not None:
            value = simplejson.dumps(value, use_decimal=True)

        return value

    def process_result_value(self, value, dialect):
        if value is not None:
            value = simplejson.loads(value, use_decimal=True)
        return value

class MutableJSONEncodedDict(MutableType, JSONEncodedDict):
    """Adds mutability to JSONEncodedDict."""

    def copy_value(self, value):
        return simplejson.loads(
                    simplejson.dumps(value, use_decimal=True),
                    use_decimal=True)
```

Creating New Types

The `UserDefinedType` class is provided as a simple base class for defining entirely new database types:

class `sqlalchemy.types.`**`UserDefinedType`**`(*args, **kwargs)`

Bases: `sqlalchemy.types.TypeEngine`

Base for user defined types.

This should be the base of new types. Note that for most cases, `TypeDecorator` is probably more appropriate:

```
import sqlalchemy.types as types
```

```
class MyType(types.UserDefinedType):
    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, coltype):
        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*)
A hook which allows the given operator to be adapted to something new.

See also UserDefinedType._adapt_expression(), an as-yet- semi-public method with greater capability in this regard.

`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.

`compile`(*dialect=None*)

`copy_value`(*value*)

`dialect_impl`(*dialect*, ***kwargs*)

`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. Values such as dicts, lists which are serialized into strings are examples of "mutable" column structures.

When this method is overridden, `copy_value()` should also be supplied. The `MutableType` mixin is recommended as a helper.

`result_processor`(*dialect*, *coltype*)
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`.

3.7.5 Base Type API

class `sqlalchemy.types.`**`AbstractType`**(**args*, ***kwargs*)
Bases: `sqlalchemy.sql.visitors.Visitable`

`__init__`(**args*, ***kwargs*)

`bind_processor`(*dialect*)
Defines a bind parameter processing function.

Parameters

- **dialect** – Dialect instance in use.

`compare_values`(*x*, *y*)
Compare two values for equality.

`compile`(*dialect=None*)

`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. Values such as dicts, lists which are serialized into strings are examples of "mutable" column structures.

When this method is overridden, `copy_value()` should also be supplied. The `MutableType` mixin is recommended as a helper.

`result_processor`(*dialect*, *coltype*)
Defines a result-column processing function.

Parameters

- **dialect** – Dialect instance in use.
- **coltype** – DBAPI coltype argument received in cursor.description.

class `sqlalchemy.types.`**`TypeEngine`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.AbstractType`

Base for built-in types.

`__init__`(**args*, ***kwargs*)

`adapt`(*cls*)

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.

compile (*dialect=None*)

copy_value (*value*)

dialect_impl (*dialect, **kwargs*)

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. Values such as dicts, lists which are serialized into strings are examples of "mutable" column structures.

When this method is overridden, `copy_value()` should also be supplied. The `MutableType` mixin is recommended as a helper.

result_processor (*dialect, coltype*)
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 `sqlalchemy.types.`**MutableType**
Bases: `object`

A mixin that marks a `TypeEngine` as representing a mutable Python object type.

"mutable" means that changes can occur in place to a value of this type. Examples includes Python lists, dictionaries, and sets, as well as user-defined objects. The primary need for identification of "mutable" types is by the ORM, which applies special rules to such values in order to guarantee that changes are detected. These rules may have a significant performance impact, described below.

A `MutableType` usually allows a flag called `mutable=True` to enable/disable the "mutability" flag, represented on this class by `is_mutable()`. Examples include `PickleType` and `ARRAY`. Setting this flag to `False` effectively disables any mutability- specific behavior by the ORM.

`copy_value()` and `compare_values()` represent a copy and compare function for values of this type - implementing subclasses should override these appropriately.

The usage of mutable types has significant performance implications when using the ORM. In order to detect changes, the ORM must create a copy of the value when it is first accessed, so that changes to the current value can be compared against the "clean" database-loaded value. Additionally, when the ORM checks to see if any data requires flushing, it must scan through all instances in the session which are known to have "mutable" attributes and compare the current value of each one to its "clean" value. So for example, if the Session contains 6000 objects (a fairly large amount) and autoflush is enabled, every individual execution of `Query` will require a full scan of that subset of the 6000 objects that have mutable attributes, possibly resulting in tens of thousands of additional method calls for every query.

Note that for small numbers (< 100 in the Session at a time) of objects with "mutable" values, the performance degradation is negligible. In most cases it's likely that the convenience allowed by "mutable" change detection outweighs the performance penalty.

It is perfectly fine to represent "mutable" data types with the "mutable" flag set to False, which eliminates any performance issues. It means that the ORM will only reliably detect changes for values of this type if a newly modified value is of a different identity (i.e., `id(value)`) than what was present before - i.e., instead of operations like these:

```
myobject.somedict['foo'] = 'bar'
myobject.someset.add('bar')
myobject.somelist.append('bar')
```

You'd instead say:

```
myobject.somevalue = {'foo':'bar'}
myobject.someset = myobject.someset.union(['bar'])
myobject.somelist = myobject.somelist + ['bar']
```

A future release of SQLAlchemy will include instrumented collection support for mutable types, such that at least usage of plain Python datastructures will be able to emit events for in-place changes, removing the need for pessimistic scanning for changes.

`__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 if the target Python type is 'mutable'.

For `MutableType`, this method is set to return `True`.

class `sqlalchemy.types.`**`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

class `sqlalchemy.types.`**`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.

3.8 Core Event Interfaces

This section describes the various categories of events which can be intercepted in SQLAlchemy core, including execution and connection pool events.

For ORM event documentation, see *ORM Event Interfaces*.

A new version of this API with a significantly more flexible and consistent interface will be available in version 0.7.

3.8.1 Execution, Connection and Cursor Events

class `sqlalchemy.interfaces.`**`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())
```

`begin`(*conn*, *begin*)
Intercept begin() events.

`begin_twophase`(*conn*, *begin_twophase*, *xid*)
Intercept begin_twophase() events.

`commit`(*conn*, *commit*)
Intercept commit() events.

`commit_twophase`(*conn*, *commit_twophase*, *xid*, *is_prepared*)
Intercept commit_twophase() events.

`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.

`prepare_twophase`(*conn*, *prepare_twophase*, *xid*)
Intercept prepare_twophase() events.

`release_savepoint`(*conn*, *release_savepoint*, *name*, *context*)
Intercept release_savepoint() events.

`rollback`(*conn*, *rollback*)
Intercept rollback() events.

rollback_savepoint (*conn*, *rollback_savepoint*, *name*, *context*)
Intercept rollback_savepoint() events.

rollback_twophase (*conn*, *rollback_twophase*, *xid*, *is_prepared*)
Intercept rollback_twophase() events.

savepoint (*conn*, *savepoint*, *name=None*)
Intercept savepoint() events.

3.8.2 Connection Pool Events

class `sqlalchemy.interfaces.`**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

first_connect (*dbapi_con*, *con_record*)
Called exactly once for the first DB-API connection.

dbapi_con A newly connected raw DB-API connection (not a SQLAlchemy `Connection` wrapper).

con_record The `_ConnectionRecord` that persistently manages the connection

3.9 Core Exceptions

Exceptions used with SQLAlchemy.

The base exception class is `SQLAlchemyError`. Exceptions which are raised as a result of DBAPI exceptions are all subclasses of `DBAPIError`.

exception `sqlalchemy.exc.`**ArgumentError**
Bases: `sqlalchemy.exc.SQLAlchemyError`

Raised when an invalid or conflicting function argument is supplied.

This error generally corresponds to construction time state errors.

exception `sqlalchemy.exc.`**CircularDependencyError** (*message*, *cycles*, *edges*)
Bases: `sqlalchemy.exc.SQLAlchemyError`

Raised by topological sorts when a circular dependency is detected

exception `sqlalchemy.exc.`**CompileError**
Bases: `sqlalchemy.exc.SQLAlchemyError`

Raised when an error occurs during SQL compilation

exception `sqlalchemy.exc.`**DBAPIError** (*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.SQLAlchemyError`

Raised when the execution of a database operation fails.

`DBAPIError` wraps exceptions raised by the DB-API underlying the database operation. Driver-specific implementations of the standard DB-API exception types are wrapped by matching sub-types of SQLAlchemy's `DBAPIError` when possible. DB-API's `Error` type maps to `DBAPIError` in SQLAlchemy, otherwise the names are identical. Note that there is no guarantee that different DB-API implementations will raise the same exception type for any given error condition.

If the error-raising operation occured in the execution of a SQL statement, that statement and its parameters will be available on the exception object in the `statement` and `params` attributes.

The wrapped exception object is available in the `orig` attribute. Its type and properties are DB-API implementation specific.

exception `sqlalchemy.exc.`**`DataError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DatabaseError`

Wraps a DB-API DataError.

exception `sqlalchemy.exc.`**`DatabaseError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DBAPIError`

Wraps a DB-API DatabaseError.

exception `sqlalchemy.exc.`**`DisconnectionError`**
Bases: `sqlalchemy.exc.SQLAlchemyError`

A disconnect is detected on a raw DB-API connection.

This error is raised and consumed internally by a connection pool. It can be raised by a `PoolListener` so that the host pool forces a disconnect.

exception `sqlalchemy.exc.`**`IdentifierError`**
Bases: `sqlalchemy.exc.SQLAlchemyError`

Raised when a schema name is beyond the max character limit

exception `sqlalchemy.exc.`**`IntegrityError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DatabaseError`

Wraps a DB-API IntegrityError.

exception `sqlalchemy.exc.`**`InterfaceError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DBAPIError`

Wraps a DB-API InterfaceError.

exception `sqlalchemy.exc.`**`InternalError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DatabaseError`

Wraps a DB-API InternalError.

exception `sqlalchemy.exc.`**`InvalidRequestError`**
Bases: `sqlalchemy.exc.SQLAlchemyError`

SQLAlchemy was asked to do something it can't do.

This error generally corresponds to runtime state errors.

exception `sqlalchemy.exc.`**`NoReferenceError`**
Bases: `sqlalchemy.exc.InvalidRequestError`

Raised by `ForeignKey` to indicate a reference cannot be resolved.

exception `sqlalchemy.exc.`**`NoReferencedColumnError`**
Bases: `sqlalchemy.exc.NoReferenceError`

Raised by `ForeignKey` when the referred `Column` cannot be located.

exception `sqlalchemy.exc.`**`NoReferencedTableError`**
Bases: `sqlalchemy.exc.NoReferenceError`

Raised by `ForeignKey` when the referred `Table` cannot be located.

exception `sqlalchemy.exc.`**`NoSuchColumnError`**
Bases: `exceptions.KeyError`, `sqlalchemy.exc.InvalidRequestError`

A nonexistent column is requested from a `RowProxy`.

exception `sqlalchemy.exc.`**`NoSuchTableError`**
Bases: `sqlalchemy.exc.InvalidRequestError`

Table does not exist or is not visible to a connection.

exception `sqlalchemy.exc.`**`NotSupportedError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DatabaseError`

Wraps a DB-API NotSupportedError.

exception `sqlalchemy.exc.`**`OperationalError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DatabaseError`

Wraps a DB-API OperationalError.

exception `sqlalchemy.exc.`**`ProgrammingError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Bases: `sqlalchemy.exc.DatabaseError`

Wraps a DB-API ProgrammingError.

exception `sqlalchemy.exc.`**`ResourceClosedError`**
Bases: `sqlalchemy.exc.InvalidRequestError`

An operation was requested from a connection, cursor, or other object that's in a closed state.

exception `sqlalchemy.exc.`**`SADeprecationWarning`**
Bases: `exceptions.DeprecationWarning`

Issued once per usage of a deprecated API.

exception `sqlalchemy.exc.`**`SAPendingDeprecationWarning`**
Bases: `exceptions.PendingDeprecationWarning`

Issued once per usage of a deprecated API.

exception `sqlalchemy.exc.`**`SAWarning`**
Bases: `exceptions.RuntimeWarning`

Issued at runtime.

exception `sqlalchemy.exc.`**`SQLAlchemyError`**
Bases: `exceptions.Exception`

Generic error class.

`sqlalchemy.exc.`**`SQLError`**
alias of `DBAPIError`

exception `sqlalchemy.exc.`**`TimeoutError`**
Bases: `sqlalchemy.exc.SQLAlchemyError`

Raised when a connection pool times out on getting a connection.

exception `sqlalchemy.exc.`**`UnboundExecutionError`**
Bases: `sqlalchemy.exc.InvalidRequestError`

SQL was attempted without a database connection to execute it on.

3.10 Custom SQL Constructs and Compilation Extension

Provides an API for creation of custom ClauseElements and compilers.

3.10.1 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 named 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]
```

3.10.2 Dialect-specific compilation rules

Compilers can also be made dialect-specific. The appropriate compiler will be invoked for the dialect in use:

```
from sqlalchemy.schema import DDLElement

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, 'postgresql')
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 `postgresql` dialect is used.

3.10.3 Compiling sub-elements of a custom expression construct

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` both include a `process()` method which can be used for compilation of embedded attributes:

```
from sqlalchemy.sql.expression import Executable, ClauseElement

class InsertFromSelect(Executable, 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 >
```

Cross Compiling between SQL and DDL compilers

SQL and DDL constructs are each compiled using different base compilers - `SQLCompiler` and `DDLCompiler`. A common need is to access the compilation rules of SQL expressions from within a DDL expression. The `DDLCompiler` includes an accessor `sql_compiler` for this reason, such as below where we generate a CHECK constraint that embeds a SQL expression:

```
@compiles(MyConstraint)
def compile_my_constraint(constraint, ddlcompiler, **kw):
    return "CONSTRAINT %s CHECK (%s)" % (
        constraint.name,
        ddlcompiler.sql_compiler.process(constraint.expression)
    )
```

3.10.4 Changing the default compilation of existing constructs

The compiler extension applies just as well to the existing constructs. When overriding the compilation of a built in SQL construct, the @compiles decorator is invoked upon the appropriate class (be sure to use the class, i.e. `Insert` or `Select`, instead of the creation function such as `insert()` or `select()`).

Within the new compilation function, to get at the "original" compilation routine, use the appropriate visit_XXX method - this because compiler.process() will call upon the overriding routine and cause an endless loop. Such as, to add "prefix" to all insert statements:

```
from sqlalchemy.sql.expression import Insert

@compiles(Insert)
```

```
def prefix_inserts(insert, compiler, **kw):
    return compiler.visit_insert(insert.prefix_with("some prefix"), **kw)
```

The above compiler will prefix all INSERT statements with "some prefix" when compiled.

3.10.5 Changing Compilation of Types

`compiler` works for types, too, such as below where we implement the MS-SQL specific 'max' keyword for `String`/`VARCHAR`:

```
@compiles(String, 'mssql')
@compiles(VARCHAR, 'mssql')
def compile_varchar(element, compiler, **kw):
    if element.length == 'max':
        return "VARCHAR('max')"
    else:
        return compiler.visit_VARCHAR(element, **kw)

foo = Table('foo', metadata,
    Column('data', VARCHAR('max'))
)
```

3.10.6 Subclassing Guidelines

A big part of using the compiler extension is subclassing SQLAlchemy expression constructs. To make this easier, the expression and schema packages feature a set of "bases" intended for common tasks. A synopsis is as follows:

- `ClauseElement` - This is the root expression class. Any SQL expression can be derived from this base, and is probably the best choice for longer constructs such as specialized INSERT statements.
- `ColumnElement` - The root of all "column-like" elements. Anything that you'd place in the "columns" clause of a SELECT statement (as well as order by and group by) can derive from this - the object will automatically have Python "comparison" behavior.

  `ColumnElement` classes want to have a `type` member which is expression's return type. This can be established at the instance level in the constructor, or at the class level if its generally constant:

  ```
  class timestamp(ColumnElement):
      type = TIMESTAMP()
  ```

- `FunctionElement` - This is a hybrid of a `ColumnElement` and a "from clause" like object, and represents a SQL function or stored procedure type of call. Since most databases support statements along the line of "SELECT FROM <some function>" `FunctionElement` adds in the ability to be used in the FROM clause of a `select()` construct:

  ```
  from sqlalchemy.sql.expression import FunctionElement

  class coalesce(FunctionElement):
      name = 'coalesce'

  @compiles(coalesce)
  def compile(element, compiler, **kw):
      return "coalesce(%s)" % compiler.process(element.clauses)

  @compiles(coalesce, 'oracle')
  def compile(element, compiler, **kw):
  ```

```
        if len(element.clauses) > 2:
            raise TypeError("coalesce only supports two arguments on Oracle")
        return "nvl(%s)" % compiler.process(element.clauses)
```

- `DDLElement` - The root of all DDL expressions, like CREATE TABLE, ALTER TABLE, etc. Compilation of `DDLElement` subclasses is issued by a `DDLCompiler` instead of a `SQLCompiler`. `DDLElement` also features `Table` and `MetaData` event hooks via the `execute_at()` method, allowing the construct to be invoked during CREATE TABLE and DROP TABLE sequences.
- `Executable` - This is a mixin which should be used with any expression class that represents a "standalone" SQL statement that can be passed directly to an `execute()` method. It is already implicit within `DDLElement` and `FunctionElement`.

3.11 Expression Serializer Extension

Serializer/Deserializer objects for usage with SQLAlchemy query structures, allowing "contextual" deserialization.

Any SQLAlchemy query structure, either based on sqlalchemy.sql.* or sqlalchemy.orm.* can be used. The mappers, Tables, Columns, Session etc. which are referenced by the structure are not persisted in serialized form, but are instead re-associated with the query structure when it is deserialized.

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.

The serializer module is only appropriate for query structures. It is not needed for:

- instances of user-defined classes. These contain no references to engines, sessions or expression constructs in the typical case and can be serialized directly.
- Table metadata that is to be loaded entirely from the serialized structure (i.e. is not already declared in the application). Regular pickle.loads()/dumps() can be used to fully dump any `MetaData` object, typically one which was reflected from an existing database at some previous point in time. The serializer module is specifically for the opposite case, where the Table metadata is already present in memory.

`sqlalchemy.ext.serializer.`**`Serializer`**(**args*, ***kw*)

`sqlalchemy.ext.serializer.`**`Deserializer`**(*file*, *metadata=None*, *scoped_session=None*, *engine=None*)

`sqlalchemy.ext.serializer.`**`dumps`**(*obj*, *protocol=0*) `sqlalchemy.ext.serializer.`**`loads`**(*data*, *metadata=None*, *scoped_session=None*, *engine=None*)

CHAPTER

FOUR

DIALECTS

The **dialect** is the system SQLAlchemy uses to communicate with various types of DBAPIs and databases. A compatibility chart of supported backends can be found at *Supported Databases*. The sections that follow contain reference documentation and notes specific to the usage of each backend, as well as notes for the various DBAPIs.

Note that not all backends are fully ported and tested with current versions of SQLAlchemy. The compatibility chart should be consulted to check for current support level.

4.1 Firebird

Support for the Firebird database.

Connectivity is usually supplied via the kinterbasdb DBAPI module.

4.1.1 Dialects

Firebird offers two distinct dialects (not to be confused with a SQLAlchemy `Dialect`):

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.

The SQLAlchemy Firebird dialect detects these versions and adjusts its representation of SQL accordingly. However, support for dialect 1 is not well tested and probably has incompatibilities.

4.1.2 Locking Behavior

Firebird locks tables aggressively. For this reason, a DROP TABLE may hang until other transactions are released. SQLAlchemy does its best to release transactions as quickly as possible. The most common cause of hanging transactions is a non-fully consumed result set, i.e.:

```
result = engine.execute("select * from table")
row = result.fetchone()
return
```

Where above, the `ResultProxy` has not been fully consumed. The connection will be returned to the pool and the transactional state rolled back once the Python garbage collector reclaims the objects which hold onto the connection, which often occurs asynchronously. The above use case can be alleviated by calling `first()` on the `ResultProxy` which will fetch the first row and immediately close all remaining cursor/connection resources.

4.1.3 RETURNING support

Firebird 2.0 supports returning a result set from inserts, and 2.1 extends that to deletes and updates. This is generically exposed by the SQLAlchemy `returning()` method, such as:

```
# INSERT..RETURNING
result = table.insert().returning(table.c.col1, table.c.col2).\
               values(name='foo')
print result.fetchall()

# UPDATE..RETURNING
raises = empl.update().returning(empl.c.id, empl.c.salary).\
              where(empl.c.sales>100).\
              values(dict(salary=empl.c.salary * 1.1))
print raises.fetchall()
```

4.1.4 kinterbasdb

The most common way to connect to a Firebird engine is implemented by kinterbasdb, currently maintained directly by the Firebird people.

The connection URL is of the form `firebird[+kinterbasdb]://user:password@host:port/path/to/db[?key=va`

Kinterbasedb backend specific keyword arguments are:

- type_conv - select the kind of mapping done on the types: by default SQLAlchemy uses 200 with Unicode, datetime and decimal support (see details).
- concurrency_level - set the backend policy with regards to threading issues: by default SQLAlchemy uses policy 1 (see details).
- enable_rowcount - True by default, setting this to False disables the usage of "cursor.rowcount" with the Kinterbasdb dialect, which SQLAlchemy ordinarily calls upon automatically after any UPDATE or DELETE statement. When disabled, SQLAlchemy's ResultProxy will return -1 for result.rowcount. The rationale here is that Kinterbasdb requires a second round trip to the database when .rowcount is called - since SQLA's resultproxy automatically closes the cursor after a non-result-returning statement, rowcount must be called, if at all, before the result object is returned. Additionally, cursor.rowcount may not return correct results with older versions of Firebird, and setting this flag to False will also cause the SQLAlchemy ORM to ignore its usage. The behavior can also be controlled on a per-execution basis using the *enable_rowcount* option with `execution_options()`:

  ```
  conn = engine.connect().execution_options(enable_rowcount=True)
  r = conn.execute(stmt)
  print r.rowcount
  ```

4.2 Informix

Support for the Informix database.

This dialect is mostly functional as of SQLAlchemy 0.6.5.

4.2.1 informixdb Notes

Support for the informixdb DBAPI.

informixdb is available at:

http://informixdb.sourceforge.net/

Connecting

Sample informix connection:

```
engine = create_engine('informix+informixdb://user:password@host/dbname')
```

4.3 MaxDB

Support for the MaxDB database.

This dialect is *not* ported to SQLAlchemy 0.6.

This dialect is *not* tested on SQLAlchemy 0.6.

4.3.1 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.

4.3.2 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.

4.3.3 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.

4.4 Microsoft Access

Support for the Microsoft Access database.

This dialect is *not* ported to SQLAlchemy 0.6.

This dialect is *not* tested on SQLAlchemy 0.6.

4.5 Microsoft SQL Server

Support for the Microsoft SQL Server database.

4.5.1 Connecting

See the individual driver sections below for details on connecting.

4.5.2 Auto Increment Behavior

`IDENTITY` columns are supported by using SQLAlchemy `schema.Sequence()` objects. In other words:

```
from sqlalchemy import Table, Integer, Sequence, Column

Table('test', metadata,
       Column('id', Integer,
              Sequence('blah',100,10), primary_key=True),
       Column('name', String(20))
     ).create(some_engine)
```

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.

Implicit `autoincrement` behavior works the same in MSSQL as it does in other dialects and results in an `IDENTITY` column.

- Support for `SET IDENTITY_INSERT ON` mode (automagic on / off for `INSERT` s)
- Support for auto-fetching of `@@IDENTITY/@@SCOPE_IDENTITY()` on `INSERT`

4.5.3 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:

```
from sqlalchemy.dialects.mssql import VARCHAR
Column('login', VARCHAR(32, collation='Latin1_General_CI_AS'))
```

When such a column is associated with a `Table`, the CREATE TABLE statement for this column will yield:

```
login VARCHAR(32) COLLATE Latin1_General_CI_AS NULL
```

4.5.4 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 using SQL Server 2005 or above, LIMIT with OFFSET support is available through the `ROW_NUMBER OVER` construct. For versions below 2005, LIMIT with OFFSET usage will fail.

4.5.5 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.

4.5.6 Date / Time Handling

DATE and TIME are supported. Bind parameters are converted to datetime.datetime() objects as required by most MSSQL drivers, and results are processed from strings if needed. The DATE and TIME types are not available for MSSQL 2005 and previous - if a server version below 2008 is detected, DDL for these types will be issued as DATETIME.

4.5.7 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 return 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.

4.5.8 Known Issues

- No support for more than one `IDENTITY` column per table
- reflection of indexes does not work with versions older than SQL Server 2005

4.5.9 SQL Server Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with SQL server are importable from the top level dialect, whether they originate from `sqlalchemy.types` or from the local dialect:

```
from sqlalchemy.dialects.mssql import \
    BIGINT, BINARY, BIT, CHAR, DATE, DATETIME, DATETIME2, \
    DATETIMEOFFSET, DECIMAL, FLOAT, IMAGE, INTEGER, MONEY, \
    NCHAR, NTEXT, NUMERIC, NVARCHAR, REAL, SMALLDATETIME, \
    SMALLINT, SMALLMONEY, SQL_VARIANT, TEXT, TIME, \
    TIMESTAMP, TINYINT, UNIQUEIDENTIFIER, VARBINARY, VARCHAR
```

Types which are specific to SQL Server, or have SQL Server-specific construction arguments, are as follows:

class `sqlalchemy.dialects.mssql.base.`**`BIT`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.dialects.mssql.base.`**`CHAR`**(**args*, ***kw*)
Bases: `sqlalchemy.dialects.mssql.base._StringType`, `sqlalchemy.types.CHAR`

MSSQL CHAR type, for fixed-length non-Unicode data with a maximum of 8,000 characters.

`__init__`(**args*, ***kw*)
Construct a CHAR.

Parameters

- **length** – Optinal, maximum data length, in characters.
- **convert_unicode** – defaults to False. If True, convert `unicode` data sent to the database to a `str` bytestring, and convert bytestrings coming back from the database into `unicode`.

  Bytestrings are encoded using the dialect's `encoding`, which defaults to *utf-8*.

  If False, may be overridden by `sqlalchemy.engine.base.Dialect.convert_unicode`.
- **collation** – Optional, a column-level collation for this string value. Accepts a Windows Collation Name or a SQL Collation Name.

class `sqlalchemy.dialects.mssql.base.`**`DATETIME2`**(*precision=None*, ***kwargs*)
Bases: `sqlalchemy.dialects.mssql.base._DateTimeBase`, `sqlalchemy.types.DateTime`

`__init__`(*precision=None*, ***kwargs*)

class `sqlalchemy.dialects.mssql.base.`**`DATETIMEOFFSET`**(*precision=None*, ***kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

`__init__`(*precision=None*, ***kwargs*)

class `sqlalchemy.dialects.mssql.base.`**`IMAGE`**(*length=None*)
Bases: `sqlalchemy.types.LargeBinary`

`__init__`(*length=None*)
Construct a LargeBinary type.

Parameters

- **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a small BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class `sqlalchemy.dialects.mssql.base.`**`MONEY`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.dialects.mssql.base.`**`NCHAR`**(**args*, ***kw*)
Bases: `sqlalchemy.dialects.mssql.base._StringType`, `sqlalchemy.types.NCHAR`

MSSQL NCHAR type.

For fixed-length unicode character data up to 4,000 characters.

`__init__`(**args*, ***kw*)
Construct an NCHAR.

Parameters

- **length** – Optional, Maximum data length, in characters.
- **collation** – Optional, a column-level collation for this string value. Accepts a Windows Collation Name or a SQL Collation Name.

class `sqlalchemy.dialects.mssql.base.`**`NTEXT`**(**args*, ***kwargs*)
Bases: `sqlalchemy.dialects.mssql.base._StringType`, `sqlalchemy.types.UnicodeText`

MSSQL NTEXT type, for variable-length unicode text up to 2^30 characters.

`__init__`(**args*, ***kwargs*)
Construct a NTEXT.

Parameters

- **collation** – Optional, a column-level collation for this string value. Accepts a Windows Collation Name or a SQL Collation Name.

class `sqlalchemy.dialects.mssql.base.`**`NVARCHAR`**(**args*, ***kw*)
Bases: `sqlalchemy.dialects.mssql.base._StringType`, `sqlalchemy.types.NVARCHAR`

MSSQL NVARCHAR type.

For variable-length unicode character data up to 4,000 characters.

`__init__`(**args*, ***kw*)
Construct a NVARCHAR.

Parameters

- **length** – Optional, Maximum data length, in characters.
- **collation** – Optional, a column-level collation for this string value. Accepts a Windows Collation Name or a SQL Collation Name.

class `sqlalchemy.dialects.mssql.base.`**`REAL`**
Bases: `sqlalchemy.types.Float`

A type for `real` numbers.

`__init__`()

class `sqlalchemy.dialects.mssql.base.`**`SMALLDATETIME`**(*timezone=False*)
Bases: `sqlalchemy.dialects.mssql.base._DateTimeBase`, `sqlalchemy.types.DateTime`

__init__(*timezone=False*)

class sqlalchemy.dialects.mssql.base.**SMALLMONEY**(**args*, ***kwargs*)
Bases: sqlalchemy.types.TypeEngine

__init__(**args*, ***kwargs*)

class sqlalchemy.dialects.mssql.base.**SQL_VARIANT**(**args*, ***kwargs*)
Bases: sqlalchemy.types.TypeEngine

__init__(**args*, ***kwargs*)

class sqlalchemy.dialects.mssql.base.**TEXT**(**args*, ***kw*)
Bases: sqlalchemy.dialects.mssql.base._StringType, sqlalchemy.types.TEXT

MSSQL TEXT type, for variable-length text up to 2^31 characters.

__init__(**args*, ***kw*)
Construct a TEXT.

Parameters

- **collation** – Optional, a column-level collation for this string value. Accepts a Windows Collation Name or a SQL Collation Name.

class sqlalchemy.dialects.mssql.base.**TIME**(*precision=None*, ***kwargs*)
Bases: sqlalchemy.types.TIME

__init__(*precision=None*, ***kwargs*)

class sqlalchemy.dialects.mssql.base.**TINYINT**(**args*, ***kwargs*)
Bases: sqlalchemy.types.Integer

__init__(**args*, ***kwargs*)

class sqlalchemy.dialects.mssql.base.**UNIQUEIDENTIFIER**(**args*, ***kwargs*)
Bases: sqlalchemy.types.TypeEngine

__init__(**args*, ***kwargs*)

class sqlalchemy.dialects.mssql.base.**VARCHAR**(**args*, ***kw*)
Bases: sqlalchemy.dialects.mssql.base._StringType, sqlalchemy.types.VARCHAR

MSSQL VARCHAR type, for variable-length non-Unicode data with a maximum of 8,000 characters.

__init__(**args*, ***kw*)
Construct a VARCHAR.

Parameters

- **length** – Optinal, maximum data length, in characters.
- **convert_unicode** – defaults to False. If True, convert `unicode` data sent to the database to a `str` bytestring, and convert bytestrings coming back from the database into `unicode`.

  Bytestrings are encoded using the dialect's `encoding`, which defaults to *utf-8*.

  If False, may be overridden by `sqlalchemy.engine.base.Dialect.convert_unicode`.
- **collation** – Optional, a column-level collation for this string value. Accepts a Windows Collation Name or a SQL Collation Name.

4.5.10 PyODBC

Support for MS-SQL via pyodbc.

pyodbc is available at:

> http://pypi.python.org/pypi/pyodbc/

Connecting

Examples of pyodbc connection string URLs:

- `mssql+pyodbc://mydsn` - connects using the specified DSN named `mydsn`. The connection string that is created will appear like:

  ```
  dsn=mydsn;Trusted_Connection=Yes
  ```

- `mssql+pyodbc://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+pyodbc://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+pyodbc://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+pyodbc://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+pyodbc://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+pyodbc:///?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'
```

4.5.11 mxODBC

Support for MS-SQL via mxODBC.

mxODBC is available at:

http://www.egenix.com/

This was tested with mxODBC 3.1.2 and the SQL Server Native Client connected to MSSQL 2005 and 2008 Express Editions.

Connecting

Connection is via DSN:

```
mssql+mxodbc://<username>:<password>@<dsnname>
```

Execution Modes

mxODBC features two styles of statement execution, using the `cursor.execute()` and `cursor.executedirect()` methods (the second being an extension to the DBAPI specification). The former makes use of a particular API call specific to the SQL Server Native Client ODBC driver known SQLDescribeParam, while the latter does not.

mxODBC apparently only makes repeated use of a single prepared statement when SQLDescribeParam is used. The advantage to prepared statement reuse is one of performance. The disadvantage is that SQLDescribeParam has a limited set of scenarios in which bind parameters are understood, including that they cannot be placed within the argument lists of function calls, anywhere outside the FROM, or even within subqueries within the FROM clause - making the usage of bind parameters within SELECT statements impossible for all but the most simplistic statements.

For this reason, the mxODBC dialect uses the "native" mode by default only for INSERT, UPDATE, and DELETE statements, and uses the escaped string mode for all other statements.

This behavior can be controlled via `execution_options()` using the `native_odbc_execute` flag with a value of `True` or `False`, where a value of `True` will unconditionally use native bind parameters and a value of `False` will uncondtionally use string-escaped parameters.

4.5.12 pymssql

Support for the pymssql dialect.

This dialect supports pymssql 1.0 and greater.

pymssql is available at:

http://pymssql.sourceforge.net/

Connecting

Sample connect string:

```
mssql+pymssql://<username>:<password>@<freetds_name>
```

Adding "?charset=utf8" or similar will cause pymssql to return strings as Python unicode objects. This can potentially improve performance in some scenarios as decoding of strings is handled natively.

Limitations

pymssql inherits a lot of limitations from FreeTDS, including:

- no support for multibyte schema identifiers
- poor support for large decimals
- poor support for binary fields
- poor support for VARCHAR/CHAR fields over 255 characters

Please consult the pymssql documentation for further information.

4.5.13 zxjdbc Notes

Support for the Microsoft SQL Server database via the zxjdbc JDBC connector.

JDBC Driver

Requires the jTDS driver, available from: http://jtds.sourceforge.net/

Connecting

URLs are of the standard form of `mssql+zxjdbc://user:pass@host:port/dbname[?key=value&key=value...]`.

Additional arguments which may be specified either as query string arguments on the URL, or as keyword arguments to `create_engine()` will be passed as Connection properties to the underlying JDBC driver.

4.5.14 AdoDBAPI

The adodbapi dialect is not implemented for 0.6 at this time.

4.6 MySQL

Support for the MySQL database.

4.6.1 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.

Most available DBAPI drivers are supported; see below.

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.

4.6.2 Connecting

See the API documentation on individual drivers for details on connecting.

4.6.3 Connection Timeouts

MySQL features an automatic connection close behavior, for connections that have been idle for eight hours or more. To circumvent having this issue, use the `pool_recycle` option which controls the maximum age of any connection:

```
engine = create_engine('mysql+mysqldb://...', pool_recycle=3600)
```

4.6.4 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'
      )
```

4.6.5 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)
     )
```

4.6.6 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
```

4.6.7 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)
  ```

4.6.8 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.

4.6.9 MySQL Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with MySQL are importable from the top level dialect:

```
from sqlalchemy.dialects.mysql import \
        BIGINT, BINARY, BIT, BLOB, BOOLEAN, CHAR, DATE, \
        DATETIME, DECIMAL, DECIMAL, DOUBLE, ENUM, FLOAT, INTEGER, \
        LONGBLOB, LONGTEXT, MEDIUMBLOB, MEDIUMINT, MEDIUMTEXT, NCHAR, \
        NUMERIC, NVARCHAR, REAL, SET, SMALLINT, TEXT, TIME, TIMESTAMP, \
        TINYBLOB, TINYINT, TINYTEXT, VARBINARY, VARCHAR, YEAR
```

Types which are specific to MySQL, or have MySQL-specific construction arguments, are as follows:

class `sqlalchemy.dialects.mysql.base.`**`BIGINT`**(*display_width=None*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.base._IntegerType`, `sqlalchemy.types.BIGINT`

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 `sqlalchemy.dialects.mysql.base.`**`BINARY`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

The SQL BINARY type.

`__init__`(*length=None*)

class `sqlalchemy.dialects.mysql.base.`**`BIT`**(*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.

Parameters

- **length** – Optional, number of bits.

class `sqlalchemy.dialects.mysql.base.`**`BLOB`**(*length=None*)
Bases: `sqlalchemy.types.LargeBinary`

The SQL BLOB type.

`__init__`(*length=None*)
Construct a LargeBinary type.

Parameters

- **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a small BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class `sqlalchemy.dialects.mysql.base.`**`BOOLEAN`**(*create_constraint=True, name=None*)
Bases: `sqlalchemy.types.Boolean`

The SQL BOOLEAN type.

`__init__`(*create_constraint=True, name=None*)
Construct a Boolean.

Parameters

- **create_constraint** – defaults to True. If the boolean is generated as an int/smallint, also create a CHECK constraint on the table that ensures 1 or 0 as a value.
- **name** – if a CHECK constraint is generated, specify the name of the constraint.

class `sqlalchemy.dialects.mysql.base.`**`CHAR`**(*length, **kwargs*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`, `sqlalchemy.types.CHAR`

MySQL CHAR type, for fixed-length character data.

`__init__`(*length, **kwargs*)
Construct a CHAR.

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 `sqlalchemy.dialects.mysql.base.`**`DATE`**(**args, **kwargs*)
Bases: `sqlalchemy.types.Date`

The SQL DATE type.

`__init__`(**args, **kwargs*)

class `sqlalchemy.dialects.mysql.base.`**`DATETIME`**(*timezone=False*)
Bases: `sqlalchemy.types.DateTime`

The SQL DATETIME type.

`__init__`(*timezone=False*)

class `sqlalchemy.dialects.mysql.base.`**`DECIMAL`**(*precision=None, scale=None, asdecimal=True, **kw*)
Bases: `sqlalchemy.dialects.mysql.base._NumericType`, `sqlalchemy.types.DECIMAL`

MySQL DECIMAL type.

__init__(*precision=None*, *scale=None*, *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 `sqlalchemy.dialects.mysql.base.`**`DOUBLE`**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.base._FloatType`

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 `sqlalchemy.dialects.mysql.base.`**`ENUM`**(**enums*, ***kw*)
Bases: `sqlalchemy.types.Enum`, `sqlalchemy.dialects.mysql.base._StringType`

MySQL ENUM type.

__init__(**enums*, ***kw*)
Construct an ENUM.

Example:

Column('myenum', MSEnum("foo", "bar", "baz"))

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 - this usage is deprecated.

  '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 `sqlalchemy.dialects.mysql.base.`**`FLOAT`**(*precision=None*, *scale=None*, *asdecimal=False*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.base._FloatType`, `sqlalchemy.types.FLOAT`

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 `sqlalchemy.dialects.mysql.base.`**`INTEGER`**(*display_width=None*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.base._IntegerType`, `sqlalchemy.types.INTEGER`

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 `sqlalchemy.dialects.mysql.base.`**`LONGBLOB`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

MySQL LONGBLOB type, for binary data up to 2^32 bytes.

`__init__`(*length=None*)

class `sqlalchemy.dialects.mysql.base.`**`LONGTEXT`**(***kwargs*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`

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 `sqlalchemy.dialects.mysql.base.`**`MEDIUMBLOB`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

MySQL MEDIUMBLOB type, for binary data up to 2^24 bytes.

`__init__`(*length=None*)

class `sqlalchemy.dialects.mysql.base.`**`MEDIUMINT`**(*display_width=None*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.base._IntegerType`

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 `sqlalchemy.dialects.mysql.base.`**`MEDIUMTEXT`**(***kwargs*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`

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 `sqlalchemy.dialects.mysql.base.`**`NCHAR`**(*length=None*, ***kwargs*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`, `sqlalchemy.types.NCHAR`

MySQL NCHAR type.

For fixed-length character data in the server's configured national character set.

`__init__`(*length=None*, ***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 `sqlalchemy.dialects.mysql.base.`**`NUMERIC`**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.base._NumericType`, `sqlalchemy.types.NUMERIC`

MySQL NUMERIC type.

`__init__`(*precision=None*, *scale=None*, *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 `sqlalchemy.dialects.mysql.base.`**`NVARCHAR`**(*length=None*, ***kwargs*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`, `sqlalchemy.types.NVARCHAR`

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 `sqlalchemy.dialects.mysql.base.`**`REAL`**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.base._FloatType`

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 `sqlalchemy.dialects.mysql.base.`**`SET`**(**values*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`

MySQL SET type.

`__init__`(**values*, ***kw*)
Construct a SET.

Example:

```
Column('myset', MSSet("'foo'", "'bar'", "'baz'"))
```

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 `sqlalchemy.dialects.mysql.base.`**`SMALLINT`**(*display_width=None*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.base._IntegerType`, `sqlalchemy.types.SMALLINT`

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 `sqlalchemy.dialects.mysql.base.`**`TEXT`**(*length=None*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.base._StringType`, `sqlalchemy.types.TEXT`

MySQL TEXT type, for text up to 2^16 characters.

`__init__`(*length=None*, ***kw*)

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 `sqlalchemy.dialects.mysql.base.`**`TIME`**(*timezone=False*)

Bases: `sqlalchemy.types.Time`

The SQL TIME type.

`__init__`(*timezone=False*)

class `sqlalchemy.dialects.mysql.base.TIMESTAMP`(*timezone=False*)
Bases: `sqlalchemy.types.TIMESTAMP`

MySQL TIMESTAMP type.

`__init__`(*timezone=False*)

class `sqlalchemy.dialects.mysql.base.TINYBLOB`(*length=None*)
Bases: `sqlalchemy.types._Binary`

MySQL TINYBLOB type, for binary data up to 2^8 bytes.

`__init__`(*length=None*)

class `sqlalchemy.dialects.mysql.base.TINYINT`(*display_width=None*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.base._IntegerType`

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 `sqlalchemy.dialects.mysql.base.TINYTEXT`(***kwargs*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`

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 `sqlalchemy.dialects.mysql.base.`**`VARBINARY`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

The SQL VARBINARY type.

`__init__`(*length=None*)

class `sqlalchemy.dialects.mysql.base.`**`VARCHAR`**(*length=None*, ***kwargs*)
Bases: `sqlalchemy.dialects.mysql.base._StringType`, `sqlalchemy.types.VARCHAR`

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 `sqlalchemy.dialects.mysql.base.`**`YEAR`**(*display_width=None*)
Bases: `sqlalchemy.types.TypeEngine`

MySQL YEAR type, for single byte storage of years 1901-2155.

`__init__`(*display_width=None*)

4.6.10 MySQL-Python Notes

Support for the MySQL database via the MySQL-python adapter.

MySQL-Python is available at:

http://sourceforge.net/projects/mysql-python

At least version 1.2.1 or 1.2.2 should be used.

Connecting

Connect string format:

```
mysql+mysqldb://<user>:<password>@<host>[:<port>]/<dbname>
```

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+mysqldb:///mydb?charset=utf8')

# set client encoding to utf8; all strings come back as utf8 str
create_engine('mysql+mysqldb:///mydb?charset=utf8&use_unicode=0')
```

Known Issues

MySQL-python at least as of version 1.2.2 has a serious memory leak related to unicode conversion, a feature which is disabled via `use_unicode=0`. The recommended connection form with SQLAlchemy is:

```
engine = create_engine('mysql://scott:tiger@localhost/test?charset=utf8&use_unicode=0', poo
```

4.6.11 OurSQL Notes

Support for the MySQL database via the oursql adapter.

OurSQL is available at:

> http://packages.python.org/oursql/

Connecting

Connect string format:

```
mysql+oursql://<user>:<password>@<host>[:<port>]/<dbname>
```

Character Sets

oursql defaults to using `utf8` as the connection charset, but other encodings may be used instead. Like the MySQL-Python driver, unicode support can be completely disabled:

```
# oursql sets the connection charset to utf8 automatically; all strings come
# back as utf8 str
create_engine('mysql+oursql:///mydb?use_unicode=0')
```

To not automatically use `utf8` and instead use whatever the connection defaults to, there is a separate parameter:

```
# use the default connection charset; all strings come back as unicode
create_engine('mysql+oursql:///mydb?default_charset=1')

# use latin1 as the connection charset; all strings come back as unicode
create_engine('mysql+oursql:///mydb?charset=latin1')
```

4.6.12 MySQL-Connector Notes

Support for the MySQL database via the MySQL Connector/Python adapter.

MySQL Connector/Python is available at:

> https://launchpad.net/myconnpy

Connecting

Connect string format:

`mysql+mysqlconnector://<user>:<password>@<host>[:<port>]/<dbname>`

4.6.13 pyodbc Notes

Support for the MySQL database via the pyodbc adapter.

pyodbc is available at:

> http://pypi.python.org/pypi/pyodbc/

Connecting

Connect string:

`mysql+pyodbc://<username>:<password>@<dsnname>`

Limitations

The mysql-pyodbc dialect is subject to unresolved character encoding issues which exist within the current ODBC drivers available. (see http://code.google.com/p/pyodbc/issues/detail?id=25). Consider usage of OurSQL, MySQLdb, or MySQL-connector/Python.

4.6.14 zxjdbc Notes

Support for the MySQL database via Jython's zxjdbc JDBC connector.

JDBC Driver

The official MySQL JDBC driver is at http://dev.mysql.com/downloads/connector/j/.

Connecting

Connect string format:

> mysql+zxjdbc://<user>:<password>@<hostname>[:<port>]/<database>

Character Sets

SQLAlchemy zxjdbc dialects pass unicode straight through to the zxjdbc/JDBC layer. To allow multiple character sets to be sent from the MySQL Connector/J JDBC driver, by default SQLAlchemy sets its `characterEncoding` connection property to `UTF-8`. It may be overriden via a `create_engine` URL parameter.

4.7 Oracle

Support for the Oracle database.

Oracle version 8 through current (11g at the time of this writing) are supported.

For information on connecting via specific drivers, see the documentation for that driver.

4.7.1 Connect Arguments

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

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

4.7.2 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
)
```

4.7.3 Identifier Casing

In Oracle, the data dictionary represents all case insensitive identifier names using UPPERCASE text. SQLAlchemy on the other hand considers an all-lower case identifier name to be case insensitive. The Oracle dialect converts all case insensitive identifiers to and from those two formats during schema level communication, such as reflection of tables and indexes. Using an UPPERCASE name on the SQLAlchemy side indicates a case sensitive identifier, and SQLAlchemy will quote the name - this will cause mismatches against data dictionary data received from Oracle, so unless identifier names have been truly created as case sensitive (i.e. using quoted names), all lowercase names should be used on the SQLAlchemy side.

4.7.4 Unicode

SQLAlchemy 0.6 uses the "native unicode" mode provided as of cx_oracle 5. cx_oracle 5.0.2 or greater is recommended for support of NCLOB. If not using cx_oracle 5, the NLS_LANG environment variable needs to be set in order for the oracle client library to use proper encoding, such as "AMERICAN_AMERICA.UTF8".

Also note that Oracle supports unicode data through the NVARCHAR and NCLOB data types. When using the SQLAlchemy Unicode and UnicodeText types, these DDL types will be used within CREATE TABLE statements. Usage of VARCHAR2 and CLOB with unicode text still requires NLS_LANG to be set.

4.7.5 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().

4.7.6 ON UPDATE CASCADE

Oracle doesn't have native ON UPDATE CASCADE functionality. A trigger based solution is available at http://asktom.oracle.com/tkyte/update_cascade/index.html .

When using the SQLAlchemy ORM, the ORM has limited ability to manually issue cascading updates - specify ForeignKey objects using the "deferrable=True, initially='deferred'" keyword arguments, and specify "passive_updates=False" on each relationship().

4.7.7 Oracle 8 Compatibility

When Oracle 8 is detected, the dialect internally configures itself to the following behaviors:

- the use_ansi flag is set to False. This has the effect of converting all JOIN phrases into the WHERE clause, and in the case of LEFT OUTER JOIN makes use of Oracle's (+) operator.
- the NVARCHAR2 and NCLOB datatypes are no longer generated as DDL when the `Unicode` is used - VARCHAR2 and CLOB are issued instead. This because these types don't seem to work correctly on Oracle 8 even though they are available. The `NVARCHAR` and `NCLOB` types will always generate NVARCHAR2 and NCLOB.
- the "native unicode" mode is disabled when using cx_oracle, i.e. SQLAlchemy encodes all Python unicode objects to "string" before passing in as bind parameters.

4.7.8 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.

4.7.9 Oracle Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with Oracle are importable from the top level dialect, whether they originate from `sqlalchemy.types` or from the local dialect:

```
from sqlalchemy.dialects.oracle import \
            BFILE, BLOB, CHAR, CLOB, DATE, DATETIME, \
            DOUBLE_PRECISION, FLOAT, INTERVAL, LONG, NCLOB, \
            NUMBER, NVARCHAR, NVARCHAR2, RAW, TIMESTAMP, VARCHAR, \
            VARCHAR2
```

Types which are specific to Oracle, or have Oracle-specific construction arguments, are as follows:

class `sqlalchemy.dialects.oracle.base.`**`BFILE`**(*length=None*)

Bases: `sqlalchemy.types.LargeBinary`

`__init__`(*length=None*)

Construct a LargeBinary type.

Parameters

- **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a small BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class `sqlalchemy.dialects.oracle.base.`**`DOUBLE_PRECISION`**(*precision=None, scale=None, asdecimal=None*)

Bases: `sqlalchemy.types.Numeric`

`__init__`(*precision=None, scale=None, asdecimal=None*)

class `sqlalchemy.dialects.oracle.base.`**`INTERVAL`**(*day_precision=None, second_precision=None*)

Bases: `sqlalchemy.types.TypeEngine`

`__init__`(*day_precision=None, second_precision=None*)

Construct an INTERVAL.

Note that only DAY TO SECOND intervals are currently supported. This is due to a lack of support for YEAR TO MONTH intervals within available DBAPIs (cx_oracle and zxjdbc).

Parameters

- **day_precision** – the day precision value. this is the number of digits to store for the day field. Defaults to "2"
- **second_precision** – the second precision value. this is the number of digits to store for the fractional seconds field. Defaults to "6".

class `sqlalchemy.dialects.oracle.base.`**`NCLOB`**(*length=None, convert_unicode=False, assert_unicode=None, unicode_error=None, _warn_on_bytestring=False*)

Bases: `sqlalchemy.types.Text`

`__init__`(*length=None, convert_unicode=False, assert_unicode=None, unicode_error=None, _warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL statements. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued. Whether the value is interpreted as bytes or characters is database specific.
- **convert_unicode** – defaults to False. If True, the type will do what is necessary in order to accept Python Unicode objects as bind parameters, and to return Python Unicode objects in result rows. This may require SQLAlchemy to explicitly coerce incoming Python unicodes into an encoding, and from an encoding back to Unicode, or it may not require any interaction from SQLAlchemy at all, depending on the DBAPI in use.

  When SQLAlchemy performs the encoding/decoding, the encoding used is configured via `encoding`, which defaults to *utf-8*.

  The "convert_unicode" behavior can also be turned on for all String types by setting `sqlalchemy.engine.base.Dialect.convert_unicode` on create_engine().

  To instruct SQLAlchemy to perform Unicode encoding/decoding even on a platform that already handles Unicode natively, set convert_unicode='force'. This will incur significant performance overhead when fetching unicode result columns.
- **assert_unicode** – Deprecated. A warning is raised in all cases when a non-Unicode object is passed when SQLAlchemy would coerce into an encoding (note: but **not** when the DBAPI handles unicode objects natively). To suppress or raise this warning to an error, use the Python warnings filter documented at: http://docs.python.org/library/warnings.html
- **unicode_error** – Optional, a method to use to handle Unicode conversion errors. Behaves like the 'errors' keyword argument to the standard library's string.decode() functions. This flag requires that *convert_unicode* is set to *"force"* - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as an absolute last resort for reading strings from a column with varied or corrupted encodings, which only applies to databases that accept invalid encodings in the first place (i.e. MySQL. *not* PG, Sqlite, etc.)

class `sqlalchemy.dialects.oracle.base.`**`NUMBER`**(*precision=None*, *scale=None*, *asdecimal=None*)

Bases: `sqlalchemy.types.Numeric`, `sqlalchemy.types.Integer`

`__init__`(*precision=None*, *scale=None*, *asdecimal=None*)

class `sqlalchemy.dialects.oracle.base.`**`LONG`**(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)

Bases: `sqlalchemy.types.Text`

`__init__`(*length=None*, *convert_unicode=False*, *assert_unicode=None*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL statements. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE`

TABLE DDL is issued. Whether the value is interpreted as bytes or characters is database specific.

- **convert_unicode** – defaults to False. If True, the type will do what is necessary in order to accept Python Unicode objects as bind parameters, and to return Python Unicode objects in result rows. This may require SQLAlchemy to explicitly coerce incoming Python unicodes into an encoding, and from an encoding back to Unicode, or it may not require any interaction from SQLAlchemy at all, depending on the DBAPI in use.

  When SQLAlchemy performs the encoding/decoding, the encoding used is configured via `encoding`, which defaults to *utf-8*.

  The "convert_unicode" behavior can also be turned on for all String types by setting `sqlalchemy.engine.base.Dialect.convert_unicode` on create_engine().

  To instruct SQLAlchemy to perform Unicode encoding/decoding even on a platform that already handles Unicode natively, set convert_unicode='force'. This will incur significant performance overhead when fetching unicode result columns.

- **assert_unicode** – Deprecated. A warning is raised in all cases when a non-Unicode object is passed when SQLAlchemy would coerce into an encoding (note: but **not** when the DBAPI handles unicode objects natively). To suppress or raise this warning to an error, use the Python warnings filter documented at: http://docs.python.org/library/warnings.html

- **unicode_error** – Optional, a method to use to handle Unicode conversion errors. Behaves like the 'errors' keyword argument to the standard library's string.decode() functions. This flag requires that *convert_unicode* is set to *"force"* - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as an absolute last resort for reading strings from a column with varied or corrupted encodings, which only applies to databases that accept invalid encodings in the first place (i.e. MySQL. *not* PG, Sqlite, etc.)

class `sqlalchemy.dialects.oracle.base.`**`RAW`**(*length=None*)

Bases: `sqlalchemy.types.LargeBinary`

`__init__`(*length=None*)

Construct a LargeBinary type.

Parameters

- **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a small BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

4.7.10 cx_Oracle Notes

Support for the Oracle database via the cx_oracle driver.

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. Version 5.0 or greater is **strongly** recommended, as SQLAlchemy makes extensive use of the cx_oracle output converters for numeric and string conversions.

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`.
- *arraysize* - set the cx_oracle.arraysize value on cursors, in SQLAlchemy it defaults to 50. See the section on "LOB Objects" below.
- *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.

Unicode

cx_oracle 5 fully supports Python unicode objects. SQLAlchemy will pass all unicode strings directly to cx_oracle, and additionally uses an output handler so that all string based result values are returned as unicode as well.

Note that this behavior is disabled when Oracle 8 is detected, as it has been observed that issues remain when passing Python unicodes to cx_oracle with Oracle 8.

LOB Objects

cx_oracle returns oracle LOBs using the cx_oracle.LOB object. SQLAlchemy converts these to strings so that the interface of the Binary type is consistent with that of other backends, and so that the linkage to a live cursor is not needed in scenarios like result.fetchmany() and result.fetchall(). This means that by default, LOB objects are fully fetched unconditionally by SQLAlchemy, and the linkage to a live cursor is broken.

To disable this processing, pass `auto_convert_lobs=False` to `create_engine()`.

Two Phase Transaction Support

Two Phase transactions are implemented using XA transactions. Success has been reported with this feature but it should be regarded as experimental.

Precision Numerics

The SQLAlchemy dialect goes thorugh a lot of steps to ensure that decimal numbers are sent and received with full accuracy. An "outputtypehandler" callable is associated with each cx_oracle connection object which detects numeric types and receives them as string values, instead of receiving a Python `float` directly, which is then passed to the Python `Decimal` constructor. The `Numeric` and `Float` types under the cx_oracle dialect are aware of this behavior, and will coerce the `Decimal` to `float` if the `asdecimal` flag is `False` (default on `Float`, optional on `Numeric`).

The handler attempts to use the "precision" and "scale" attributes of the result set column to best determine if subsequent incoming values should be received as `Decimal` as opposed to int (in which case no processing is added). There are several scenarios where OCI does not provide unambiguous data as to the numeric type, including some situations where individual rows may return a combination of floating point and integer values. Certain values for "precision" and "scale" have been observed to determine this scenario. When it occurs, the outputtypehandler receives as string and then passes off to a processing function which detects, for each returned value, if a decimal point is present, and if so converts to `Decimal`, otherwise to int. The intention is that simple int-based statements like "SELECT my_seq.nextval() FROM DUAL" continue to return ints and not `Decimal` objects, and that any kind of floating point value is received as a string so that there is no floating point loss of precision.

The "decimal point is present" logic itself is also sensitive to locale. Under OCI, this is controlled by the NLS_LANG environment variable. Upon first connection, the dialect runs a test to determine the current "decimal" character, which can be a comma "," for european locales. From that point forward the outputtypehandler uses that character to represent a decimal point (this behavior is new in version 0.6.6). Note that cx_oracle 5.0.3 or greater is required when dealing with numerics with locale settings that don't use a period "." as the decimal character.

4.7.11 zxjdbc Notes

Support for the Oracle database via the zxjdbc JDBC connector.

JDBC Driver

The official Oracle JDBC driver is at http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html.

4.8 PostgreSQL

Support for the PostgreSQL database.

For information on connecting using specific drivers, see the documentation section regarding that driver.

4.8.1 Sequences/SERIAL

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

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)
    )
```

When SQLAlchemy issues a single INSERT statement, to fulfill the contract of having the "last insert identifier" available, a RETURNING clause is added to the INSERT statement which specifies the primary key columns should be returned after the statement completes. The RETURNING functionality only takes place if Postgresql 8.2 or later is in use. As a fallback approach, the sequence, whether specified explicitly or implicitly via `SERIAL`, is executed independently beforehand, the returned value to be used in the subsequent insert. Note that when an `insert()` construct is executed using "executemany" semantics, the "last inserted identifier" functionality does not apply; no RETURNING clause is emitted nor is the sequence pre-executed in this case.

To force the usage of RETURNING by default off, specify the flag `implicit_returning=False` to `create_engine()`.

4.8.2 Transaction Isolation Level

`create_engine()` accepts an `isolation_level` parameter which results in the command `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL <level>` being invoked for every new connection. Valid values for this parameter are `READ_COMMITTED`, `READ_UNCOMMITTED`, `REPEATABLE_READ`, and `SERIALIZABLE`. Note that the psycopg2 dialect does *not* use this technique and uses psycopg2-specific APIs (see that dialect for details).

4.8.3 INSERT/UPDATE...RETURNING

The dialect supports PG 8.2's `INSERT..RETURNING`, `UPDATE..RETURNING` and `DELETE..RETURNING` syntaxes. `INSERT..RETURNING` is used by default for single-row INSERT statements in order to fetch newly generated primary key identifiers. To specify an explicit `RETURNING` clause, use the `_UpdateBase.returning()` method on a per-statement basis:

```
# INSERT..RETURNING
result = table.insert().returning(table.c.col1, table.c.col2).\
    values(name='foo')
print result.fetchall()

# UPDATE..RETURNING
result = table.update().returning(table.c.col1, table.c.col2).\
    where(table.c.name=='foo').values(name='bar')
print result.fetchall()

# DELETE..RETURNING
result = table.delete().returning(table.c.col1, table.c.col2).\
    where(table.c.name=='foo')
print result.fetchall()
```

4.8.4 Indexes

PostgreSQL supports partial indexes. To create them pass a postgresql_where option to the Index constructor:

```
Index('my_index', my_table.c.id, postgresql_where=tbl.c.value > 10)
```

4.8.5 PostgreSQL Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with Postgresql are importable from the top level dialect, whether they originate from `sqlalchemy.types` or from the local dialect:

```
from sqlalchemy.dialects.postgresql import \
    ARRAY, BIGINT, BIT, BOOLEAN, BYTEA, CHAR, CIDR, DATE, \
    DOUBLE_PRECISION, ENUM, FLOAT, INET, INTEGER, INTERVAL, \
    MACADDR, NUMERIC, REAL, SMALLINT, TEXT, TIME, TIMESTAMP, \
    UUID, VARCHAR
```

Types which are specific to PostgreSQL, or have PostgreSQL-specific construction arguments, are as follows:

class `sqlalchemy.dialects.postgresql.base.`**`ARRAY`**(*item_type*, *mutable=True*, *as_tuple=False*)

Bases: `sqlalchemy.types.MutableType`, `sqlalchemy.types.Concatenable`, `sqlalchemy.types.TypeEngine`

Postgresql ARRAY type.

Represents values as Python lists.

The ARRAY type may not be supported on all DBAPIs. It is known to work on psycopg2 and not pg8000.

Note: be sure to read the notes for `MutableType` regarding ORM performance implications. The `ARRAY` type's mutability can be disabled using the "mutable" flag.

`__init__`(*item_type*, *mutable=True*, *as_tuple=False*)

Construct an ARRAY.

E.g.:

```
Column('myarray', ARRAY(Integer))
```

Arguments are:

Parameters

- **item_type** – The data type of items of this array. Note that dimensionality is irrelevant here, so multi-dimensional arrays like `INTEGER[][]`, are constructed as `ARRAY(Integer)`, not as `ARRAY(ARRAY(Integer))` or such. The type mapping figures out on the fly
- **mutable=True** – Specify whether lists passed to this class should be considered mutable. If so, generic copy operations (typically used by the ORM) will shallow-copy values.
- **as_tuple=False** – Specify whether return results should be converted to tuples from lists. DBAPIs such as psycopg2 return lists by default. When tuples are returned, the results are hashable. This flag can only be set to `True` when `mutable` is set to `False`. (new in 0.6.5)

class `sqlalchemy.dialects.postgresql.base.`**`BIT`**(**args*, ***kwargs*)

Bases: `sqlalchemy.types.TypeEngine`

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.dialects.postgresql.base.`**`BYTEA`**(*length=None*)

Bases: `sqlalchemy.types.LargeBinary`

`__init__`(*length=None*)

Construct a LargeBinary type.

Parameters

- **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a small BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class `sqlalchemy.dialects.postgresql.base.`**`CIDR`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.dialects.postgresql.base.`**`DOUBLE_PRECISION`**(*precision=None*, *asdecimal=False*, ***kwargs*)
Bases: `sqlalchemy.types.Float`

`__init__`(*precision=None*, *asdecimal=False*, ***kwargs*)
Construct a Float.

Parameters

- **precision** – the numeric precision for use in DDL `CREATE TABLE`.
- **asdecimal** – the same flag as that of `Numeric`, but defaults to `False`. Note that setting this flag to `True` results in floating point conversion.

class `sqlalchemy.dialects.postgresql.base.`**`ENUM`**(**enums*, ***kw*)
Bases: `sqlalchemy.types.Enum`

`__init__`(**enums*, ***kw*)
Construct an enum.

Keyword arguments which don't apply to a specific backend are ignored by that backend.

Parameters

- ***enums** – string or unicode enumeration labels. If unicode labels are present, the *convert_unicode* flag is auto-enabled.
- **convert_unicode** – Enable unicode-aware bind parameter and result-set processing for this Enum's data. This is set automatically based on the presence of unicode label strings.
- **metadata** – Associate this type directly with a `MetaData` object. For types that exist on the target database as an independent schema construct (Postgresql), this type will be created and dropped within `create_all()` and `drop_all()` operations. If the type is not associated with any `MetaData` object, it will associate itself with each `Table` in which it is used, and will be created when any of those individual tables are created, after a check is performed for it's existence. The type is only dropped when `drop_all()` is called for that `Table` object's metadata, however.
- **name** – The name of this type. This is required for Postgresql and any future supported database which requires an explicitly named type, or an explicitly named constraint in order to generate the type and/or a table that uses it.
- **native_enum** – Use the database's native ENUM type when available. Defaults to True. When False, uses VARCHAR + check constraint for all backends.
- **schema** – Schemaname of this type. For types that exist on the target database as an independent schema construct (Postgresql), this parameter specifies the named schema in which the type is present.
- **quote** – Force quoting to be on or off on the type's name. If left as the default of *None*, the usual schema-level "case sensitive"/"reserved name" rules are used to determine if this type's name should be quoted.

class `sqlalchemy.dialects.postgresql.base.`**`INET`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.dialects.postgresql.base.`**`INTERVAL`**(*precision=None*)
Bases: `sqlalchemy.types.TypeEngine`

Postgresql INTERVAL type.

The INTERVAL type may not be supported on all DBAPIs. It is known to work on psycopg2 and not pg8000 or zxjdbc.

`__init__`(*precision=None*)

class `sqlalchemy.dialects.postgresql.base.`**`MACADDR`**(**args*, ***kwargs*)
Bases: `sqlalchemy.types.TypeEngine`

`__init__`(**args*, ***kwargs*)

class `sqlalchemy.dialects.postgresql.base.`**`REAL`**(*precision=None*, *asdecimal=False*, ***kwargs*)
Bases: `sqlalchemy.types.Float`

`__init__`(*precision=None*, *asdecimal=False*, ***kwargs*)
Construct a Float.

Parameters

- **precision** – the numeric precision for use in DDL `CREATE TABLE`.
- **asdecimal** – the same flag as that of `Numeric`, but defaults to `False`. Note that setting this flag to `True` results in floating point conversion.

class `sqlalchemy.dialects.postgresql.base.`**`UUID`**(*as_uuid=False*)
Bases: `sqlalchemy.types.TypeEngine`

Postgresql UUID type.

Represents the UUID column type, interpreting data either as natively returned by the DBAPI or as Python uuid objects.

The UUID type may not be supported on all DBAPIs. It is known to work on psycopg2 and not pg8000.

`__init__`(*as_uuid=False*)
Construct a UUID type.

Parameters

- **as_uuid=False** – if True, values will be interpreted as Python uuid objects, converting to/from string via the DBAPI.

4.8.6 psycopg2 Notes

Support for the PostgreSQL database via the psycopg2 driver.

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.

Unicode

By default, the Psycopg2 driver uses the `psycopg2.extensions.UNICODE` extension, such that the DBAPI receives and returns all strings as Python Unicode objects directly - SQLAlchemy passes these values through without change. Note that this setting requires that the PG client encoding be set to one which can accomodate the kind of character data being passed - typically `utf-8`. If the Postgresql database is configured for `SQL_ASCII` encoding, which is often the default for PG installations, it may be necessary for non-ascii strings to be encoded into a specific encoding before being passed to the DBAPI. If changing the database's client encoding setting is not an option, specify `use_native_unicode=False` as a keyword argument to `create_engine()`, and take note of the `encoding` setting as well, which also defaults to `utf-8`. Note that disabling "native unicode" mode has a slight performance penalty, as SQLAlchemy now must translate unicode strings to/from an encoding such as utf-8, a task that is handled more efficiently within the Psycopg2 driver natively.

Connecting

URLs are of the form `postgresql+psycopg2://user:password@host:port/dbname[?key=value&key=value...`

psycopg2-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.
- *use_native_unicode* - Enable the usage of Psycopg2 "native unicode" mode per connection. True by default.

Transactions

The psycopg2 dialect fully supports SAVEPOINT and two-phase commit operations.

Transaction Isolation Level

The `isolation_level` parameter of `create_engine()` here makes use psycopg2's `set_isolation_level()` connection method, rather than issuing a `SET SESSION CHARACTERISTICS` command. This because psycopg2 resets the isolation level on each new transaction, and needs to know at the API level what level should be used.

NOTICE logging

The psycopg2 dialect will log Postgresql NOTICE messages via the `sqlalchemy.dialects.postgresql` logger:

```
import logging
logging.getLogger('sqlalchemy.dialects.postgresql').setLevel(logging.INFO)
```

Per-Statement Execution Options

The following per-statement execution options are respected:

- *stream_results* - Enable or disable usage of server side cursors for the SELECT-statement. If *None* or not set, the *server_side_cursors* option of the connection is used. If auto-commit is enabled, the option is ignored.

4.8.7 py-postgresql Notes

Support for the PostgreSQL database via py-postgresql.

Connecting

URLs are of the form `postgresql+pypostgresql://user@password@host:port/dbname[?key=value&key=valu`

4.8.8 pg8000 Notes

Support for the PostgreSQL database via the pg8000 driver.

Connecting

URLs are of the form `postgresql+pg8000://user:password@host:port/dbname[?key=value&key=value...]`.

Unicode

pg8000 requires that the postgresql client encoding be configured in the postgresql.conf file in order to use encodings other than ascii. Set this value to the same value as the "encoding" parameter on create_engine(), usually "utf-8".

Interval

Passing data from/to the Interval type is not supported as of yet.

4.8.9 zxjdbc Notes

Support for the PostgreSQL database via the zxjdbc JDBC connector.

JDBC Driver

The official Postgresql JDBC driver is at http://jdbc.postgresql.org/.

4.9 SQLite

Support for the SQLite database.

For information on connecting using a specific driver, see the documentation section regarding that driver.

4.9.1 Date and Time Types

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

4.9.2 Auto Incrementing Behavior

Background on SQLite's autoincrement is at: http://sqlite.org/autoinc.html

Two things to note:

- The AUTOINCREMENT keyword is **not** required for SQLite tables to generate primary key values automatically. AUTOINCREMENT only means that the algorithm used to generate ROWID values should be slightly different.
- SQLite does **not** generate primary key (i.e. ROWID) values, even for one column, if the table has a composite (i.e. multi-column) primary key. This is regardless of the AUTOINCREMENT keyword being present or not.

To specifically render the AUTOINCREMENT keyword on the primary key column when rendering DDL, add the flag `sqlite_autoincrement=True` to the Table construct:

```
Table('sometable', metadata,
        Column('id', Integer, primary_key=True),
        sqlite_autoincrement=True)
```

4.9.3 Transaction Isolation Level

`create_engine()` accepts an `isolation_level` parameter which results in the command `PRAGMA read_uncommitted <level>` being invoked for every new connection. Valid values for this parameter are `SERIALIZABLE` and `READ UNCOMMITTED` corresponding to a value of 0 and 1, respectively.

4.9.4 SQLite Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with SQLite are importable from the top level dialect, whether they originate from `sqlalchemy.types` or from the local dialect:

```
from sqlalchemy.dialects.sqlite import \
            BLOB, BOOLEAN, CHAR, DATE, DATETIME, DECIMAL, FLOAT, \
            INTEGER, NUMERIC, SMALLINT, TEXT, TIME, TIMESTAMP, \
            VARCHAR
```

4.9.5 Pysqlite

Support for the SQLite database via pysqlite.

Note that pysqlite is the same driver as the `sqlite3` module included with the Python distribution.

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+pysqlite:///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://')
```

Compatibility with sqlite3 "native" date and datetime types

The pysqlite driver includes the sqlite3.PARSE_DECLTYPES and sqlite3.PARSE_COLNAMES options, which have the effect of any column or expression explicitly cast as "date" or "timestamp" will be converted to a Python date or datetime object. The date and datetime types provided with the pysqlite dialect are not currently compatible with these options, since they render the ISO date/datetime including microseconds, which pysqlite's driver does not. Additionally, SQLAlchemy does not at this time automatically render the "cast" syntax required for the freestanding functions "current_timestamp" and "current_date" to return datetime/date types natively. Unfortunately, pysqlite does not provide the standard DBAPI types in `cursor.description`, leaving SQLAlchemy with no way to detect these types on the fly without expensive per-row type checks.

Keeping in mind that pysqlite's parsing option is not recommended, nor should be necessary, for use with SQLAlchemy, usage of PARSE_DECLTYPES can be forced if one configures "native_datetime=True" on create_engine():

```
engine = create_engine('sqlite://',
                connect_args={'detect_types': sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAM
                native_datetime=True
                )
```

With this flag enabled, the DATE and TIMESTAMP types (but note - not the DATETIME or TIME types...confused yet ?) will not perform any bind parameter or result processing. Execution of "func.current_date()" will return a string. "func.current_timestamp()" is registered as returning a DATETIME type in SQLAlchemy, so this function still receives SQLAlchemy-level result processing.

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.

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.

4.10 Sybase

Support for Sybase Adaptive Server Enterprise (ASE).

Note that this dialect is no longer specific to Sybase iAnywhere. ASE is the primary support platform.

4.10.1 python-sybase notes

Support for Sybase via the python-sybase driver.

http://python-sybase.sourceforge.net/

Connect strings are of the form:

```
sybase+pysybase://<username>:<password>@<dsn>/[database name]
```

Unicode Support

The python-sybase driver does not appear to support non-ASCII strings of any kind at this time.

4.10.2 pyodbc notes

Support for Sybase via pyodbc.

http://pypi.python.org/pypi/pyodbc/

Connect strings are of the form:

```
sybase+pyodbc://<username>:<password>@<dsn>/
sybase+pyodbc://<username>:<password>@<host>/<database>
```

Unicode Support

The pyodbc driver currently supports usage of these Sybase types with Unicode or multibyte strings:

```
CHAR
NCHAR
NVARCHAR
TEXT
VARCHAR
```

Currently *not* supported are:

```
UNICHAR
UNITEXT
UNIVARCHAR
```

4.10.3 mxodbc notes

Support for Sybase via mxodbc.

This dialect is a stub only and is likely non functional at this time.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *search*

PYTHON MODULE INDEX

V