SQL

Release 1.2.4

Mar 07, 2018

CONTENTS

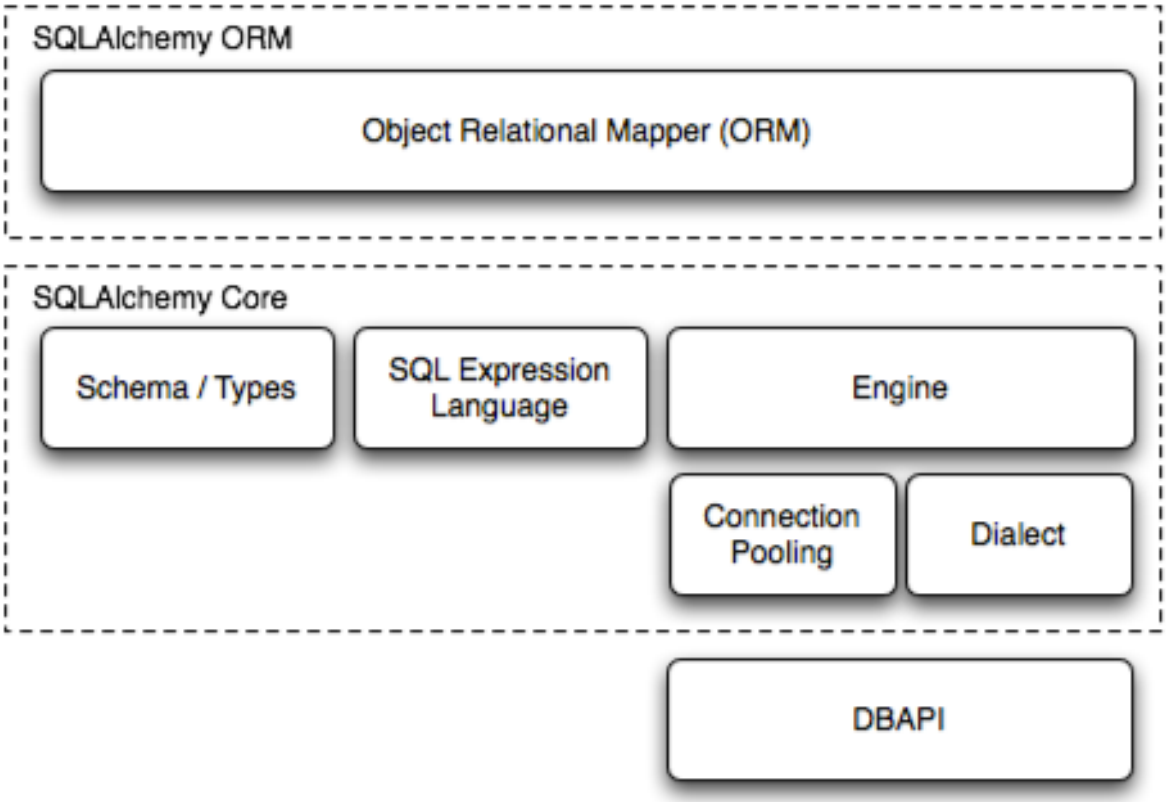
Full table of contents. For a high level overview of all documentation, see index_toplevel.

CHAPTER

ONE

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, with component dependencies organized into layers:



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.

1.1 Documentation Overview

The documentation is separated into three sections: *SQLAlchemy ORM*, *SQLAlchemy core*, and dialect_toplevel.

In *SQLAlchemy ORM*, the Object Relational Mapper is introduced and fully described. New users should begin with the ormtutorial_toplevel. 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 sqlexpression_toplevel. SQLAlchemy engine, connection, and pooling services are also described in *SQLAlchemy core*.

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

1.2 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_toplevel.

There is also a wide variety of examples involving both core SQLAlchemy constructs as well as the ORM on the wiki. See Theatrum Chemicum.

1.3 Installation Guide

1.3.1 Supported Platforms

SQLAlchemy has been tested against the following platforms:

- cPython since version 2.7, through the 2.xx series
- cPython version 3, throughout all 3.xx series
- Pypy 2.1 or greater

Changed in version 1.2: Python 2.7 is now the minimum Python version supported.

Platforms that don't currently have support include Jython and IronPython. Jython has been supported in the past and may be supported in future releases as well, depending on the state of Jython itself.

1.3.2 Supported Installation Methods

SQLAlchemy installation is via standard Python methodologies that are based on setuptools, either by referring to `setup.py` directly or by using pip or other setuptools-compatible approaches.

Changed in version 1.1: setuptools is now required by the setup.py file; plain distutils installs are no longer supported.

1.3.3 Install via pip

When `pip` is available, the distribution can be downloaded from Pypi and installed in one step:

```
pip install SQLAlchemy
```

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

In order to install the latest **prerelease** version, such as `1.2.0b1`, pip requires that the `--pre` flag be used:

```
pip install --pre SQLAlchemy
```

Where above, if the most recent version is a prerelease, it will be installed instead of the latest released version.

1.3.4 Installing using setup.py

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

```
python setup.py install
```

1.3.5 Installing the C Extensions

SQLAlchemy includes C extensions which provide an extra speed boost for dealing with result sets. The extensions are supported on both the 2.xx and 3.xx series of cPython.

`setup.py` will automatically build the extensions if an appropriate platform is detected. If the build of the C extensions fails due to a missing compiler or other issue, the setup process will output a warning message and re-run the build without the C extensions upon completion, reporting final status.

To run the build/install without even attempting to compile the C extensions, the `DISABLE_SQLALCHEMY_CEXT` environment variable may be specified. The use case for this is either for special testing circumstances, or in the rare case of compatibility/build issues not overcome by the usual "rebuild" mechanism:

```
export DISABLE_SQLALCHEMY_CEXT=1; python setup.py install
```

Changed in version 1.1: The legacy `--without-cextensions` flag has been removed from the installer as it relies on deprecated features of setuptools.

1.3.6 Installing a Database API

SQLAlchemy is designed to operate with a DBAPI implementation built for a particular database, and includes support for the most popular databases. The individual database sections in *Dialects* enumerate the available DBAPIs for each database, including external links.

1.3.7 Checking the Installed SQLAlchemy Version

This documentation covers SQLAlchemy version 1.2. 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__ # doctest: +SKIP
1.2.0
```

1.4 1.1 to 1.2 Migration

Notes on what's changed from 1.1 to 1.2 is available here at changelog/migration_12.

CHAPTER

TWO

SQLALCHEMY ORM

Here, the Object Relational Mapper is introduced and fully described. If you want to work with higher-level SQL which is constructed automatically for you, as well as automated persistence of Python objects, proceed first to the tutorial.

2.1 Object Relational Tutorial

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 sqlexpression_toplevel, 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.

2.1.1 Version Check

A quick check to verify that we are on at least **version 1.2** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
1.2.0
```

2.1.2 Connecting

For this tutorial we will use an in-memory-only SQLite database. To connect we use `create_engine()`:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:', echo=True)
```

The `echo` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard `logging` module. With it enabled, we'll see all the generated SQL produced. If you are working through this tutorial and want less output generated, set it to `False`. This tutorial will format the SQL behind a popup window so it doesn't get in our way; just click the "SQL" links to see what's being generated.

The return value of `create_engine()` is an instance of `Engine`, and it represents the core interface to the database, adapted through a dialect that handles the details of the database and DBAPI in use. In this case the SQLite dialect will interpret instructions to the Python built-in `sqlite3` module.

Lazy Connecting

The `Engine`, when first returned by `create_engine()`, has not actually tried to connect to the database yet; that happens only the first time it is asked to perform a task against the database.

The first time a method like `Engine.execute()` or `Engine.connect()` is called, the `Engine` establishes a real DBAPI connection to the database, which is then used to emit the SQL. When using the ORM, we typically don't use the `Engine` directly once created; instead, it's used behind the scenes by the ORM as we'll see shortly.

See also:

database_urls - includes examples of `create_engine()` connecting to several kinds of databases with links to more information.

2.1.3 Declare a Mapping

When using the ORM, the configurational process starts by describing the database tables we'll be dealing with, and then by defining our own classes which will be mapped to those tables. In modern SQLAlchemy, these two tasks are usually performed together, using a system known as declarative_toplevel, which allows us to create classes that include directives to describe the actual database table they will be mapped to.

Classes mapped using the Declarative system are defined in terms of a base class which maintains a catalog of classes and tables relative to that base - this is known as the **declarative base class**. Our application will usually have just one instance of this base in a commonly imported module. We create the base class using the `declarative_base()` function, as follows:

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

>>> Base = declarative_base()
```

Now that we have a "base", we can define any number of mapped classes in terms of it. We will start with just a single table called `users`, which will store records for the end-users using our application. A new class called `User` will be the class to which we map this table. Within the class, we define details about the table to which we'll be mapping, primarily the table name, and names and datatypes of columns:

```
>>> from sqlalchemy import Column, Integer, String
>>> class User(Base):
...     __tablename__ = 'users'
...
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
```

```
...     def __repr__(self):
...        return "<User(name='%s', fullname='%s', password='%s')>" % (
...                             self.name, self.fullname, self.password)
```

Tip

The `User` class defines a `__repr__()` method, but note that is **optional**; we only implement it in this tutorial so that our examples show nicely formatted `User` objects.

A class using Declarative at a minimum needs a `__tablename__` attribute, and at least one `Column` which is part of a primary key[1]. SQLAlchemy never makes any assumptions by itself about the table to which a class refers, including that it has no built-in conventions for names, datatypes, or constraints. But this doesn't mean boilerplate is required; instead, you're encouraged to create your own automated conventions using helper functions and mixin classes, which is described in detail at declarative_mixins.

When our class is constructed, Declarative replaces all the `Column` objects with special Python accessors known as descriptors; this is a process known as instrumentation. The "instrumented" mapped class will provide us with the means to refer to our table in a SQL context as well as to persist and load the values of columns from the database.

Outside of what the mapping process does to our class, the class remains otherwise mostly a normal Python class, to which we can define any number of ordinary attributes and methods needed by our application.

2.1.4 Create a Schema

With our `User` class constructed via the Declarative system, we have defined information about our table, known as table metadata. The object used by SQLAlchemy to represent this information for a specific table is called the `Table` object, and here Declarative has made one for us. We can see this object by inspecting the `__table__` attribute:

```
>>> User.__table__
Table('users', MetaData(bind=None),
            Column('id', Integer(), table=<users>, primary_key=True, nullable=False),
            Column('name', String(), table=<users>),
            Column('fullname', String(), table=<users>),
            Column('password', String(), table=<users>), schema=None)
```

Classical Mappings

The Declarative system, though highly recommended, is not required in order to use SQLAlchemy's ORM. Outside of Declarative, any plain Python class can be mapped to any `Table` using the `mapper()` function directly; this less common usage is described at classical_mapping.

When we declared our class, Declarative used a Python metaclass in order to perform additional activities once the class declaration was complete; within this phase, it then created a `Table` object according to our specifications, and associated it with the class by constructing a `Mapper` object. This object is a behind-the-scenes object we normally don't need to deal with directly (though it can provide plenty of information about our mapping when we need it).

The `Table` object is a member of a larger collection known as `MetaData`. When using Declarative, this object is available using the `.metadata` attribute of our declarative base class.

The `MetaData` is a registry which includes the ability to emit a limited set of schema generation commands to the database. As our SQLite database does not actually have a `users` table present, we can use

[1] For information on why a primary key is required, see faq_mapper_primary_key.

MetaData to issue CREATE TABLE statements to the database for all tables that don't yet exist. Below, we call the MetaData.create_all() method, passing in our Engine as a source of database connectivity. We will see that special commands are first emitted to check for the presence of the users table, and following that the actual CREATE TABLE statement:

```
>>> Base.metadata.create_all(engine)
SELECT ...
PRAGMA table_info("users")
()
CREATE TABLE users (
    id INTEGER NOT NULL, name VARCHAR,
    fullname VARCHAR,
    password VARCHAR,
    PRIMARY KEY (id)
)
()
COMMIT
```

Minimal Table Descriptions vs. Full Descriptions

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(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(Integer, Sequence('user_id_seq'), primary_key=True)
```

A full, foolproof Table generated via our declarative mapping is therefore:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, Sequence('user_id_seq'), primary_key=True)
    name = Column(String(50))
    fullname = Column(String(50))
    password = Column(String(12))

    def __repr__(self):
        return "<User(name='%s', fullname='%s', password='%s')>" % (
                                self.name, self.fullname, self.password)
```

We include this more verbose table definition separately to highlight the difference between a minimal construct geared primarily towards in-Python usage only, versus one that will be used to emit CREATE TABLE statements on a particular set of backends with more stringent requirements.

2.1.5 Create an Instance of the Mapped Class

With mappings complete, let's now create and inspect a User object:

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> ed_user.name
```

```
'ed'
>>> ed_user.password
'edspassword'
>>> str(ed_user.id)
'None'
```

the `__init__()` method

Our `User` class, as defined using the Declarative system, has been provided with a constructor (e.g. `__init__()` method) which automatically accepts keyword names that match the columns we've mapped. We are free to define any explicit `__init__()` method we prefer on our class, which will override the default method provided by Declarative.

Even though we didn't specify it in the constructor, the `id` attribute still produces a value of `None` when we access it (as opposed to Python's usual behavior of raising `AttributeError` for an undefined attribute). SQLAlchemy's instrumentation normally produces this default value for column-mapped attributes when first accessed. For those attributes where we've actually assigned a value, the instrumentation system is tracking those assignments for use within an eventual INSERT statement to be emitted to the database.

2.1.6 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
```

Session Lifecycle Patterns

The question of when to make a `Session` depends a lot on what kind of application is being built. Keep in mind, the `Session` is just a workspace for your objects, local to a particular database connection - if you think of an application thread as a guest at a dinner party, the `Session` is the guest's plate and the objects it holds are the food (and the database...the kitchen?)! More on this topic available at session_faq_whentocreate.

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-enabled `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.

2.1.7 Adding and Updating Objects

To persist our `User` object, we `add()` it to our `Session`:

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> session.add(ed_user)
```

At this point, we say that the instance is **pending**; no SQL has yet been issued and the object is not yet represented by a row in the database. 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 immediately thereafter.

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:

```
{sql}>>> our_user = session.query(User).filter_by(name='ed').first() # doctest:+NORMALIZE_
↪WHITESPACE
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,
        users.password AS users_password
FROM users
WHERE users.name = ?
 LIMIT ? OFFSET ?
('ed', 1, 0)
{stop}>>> our_user
<User(name='ed', fullname='Ed Jones', password='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(name='wendy', fullname='Wendy Williams', password='foobar'),
...     User(name='mary', fullname='Mary Contrary', password='xxg527'),
...     User(name='fred', fullname='Fred Flinstone', password='blah')])
```

Also, we've decided the password for Ed 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(name='ed', fullname='Ed Jones', password='f8s7ccs')>])
```

and that three new `User` objects are pending:

```
>>> session.new  # doctest: +SKIP
IdentitySet([<User(name='wendy', fullname='Wendy Williams', password='foobar')>,
<User(name='mary', fullname='Mary Contrary', password='xxg527')>,
<User(name='fred', fullname='Fred Flinstone', password='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()`. The `Session` emits the `UPDATE` statement for the password change on "ed", as well as `INSERT` statements for the three new `User` objects we've added:

```
{sql}>>> 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 the remaining changes 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:

```
{sql}>>> ed_user.id # doctest: +NORMALIZE_WHITESPACE
BEGIN (implicit)
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.id = ?
(1,)
{stop}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 *Using the Session*.

Session Object States

As our `User` object moved from being outside the `Session`, to inside the `Session` without a primary key, to actually being inserted, it moved between three out of four available "object states" - **transient**, **pending**, and **persistent**. Being aware of these states and what they mean is always a good idea - be sure to read session_object_states for a quick overview.

2.1.8 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(name='fakeuser', fullname='Invalid', password='12345')
>>> session.add(fake_user)
```

Querying the session, we can see that they're flushed into the current transaction:

```
{sql}>>> 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,
        users.password AS users_password
FROM users
WHERE users.name IN (?, ?)
('Edwardo', 'fakeuser')
{stop}[<User(name='Edwardo', fullname='Ed Jones', password='f8s7ccs')>, <User(name='fakeuser',␣
↪fullname='Invalid', password='12345')>]
```

Rolling back, we can see that `ed_user`'s name is back to `ed`, and `fake_user` has been kicked out of the session:

```
{sql}>>> session.rollback()
ROLLBACK
{stop}

{sql}>>> ed_user.name
BEGIN (implicit)
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.id = ?
(1,)
{stop}u'ed'
>>> fake_user in session
False
```

issuing a SELECT illustrates the changes made to the database:

```
{sql}>>> 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,
        users.password AS users_password
FROM users
WHERE users.name IN (?, ?)
('ed', 'fakeuser')
{stop}[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

2.1.9 Querying

A `Query` object is created using the `query()` method 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:

```
{sql}>>> 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
()
{stop}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:

```
{sql}>>> 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
()
{stop}ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

The tuples returned by `Query` are *named* tuples, supplied by the `KeyedTuple` class, 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:

```
{sql}>>> 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,
        users.password AS users_password
FROM users
()
{stop}<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> ed
<User(name='wendy', fullname='Wendy Williams', password='foobar')> wendy
<User(name='mary', fullname='Mary Contrary', password='xxg527')> mary
<User(name='fred', fullname='Fred Flinstone', password='blah')> fred
```

You can control the names of individual column expressions using the `label()` construct, which is available from any `ColumnElement`-derived object, as well as any class attribute which is mapped to one (such as `User.name`):

```
{sql}>>> for row in session.query(User.name.label('name_label')).all():
...    print(row.name_label)
SELECT users.name AS name_label
FROM users
(){stop}
ed
wendy
mary
fred
```

The name given to a full entity such as `User`, assuming that multiple entities are present in the call to

query(), can be controlled using aliased() :

```
>>> from sqlalchemy.orm import aliased
>>> user_alias = aliased(User, name='user_alias')

{sql}>>> for row in session.query(user_alias, user_alias.name).all():
...     print(row.user_alias)
SELECT user_alias.id AS user_alias_id,
        user_alias.name AS user_alias_name,
        user_alias.fullname AS user_alias_fullname,
        user_alias.password AS user_alias_password
FROM users AS user_alias
(){stop}
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
<User(name='wendy', fullname='Wendy Williams', password='foobar')>
<User(name='mary', fullname='Mary Contrary', password='xxg527')>
<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

Basic operations with Query include issuing LIMIT and OFFSET, most conveniently using Python array slices and typically in conjunction with ORDER BY:

```
{sql}>>> 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,
        users.password AS users_password
FROM users ORDER BY users.id
LIMIT ? OFFSET ?
(2, 1){stop}
<User(name='wendy', fullname='Wendy Williams', password='foobar')>
<User(name='mary', fullname='Mary Contrary', password='xxg527')>
```

and filtering results, which is accomplished either with filter_by(), which uses keyword arguments:

```
{sql}>>> 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',)
{stop}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:

```
{sql}>>> 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',)
{stop}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:

```
{sql}>>> 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,
        users.password AS users_password
FROM users
WHERE users.name = ? AND users.fullname = ?
('ed', 'Ed Jones')
{stop}<User(name='ed', fullname='Ed Jones', password='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%'))
```

> **Note:** `ColumnOperators.like()` renders the LIKE operator, which is case insensitive on some backends, and case sensitive on others. For guaranteed case-insensitive comparisons, use `ColumnOperators.ilike()`.

- `ILIKE` (case-insensitive LIKE):

```
query.filter(User.name.ilike('%ed%'))
```

> **Note:** most backends don't support ILIKE directly. For those, the `ColumnOperators.ilike()` operator renders an expression combining LIKE with the LOWER SQL function applied to each operand.

- `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`:

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

# alternatively, if pep8/linters are a concern
query.filter(User.name.is_(None))
```

- `IS NOT NULL`:

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

# alternatively, if pep8/linters are a concern
query.filter(User.name.isnot(None))
```

- AND:

```
# use and_()
from sqlalchemy import and_
query.filter(and_(User.name == 'ed', User.fullname == 'Ed Jones'))

# or send multiple expressions to .filter()
query.filter(User.name == 'ed', User.fullname == 'Ed Jones')

# or chain multiple filter()/filter_by() calls
query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')
```

Note: Make sure you use `and_()` and **not** the Python `and` operator!

- OR:

```
from sqlalchemy import or_
query.filter(or_(User.name == 'ed', User.name == 'wendy'))
```

Note: Make sure you use `or_()` and **not** the Python `or` operator!

- MATCH:

```
query.filter(User.name.match('wendy'))
```

Note: `match()` uses a database-specific `MATCH` or `CONTAINS` function; its behavior will vary by backend and is not available on some backends such as SQLite.

Returning Lists and Scalars

A number of methods on `Query` immediately issue SQL and return a value containing loaded database results. Here's a brief tour:

- `all()` returns a list:

```
>>> query = session.query(User).filter(User.name.like('%ed')).order_by(User.id)
{sql}>>> query.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 LIKE ? ORDER BY users.id
('%ed',)
{stop}[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>,
      <User(name='fred', fullname='Fred Flinstone', password='blah')>]
```

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

```
{sql}>>> query.first()
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 LIKE ? ORDER BY users.id
 LIMIT ? OFFSET ?
('%ed', 1, 0)
{stop}<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

- `one()` fully fetches all rows, and if not exactly one object identity or composite row is present in the result, raises an error. With multiple rows found:

```
>>> user = query.one()
Traceback (most recent call last):
...
MultipleResultsFound: Multiple rows were found for one()
```

With no rows found:

```
>>> user = query.filter(User.id == 99).one()
Traceback (most recent call last):
...
NoResultFound: No row was found for one()
```

The `one()` method is great for systems that expect to handle "no items found" versus "multiple items found" differently; such as a RESTful web service, which may want to raise a "404 not found" when no results are found, but raise an application error when multiple results are found.

- `one_or_none()` is like `one()`, except that if no results are found, it doesn't raise an error; it just returns `None`. Like `one()`, however, it does raise an error if multiple results are found.
- `scalar()` invokes the `one()` method, and upon success returns the first column of the row:

```
>>> query = session.query(User.id).filter(User.name == 'ed').\
...    order_by(User.id)
{sql}>>> query.scalar()
SELECT users.id AS users_id
FROM users
WHERE users.name = ? ORDER BY users.id
('ed',)
{stop}1
```

Using Textual SQL

Literal strings can be used flexibly with `Query`, by specifying their use with the `text()` construct, which is accepted by most applicable methods. For example, `filter()` and `order_by()`:

```
>>> from sqlalchemy import text
{sql}>>> for user in session.query(User).\
...             filter(text("id<224")).\
...             order_by(text("id")).all():
...     print(user.name)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE id<224 ORDER BY id
()
```

```
{stop}ed
wendy
mary
fred
```

Bind parameters can be specified with string-based SQL, using a colon. To specify the values, use the `params()` method:

```
{sql}>>> session.query(User).filter(text("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,
        users.password AS users_password
FROM users
WHERE id<? and name=? ORDER BY users.id
(224, 'fred')
{stop}<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

To use an entirely string-based statement, a `text()` construct representing a complete statement can be passed to `from_statement()`. Without additional specifiers, the columns in the string SQL are matched to the model columns based on name, such as below where we use just an asterisk to represent loading all columns:

```
{sql}>>> session.query(User).from_statement(
...                     text("SELECT * FROM users where name=:name")).\
...                     params(name='ed').all()
SELECT * FROM users where name=?
('ed',)
{stop}[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

Matching columns on name works for simple cases but can become unwieldy when dealing with complex statements that contain duplicate column names or when using anonymized ORM constructs that don't easily match to specific names. Additionally, there is typing behavior present in our mapped columns that we might find necessary when handling result rows. For these cases, the `text()` construct allows us to link its textual SQL to Core or ORM-mapped column expressions positionally; we can achieve this by passing column expressions as positional arguments to the `TextClause.columns()` method:

```
>>> stmt = text("SELECT name, id, fullname, password "
...             "FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id, User.fullname, User.password)
{sql}>>> session.query(User).from_statement(stmt).params(name='ed').all()
SELECT name, id, fullname, password FROM users where name=?
('ed',)
{stop}[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

New in version 1.1: The `TextClause.columns()` method now accepts column expressions which will be matched positionally to a plain text SQL result set, eliminating the need for column names to match or even be unique in the SQL statement.

When selecting from a `text()` construct, the `Query` may still specify what columns and entities are to be returned; instead of `query(User)` we can also ask for the columns individually, as in any other case:

```
>>> stmt = text("SELECT name, id FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id)
{sql}>>> session.query(User.id, User.name).\
...           from_statement(stmt).params(name='ed').all()
SELECT name, id FROM users where name=?
('ed',)
{stop}[(1, u'ed')]
```

See also:

sqlexpression_text - The `text()` construct explained from the perspective of Core-only queries.

Counting

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

```
{sql}>>> session.query(User).filter(User.name.like('%ed')).count()
SELECT count(*) AS count_1
FROM (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 LIKE ?) AS anon_1
('%ed',)
{stop}2
```

Counting on `count()`

`Query.count()` used to be a very complicated method when it would try to guess whether or not a subquery was needed around the existing query, and in some exotic cases it wouldn't do the right thing. Now that it uses a simple subquery every time, it's only two lines long and always returns the right answer. Use `func.count()` if a particular statement absolutely cannot tolerate the subquery being present.

The `count()` method is used to determine how many rows the SQL statement would return. Looking at the generated SQL above, SQLAlchemy always places whatever it is we are querying into a subquery, then counts the rows from that. In some cases this can be reduced to a simpler `SELECT count(*) FROM table`, however modern versions of SQLAlchemy don't try to guess when this is appropriate, as the exact SQL can be emitted using more explicit means.

For situations where the "thing to be counted" needs to be indicated specifically, we can specify the "count" function directly using the expression `func.count()`, available from the `func` construct. Below we use it to return the count of each distinct user name:

```
>>> from sqlalchemy import func
{sql}>>> 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
()
{stop}[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]
```

To achieve our simple `SELECT count(*) FROM table`, we can apply it as:

```
{sql}>>> session.query(func.count('*')).select_from(User).scalar()
SELECT count(?) AS count_1
FROM users
('*',)
{stop}4
```

The usage of `select_from()` can be removed if we express the count in terms of the `User` primary key directly:

```
{sql}>>> session.query(func.count(User.id)).scalar()
SELECT count(users.id) AS count_1
FROM users
()
{stop}4
```

2.1.10 Building a Relationship

Let's consider how a second table, related to `User`, can be mapped and queried. Users in our system can store any number of email addresses associated with their username. This implies a basic one to many association from the `users` 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

>>> 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", back_populates="addresses")
...
...     def __repr__(self):
...         return "<Address(email_address='%s')>" % self.email_address

>>> User.addresses = relationship(
...     "Address", order_by=Address.id, back_populates="user")
```

The above class introduces the `ForeignKey` construct, which is a directive applied to `Column` that indicates that values in this column should be constrained to be values present in the named remote column. This is a core feature of relational databases, and is the "glue" that transforms an otherwise unconnected collection of tables to have rich overlapping relationships. The `ForeignKey` above expresses that values in the `addresses.user_id` column should be constrained to those values in the `users.id` column, i.e. its primary key.

A second directive, known as `relationship()`, tells the ORM that the `Address` class itself should be linked to the `User` class, using the attribute `Address.user`. `relationship()` uses the foreign key relationships between the two tables to determine the nature of this linkage, determining that `Address.user` will be many to one. An additional `relationship()` directive is placed on the `User` mapped class under the attribute `User.addresses`. In both `relationship()` directives, the parameter `relationship.back_populates` is assigned to refer to the complementary attribute names; by doing so, each `relationship()` can make intelligent decision about the same relationship as expressed in reverse; on one side, `Address.user` refers to a `User` instance, and on the other side, `User.addresses` refers to a list of `Address` instances.

Note: The `relationship.back_populates` parameter is a newer version of a very common SQLAlchemy feature called `relationship.backref`. The `relationship.backref` parameter hasn't gone anywhere and will always remain available! The `relationship.back_populates` is the same thing, except a little more verbose and easier to manipulate. For an overview of the entire topic, see the section relationships_backref.

The reverse side of a many-to-one relationship is always one to many. A full catalog of available `relationship()` configurations is at relationship_patterns.

The two complementing relationships `Address.user` and `User.addresses` are referred to as a bidirectional relationship, and is a key feature of the SQLAlchemy ORM. The section relationships_backref discusses the "backref" feature in detail.

Arguments to `relationship()` which concern the remote class can be specified using strings, assuming the Declarative system is in use. Once all mappings are complete, these strings are evaluated as Python expressions in order to produce the actual argument, in the above case the `User` class. The names which are allowed during this evaluation include, among other things, the names of all classes which have been created in terms of the declared base.

See the docstring for `relationship()` for more detail on argument style.

Did you know ?

- a FOREIGN KEY constraint in most (though not all) relational databases can only link to a primary key column, or a column that has a UNIQUE constraint.
- a FOREIGN KEY constraint that refers to a multiple column primary key, and itself has multiple columns, is known as a "composite foreign key". It can also reference a subset of those columns.
- FOREIGN KEY columns can automatically update themselves, in response to a change in the referenced column or row. This is known as the CASCADE *referential action*, and is a built in function of the relational database.
- FOREIGN KEY can refer to its own table. This is referred to as a "self-referential" foreign key.
- Read more about foreign keys at Foreign Key - Wikipedia.

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:

```
{sql}>>> Base.metadata.create_all(engine)
PRAGMA...
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
```

2.1.11 Working with Related Objects

Now when we create a `User`, a blank `addresses` collection will be present. Various collection types, such as sets and dictionaries, are possible here (see custom_collections for details), but by default, the collection is a Python list.

```
>>> jack = User(name='jack', fullname='Jack Bean', password='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@yahoo.com')]
```

When using a bidirectional relationship, elements added in one direction automatically become visible in the other direction. This behavior occurs based on attribute on-change events and is evaluated in Python, without using any SQL:

```
>>> jack.addresses[1]
<Address(email_address='j25@yahoo.com')>

>>> jack.addresses[1].user
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

Let's add and commit `Jack Bean` to the database. `jack` as well as the two `Address` members in the corresponding `addresses` collection are both added to the session at once, using a process known as **cascading**:

```
>>> session.add(jack)
{sql}>>> 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:

```
{sql}>>> 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,
        users.password AS users_password
FROM users
WHERE users.name = ?
('jack',)

{stop}>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

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

```
{sql}>>> 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 ORDER BY addresses.id
(5,)
{stop}[<Address(email_address='jack@google.com')>, <Address(email_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. We'll cover ways to optimize the loading of this collection in a bit.

2.1.12 Querying with Joins

Now that we have two tables, we can show some more features of `Query`, specifically how to create queries that deal with both tables at the same time. The Wikipedia page on SQL JOIN offers a good introduction to join techniques, several of which we'll illustrate here.

To construct a simple implicit join between `User` and `Address`, we can use `Query.filter()` to equate their related columns together. Below we load the `User` and `Address` entities at once using this method:

```
{sql}>>> for u, a in session.query(User, Address).\
...                     filter(User.id==Address.user_id).\
...                     filter(Address.email_address=='jack@google.com').\
...                     all():
...     print(u)
...     print(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',)
{stop}<User(name='jack', fullname='Jack Bean', password='gjffdd')>
<Address(email_address='jack@google.com')>
```

The actual SQL JOIN syntax, on the other hand, is most easily achieved using the `Query.join()` method:

```
{sql}>>> 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,
        users.password AS users_password
FROM users JOIN addresses ON users.id = addresses.user_id
WHERE addresses.email_address = ?
('jack@google.com',)
{stop}[<User(name='jack', fullname='Jack Bean', password='gjffdd')>]
```

`Query.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, `Query.join()` works better when one of the following forms are used:

```
query.join(Address, User.id==Address.user_id)    # explicit condition
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
```

As you would expect, the same idea is used for "outer" joins, using the `outerjoin()` function:

```
query.outerjoin(User.addresses)   # LEFT OUTER JOIN
```

The reference documentation for `join()` contains detailed information and examples of the calling styles accepted by this method; `join()` is an important method at the center of usage for any SQL-fluent application.

What does `Query` select from if there's multiple entities?

The `Query.join()` method will **typically join from the leftmost item** in the list of entities, when the ON clause is omitted, or if the ON clause is a plain SQL expression. To control the first entity in the list of JOINs, use the `Query.select_from()` method:

```
query = session.query(User, Address).select_from(Address).join(User)
```

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)
{sql}>>> for username, email1, email2 in \
...     session.query(User.name, adalias1.email_address, adalias2.email_address).\
...     join(adalias1, User.addresses).\
...     join(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')
{stop}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_count
    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 sqlexpression_toplevel:

```
>>> from sqlalchemy.sql import func
>>> stmt = session.query(Address.user_id, func.count('*').\
...         label('address_count')).\
...         group_by(Address.user_id).subquery()
```

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`:

```
{sql}>>> 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
('*',)
{stop}<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> None
<User(name='wendy', fullname='Wendy Williams', password='foobar')> None
<User(name='mary', fullname='Mary Contrary', password='xxg527')> None
<User(name='fred', fullname='Fred Flinstone', password='blah')> None
<User(name='jack', fullname='Jack Bean', password='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:

```
{sql}>>> stmt = session.query(Address).\
...                 filter(Address.email_address != 'j25@yahoo.com').\
...                 subquery()
>>> adalias = aliased(Address, stmt)
>>> for user, address in session.query(User, adalias).\
...         join(adalias, User.addresses):
...     print(user)
...     print(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,
            addresses.user_id AS user_id
    FROM addresses
    WHERE addresses.email_address != ?) AS anon_1
    ON users.id = anon_1.user_id
('j25@yahoo.com',)
{stop}<User(name='jack', fullname='Jack Bean', password='gjffdd')>
<Address(email_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)
{sql}>>> 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)
```

```
()
{stop}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()`:

```
{sql}>>> 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)
()
{stop}jack
```

`any()` takes criterion as well, to limit the rows matched:

```
{sql}>>> 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%',)
{stop}jack
```

`has()` is the same operator as `any()` for many-to-one relationships (note the `~` operator here too, which means "NOT"):

```
{sql}>>> 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',)
{stop}[]
```

Common Relationship Operators

Here's all the operators which build on relationships - each one is linked to its API documentation which includes full details on usage and behavior:

- `__eq__()` (many-to-one "equals" comparison):

  ```
  query.filter(Address.user == someuser)
  ```

- `__ne__()` (many-to-one "not equals" comparison):

  ```
  query.filter(Address.user != someuser)
  ```

- IS NULL (many-to-one comparison, also uses `__eq__()`):

```
query.filter(Address.user == None)
```

- `contains()` (used for one-to-many collections):

```
query.filter(User.addresses.contains(someaddress))
```

- `any()` (used for 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 scalar references):

```
query.filter(Address.user.has(name='ed'))
```

- `Query.with_parent()` (used for any relationship):

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

2.1.13 Eager Loading

Recall earlier that we illustrated a lazy loading operation, when we accessed the `User.addresses` collection of a `User` and SQL was emitted. If you want to reduce the number of queries (dramatically, in many cases), we can apply an eager load to the query operation. SQLAlchemy offers three types of eager loading, two of which are automatic, and a third which involves custom criterion. All three are usually invoked via functions known as query options which give additional instructions to the `Query` on how we would like various attributes to be loaded, via the `Query.options()` method.

Subquery Load

In this case we'd like to indicate that `User.addresses` should load eagerly. A good choice for loading a set of objects as well as their related collections is the `orm.subqueryload()` option, which emits a second SELECT statement that fully loads the collections associated with the results just loaded. The name "subquery" originates from the fact that the SELECT statement constructed directly via the `Query` is re-used, embedded as a subquery into a SELECT against the related table. This is a little elaborate but very easy to use:

```
>>> from sqlalchemy.orm import subqueryload
{sql}>>> jack = session.query(User).\
...                 options(subqueryload(User.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
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',)
```

```
{stop}>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>

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

Note: `subqueryload()` when used in conjunction with limiting such as `Query.first()`, `Query.limit()` or `Query.offset()` should also include `Query.order_by()` on a unique column in order to ensure correct results. See subqueryload_ordering.

Joined Load

The other automatic eager loading function is more well known and is called `orm.joinedload()`. This style of loading emits a JOIN, by default a LEFT OUTER JOIN, so that the lead object as well as the related object or collection is loaded in one step. We illustrate loading the same `addresses` collection in this way - note that even though the `User.addresses` collection on `jack` is actually populated right now, the query will emit the extra join regardless:

```
>>> from sqlalchemy.orm import joinedload

{sql}>>> jack = session.query(User).\
...                         options(joinedload(User.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_address 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',)

{stop}>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>

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

Note that even though the OUTER JOIN resulted in two rows, we still only got one instance of `User` back. This is because `Query` applies a "uniquing" strategy, based on object identity, to the returned entities. This is specifically so that joined eager loading can be applied without affecting the query results.

While `joinedload()` has been around for a long time, `subqueryload()` is a newer form of eager loading. `subqueryload()` tends to be more appropriate for loading related collections while `joinedload()` tends to be better suited for many-to-one relationships, due to the fact that only one row is loaded for both the lead and the related object.

`joinedload()` is not a replacement for `join()`

The join created by `joinedload()` is anonymously aliased such that it **does not affect the query results**. An `Query.order_by()` or `Query.filter()` call **cannot** reference these aliased tables - so-called "user space" joins are constructed using `Query.join()`. The rationale for this is that `joinedload()` is only applied in order to affect how related objects or collections are loaded as an optimizing detail -

it can be added or removed with no impact on actual results. See the section zen_of_eager_loading for a detailed description of how this is used.

Explicit Join + Eagerload

A third style of eager loading is when we are constructing a JOIN explicitly in order to locate the primary rows, and would like to additionally apply the extra table to a related object or collection on the primary object. This feature is supplied via the `orm.contains_eager()` function, and is most typically useful for pre-loading the many-to-one object on a query that needs to filter on that same object. Below we illustrate loading an `Address` row as well as the related `User` object, filtering on the `User` named "jack" and using `orm.contains_eager()` to apply the "user" columns to the `Address.user` attribute:

```
>>> from sqlalchemy.orm import contains_eager
{sql}>>> jacks_addresses = session.query(Address).\
...                             join(Address.user).\
...                             filter(User.name=='jack').\
...                             options(contains_eager(Address.user)).\
...                             all()
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',)

{stop}>>> jacks_addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]

>>> jacks_addresses[0].user
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

For more information on eager loading, including how to configure various forms of loading by default, see the section *Relationship Loading Techniques*.

2.1.14 Deleting

Let's try to delete `jack` and see how that goes. We'll mark the object as deleted in the session, then we'll issue a `count` query to see that no rows remain:

```
>>> session.delete(jack)
{sql}>>> session.query(User).filter_by(name='jack').count()
UPDATE addresses SET user_id=? WHERE addresses.id = ?
((None, 1), (None, 2))
DELETE FROM users WHERE users.id = ?
(5,)
SELECT count(*) AS count_1
FROM (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 = ?) AS anon_1
('jack',)
{stop}0
```

So far, so good. How about Jack's `Address` objects ?

```
{sql}>>> session.query(Address).filter(
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
...  ).count()
SELECT count(*) AS count_1
FROM (SELECT addresses.id AS addresses_id,
                addresses.email_address AS addresses_email_address,
                addresses.user_id AS addresses_user_id
FROM addresses
WHERE addresses.email_address IN (?, ?)) AS anon_1
('jack@google.com', 'j25@yahoo.com')
{stop}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 - we'll close the `Session`:

```
>>> session.close()
ROLLBACK
```

and use a new `declarative_base()`:

```
>>> Base = declarative_base()
```

Next we'll declare the `User` class, adding in the `addresses` relationship including the cascade configuration (we'll leave the constructor out too):

```
>>> class User(Base):
...     __tablename__ = 'users'
...
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
...     addresses = relationship("Address", back_populates='user',
...                     cascade="all, delete, delete-orphan")
...
...     def __repr__(self):
...        return "<User(name='%s', fullname='%s', password='%s')>" % (
...                                self.name, self.fullname, self.password)
```

Then we recreate `Address`, noting that in this case we've created the `Address.user` relationship via the `User` class already:

```
>>> 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", back_populates="addresses")
...
...     def __repr__(self):
...         return "<Address(email_address='%s')>" % self.email_address
```

Now when we load the user `jack` (below using `get()`, which loads by primary key), removing an address from the corresponding `addresses` collection will result in that `Address` being deleted:

```
# load Jack by primary key
{sql}>>> jack = session.query(User).get(5)
BEGIN (implicit)
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.id = ?
(5,)
{stop}

# remove one Address (lazy load fires off)
{sql}>>> del jack.addresses[1]
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,)
{stop}

# only one address remains
{sql}>>> session.query(Address).filter(
...      Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
DELETE FROM addresses WHERE addresses.id = ?
(2,)
SELECT count(*) AS count_1
FROM (SELECT addresses.id AS addresses_id,
                addresses.email_address AS addresses_email_address,
                addresses.user_id AS addresses_user_id
FROM addresses
WHERE addresses.email_address IN (?, ?)) AS anon_1
('jack@google.com', 'j25@yahoo.com')
{stop}1
```

Deleting Jack will delete both Jack and the remaining `Address` associated with the user:

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

{sql}>>> session.query(User).filter_by(name='jack').count()
DELETE FROM addresses WHERE addresses.id = ?
(1,)
DELETE FROM users WHERE users.id = ?
(5,)
SELECT count(*) AS count_1
FROM (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 = ?) AS anon_1
('jack',)
{stop}0

{sql}>>> session.query(Address).filter(
...    Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
```

```
... ).count()
SELECT count(*) AS count_1
FROM (SELECT addresses.id AS addresses_id,
                addresses.email_address AS addresses_email_address,
                addresses.user_id AS addresses_user_id
FROM addresses
WHERE addresses.email_address IN (?, ?)) AS anon_1
('jack@google.com', 'j25@yahoo.com')
{stop}0
```

More on Cascades

Further detail on configuration of cascades is at unitofwork_cascades. The cascade functionality can also integrate smoothly with the `ON DELETE CASCADE` functionality of the relational database. See passive_deletes for details.

2.1.15 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.

For a plain many-to-many, we need to create an un-mapped `Table` construct to serve as the association table. This looks like the following:

```
>>> from sqlalchemy import Table, Text
>>> # association table
>>> post_keywords = Table('post_keywords', Base.metadata,
...     Column('post_id', ForeignKey('posts.id'), primary_key=True),
...     Column('keyword_id', ForeignKey('keywords.id'), primary_key=True)
... )
```

Above, we can see declaring a `Table` directly is a little different than declaring a mapped class. `Table` is a constructor function, so each individual `Column` argument is separated by a comma. The `Column` object is also given its name explicitly, rather than it being taken from an assigned attribute name.

Next we define `BlogPost` and `Keyword`, using complementary `relationship()` constructs, each referring to the `post_keywords` table as an association table:

```
>>> 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,
...                             back_populates='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)
...     posts = relationship('BlogPost',
...                          secondary=post_keywords,
...                          back_populates='keywords')
...
...     def __init__(self, keyword):
...         self.keyword = keyword
```

Note: The above class declarations illustrate explicit `__init__()` methods. Remember, when using Declarative, it's optional!

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_pattern.

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:

```
>>> BlogPost.author = relationship(User, back_populates="posts")
>>> User.posts = relationship(BlogPost, back_populates="author", lazy="dynamic")
```

Create new tables:

```
{sql}>>> Base.metadata.create_all(engine)
PRAGMA...
CREATE TABLE keywords (
    id INTEGER NOT NULL,
    keyword VARCHAR(50) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (keyword)
)
()
COMMIT
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 post_keywords (
    post_id INTEGER NOT NULL,
    keyword_id INTEGER NOT NULL,
    PRIMARY KEY (post_id, keyword_id),
    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:

```
{sql}>>> 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,
        users.password AS users_password
FROM users
WHERE users.name = ?
('wendy',)
{stop}
>>> 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'":

```
{sql}>>> 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 (?, ?)
(...)
SELECT posts.id AS posts_id,
        posts.user_id AS posts_user_id,
        posts.headline AS posts_headline,
        posts.body AS posts_body
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 keywords.keyword = ?)
('firstpost',)
{stop}[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy␣
↪Williams', password='foobar')>)]
```

If we want to look up posts owned by the user `wendy`, we can tell the query to narrow down to that `User` object as a parent:

```
{sql}>>> 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_headline,
        posts.body AS posts_body
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 keywords.keyword = ?))
(2, 'firstpost')
{stop}[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy␣
↪Williams', password='foobar')>)]
```

Or we can use Wendy's own `posts` relationship, which is a "dynamic" relationship, to query straight from there:

```
{sql}>>> 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_headline,
        posts.body AS posts_body
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 keywords.keyword = ?))
(2, 'firstpost')
{stop}[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy␣
↪Williams', password='foobar')>)]
```

2.1.16 Further Reference

Query Reference: query_api_toplevel

Mapper Reference: mapper_config_toplevel

Relationship Reference: relationship_config_toplevel

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 ormtutorial_toplevel and know how to construct and use rudimentary mappers and relationships.

2.2.1 Types of Mappings

Modern SQLAlchemy features two distinct styles of mapper configuration. The "Classical" style is SQLAlchemy's original mapping API, whereas "Declarative" is the richer and more succinct system that builds on top of "Classical". Both styles may be used interchangeably, as the end result of each is exactly the same - a user-defined class mapped by the `mapper()` function onto a selectable unit, typically a `Table`.

Declarative Mapping

The *Declarative Mapping* is the typical way that mappings are constructed in modern SQLAlchemy. Making use of the declarative_toplevel system, the components of the user-defined class as well as the `Table` metadata to which the class is mapped are defined at once:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, ForeignKey

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)
```

Above, a basic single-table mapping with four columns. Additional attributes, such as relationships to other mapped classes, are also declared inline within the class definition:

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

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

    addresses = relationship("Address", backref="user", order_by="Address.id")

class Address(Base):
    __tablename__ = 'address'

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

The declarative mapping system is introduced in the ormtutorial_toplevel. For additional details on how this system works, see declarative_toplevel.

Classical Mappings

A *Classical Mapping* refers to the configuration of a mapped class using the `mapper()` function, without using the Declarative system. This is SQLAlchemy's original class mapping API, and is still the base mapping system provided by the ORM.

In "classical" form, the table metadata is created separately with the `Table` construct, then associated with the `User` class via the `mapper()` function:

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

metadata = MetaData()

user = Table('user', metadata,
             Column('id', Integer, primary_key=True),
             Column('name', String(50)),
             Column('fullname', String(50)),
             Column('password', String(12))
```

```
        )

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

mapper(User, user)
```

Information about mapped attributes, such as relationships to other classes, are provided via the `properties` dictionary. The example below illustrates a second `Table` object, mapped to a class called `Address`, then linked to `User` via `relationship()`:

```
address = Table('address', metadata,
            Column('id', Integer, primary_key=True),
            Column('user_id', Integer, ForeignKey('user.id')),
            Column('email_address', String(50))
            )

mapper(User, user, properties={
    'addresses' : relationship(Address, backref='user', order_by=address.c.id)
})

mapper(Address, address)
```

When using classical mappings, classes must be provided directly without the benefit of the "string lookup" system provided by Declarative. SQL expressions are typically specified in terms of the `Table` objects, i.e. `address.c.id` above for the `Address` relationship, and not `Address.id`, as `Address` may not yet be linked to table metadata, nor can we specify a string here.

Some examples in the documentation still use the classical approach, but note that the classical as well as Declarative approaches are **fully interchangeable**. Both systems ultimately create the same configuration, consisting of a `Table`, user-defined class, linked together with a `mapper()`. When we talk about "the behavior of `mapper()`", this includes when using the Declarative system as well - it's still used, just behind the scenes.

Runtime Introspection of Mappings, Objects

The `Mapper` object is available from any mapped class, regardless of method, using the core_inspection_toplevel system. Using the `inspect()` function, one can acquire the `Mapper` from a mapped class:

```
>>> from sqlalchemy import inspect
>>> insp = inspect(User)
```

Detailed information is available including `Mapper.columns`:

```
>>> insp.columns
<sqlalchemy.util._collections.OrderedProperties object at 0x102f407f8>
```

This is a namespace that can be viewed in a list format or via individual names:

```
>>> list(insp.columns)
[Column('id', Integer(), table=<user>, primary_key=True, nullable=False), Column('name',␣
↪String(length=50), table=<user>), Column('fullname', String(length=50), table=<user>),␣
↪Column('password', String(length=12), table=<user>)]
>>> insp.columns.name
Column('name', String(length=50), table=<user>)
```

Other namespaces include `Mapper.all_orm_descriptors`, which includes all mapped attributes as well as hybrids, association proxies:

```
>>> insp.all_orm_descriptors
<sqlalchemy.util._collections.ImmutableProperties object at 0x1040e2c68>
>>> insp.all_orm_descriptors.keys()
['fullname', 'password', 'name', 'id']
```

As well as `Mapper.column_attrs`:

```
>>> list(insp.column_attrs)
[<ColumnProperty at 0x10403fde0; id>, <ColumnProperty at 0x10403fce8; name>, <ColumnProperty␣
↪at 0x1040e9050; fullname>, <ColumnProperty at 0x1040e9148; password>]
>>> insp.column_attrs.name
<ColumnProperty at 0x10403fce8; name>
>>> insp.column_attrs.name.expression
Column('name', String(length=50), table=<user>)
```

See also:

core_inspection_toplevel

`Mapper`

`InstanceState`

2.2.2 Mapping Columns and Expressions

The following sections discuss how table columns and SQL expressions are mapped to individual object attributes.

Mapping Table Columns

The default behavior of `mapper()` is to assemble all the columns in the mapped `Table` into mapped object attributes, each of which are named according to the name of the column itself (specifically, the `key` attribute of `Column`). This behavior can be modified in several ways.

Naming Columns Distinctly from Attribute Names

A mapping by default shares the same name for a `Column` as that of the mapped attribute - specifically it matches the `Column.key` attribute on `Column`, which by default is the same as the `Column.name`.

The name assigned to the Python attribute which maps to `Column` can be different from either `Column.name` or `Column.key` just by assigning it that way, as we illustrate here in a Declarative mapping:

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

Where above `User.id` resolves to a column named `user_id` and `User.name` resolves to a column named `user_name`.

When mapping to an existing table, the `Column` object can be referenced directly:

```
class User(Base):
    __table__ = user_table
    id = user_table.c.user_id
    name = user_table.c.user_name
```

Or in a classical mapping, placed in the `properties` dictionary with the desired key:

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

In the next section we'll examine the usage of `.key` more closely.

Automating Column Naming Schemes from Reflected Tables

In the previous section mapper_column_distinct_names, we showed how a `Column` explicitly mapped to a class can have a different attribute name than the column. But what if we aren't listing out `Column` objects explicitly, and instead are automating the production of `Table` objects using reflection (e.g. as described in metadata_reflection_toplevel)? In this case we can make use of the `DDLEvents.column_reflect()` event to intercept the production of `Column` objects and provide them with the `Column.key` of our choice:

```
@event.listens_for(Table, "column_reflect")
def column_reflect(inspector, table, column_info):
    # set column.key = "attr_<lower_case_name>"
    column_info['key'] = "attr_%s" % column_info['name'].lower()
```

With the above event, the reflection of `Column` objects will be intercepted with our event that adds a new ".key" element, such as in a mapping as below:

```
class MyClass(Base):
    __table__ = Table("some_table", Base.metadata,
                autoload=True, autoload_with=some_engine)
```

If we want to qualify our event to only react for the specific `MetaData` object above, we can check for it in our event:

```
@event.listens_for(Table, "column_reflect")
def column_reflect(inspector, table, column_info):
    if table.metadata is Base.metadata:
        # set column.key = "attr_<lower_case_name>"
        column_info['key'] = "attr_%s" % column_info['name'].lower()
```

Naming All Columns with a Prefix

A quick approach to prefix column names, typically when mapping to an existing `Table` object, is to use `column_prefix`:

```
class User(Base):
    __table__ = user_table
    __mapper_args__ = {'column_prefix':'_'}
```

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

This approach is uncommon in modern usage. For dealing with reflected tables, a more flexible approach is to use that described in mapper_automated_reflection_schemes.

Using column_property for column level options

Options can be specified when mapping a `Column` using the `column_property()` function. This function explicitly creates the `ColumnProperty` used by the `mapper()` to keep track of the `Column`; normally,

the `mapper()` creates this automatically. Using `column_property()`, we can pass additional arguments about how we'd like the `Column` to be mapped. Below, we pass an option `active_history`, which specifies that a change to this column's value should result in the former value being loaded first:

```
from sqlalchemy.orm import column_property

class User(Base):
    __tablename__ = 'user'

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

`column_property()` is also used to map a single attribute to multiple columns. This use case arises when mapping to a `join()` which has attributes which are equated to each other:

```
class User(Base):
    __table__ = user.join(address)

    # assign "user.id", "address.user_id" to the
    # "id" attribute
    id = column_property(user_table.c.id, address_table.c.user_id)
```

For more examples featuring this usage, see maptojoin.

Another place where `column_property()` is needed is to specify SQL expressions as mapped attributes, such as below where we create an attribute `fullname` that is the string concatenation of the `firstname` and `lastname` columns:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))
    fullname = column_property(firstname + " " + lastname)
```

See examples of this usage at mapper_sql_expressions.

`sqlalchemy.orm.column_property(*columns, **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()` or `Session.is_modified()` which also need to know the "previous" value of the attribute.

  New in version 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.
- **expire_on_flush=True** – Disable expiry on flush. A column_property() which refers to a SQL expression (and not a single table-bound column) is considered to be a "read only" property; populating it has no effect on the state of data, and it can only return database state. For this reason a column_property()'s value is expired whenever the parent object is involved in a flush, that is, has any kind of "dirty" state within a flush. Setting this parameter to `False` will have the effect of leaving any existing value present after the flush proceeds. Note however that the `Session` with default expiration settings still expires all attributes after a `Session.commit()` call, however.

  New in version 0.7.3.
- **info** – Optional data dictionary which will be populated into the `MapperProperty.info` attribute of this object.

  New in version 0.8.
- **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. **Deprecated.** Please see `AttributeEvents`.

Mapping a Subset of Table Columns

Sometimes, a `Table` object was made available using the reflection process described at metadata_reflection to load the table's structure from the database. For such a table that has lots of columns that don't need to be referenced in the application, the `include_properties` or `exclude_properties` arguments can specify that only a subset of columns should be mapped. For example:

```
class User(Base):
    __table__ = user_table
    __mapper_args__ = {
        'include_properties' :['user_id', 'user_name']
    }
```

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

```
class Address(Base):
    __table__ = address_table
    __mapper_args__ = {
        'exclude_properties' : ['street', 'city', 'state', 'zip']
    }
```

...will map the `Address` class to the `address_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. `include_properties` and `exclude_properties` can also accommodate `Column` objects to more accurately describe which columns should be included or excluded:

```
class UserAddress(Base):
    __table__ = user_table.join(addresses_table)
    __mapper_args__ = {
        'exclude_properties' :[address_table.c.id],
        'primary_key' : [user_table.c.id]
    }
```

Note: insert and update defaults configured on individual `Column` objects, i.e. those described at metadata_defaults including those configured by the `default`, `update`, `server_default` and `server_onupdate` arguments, will continue to function normally even if those `Column` objects are not mapped. This is because in the case of `default` and `update`, the `Column` object is still present on the underlying `Table`, thus allowing the default functions to take place when the ORM emits an INSERT or UPDATE, and in the case of `server_default` and `server_onupdate`, the relational database itself maintains these functions.

SQL Expressions as Mapped Attributes

Attributes on a mapped class can be linked to SQL expressions, which can be used in queries.

Using a Hybrid

The easiest and most flexible way to link relatively simple SQL expressions to a class is to use a so-called "hybrid attribute", described in the section hybrids_toplevel. The hybrid provides for an expression that works at both the Python level as well as at the SQL expression level. For example, below we map a class `User`, containing attributes `firstname` and `lastname`, and include a hybrid that will provide for us the `fullname`, which is the string concatenation of the two:

```
from sqlalchemy.ext.hybrid import hybrid_property

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

    @hybrid_property
    def fullname(self):
        return self.firstname + " " + self.lastname
```

Above, the `fullname` attribute is interpreted at both the instance and class level, so that it is available from an instance:

```
some_user = session.query(User).first()
print(some_user.fullname)
```

as well as usable within queries:

```
some_user = session.query(User).filter(User.fullname == "John Smith").first()
```

The string concatenation example is a simple one, where the Python expression can be dual purposed at the instance and class level. Often, the SQL expression must be distinguished from the Python expression, which can be achieved using `hybrid_property.expression()`. Below we illustrate the case where a conditional needs to be present inside the hybrid, using the `if` statement in Python and the `sql.expression.case()` construct for SQL expressions:

```
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.sql import case

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

    @hybrid_property
    def fullname(self):
        if self.firstname is not None:
            return self.firstname + " " + self.lastname
        else:
            return self.lastname

    @fullname.expression
    def fullname(cls):
        return case([
            (cls.firstname != None, cls.firstname + " " + cls.lastname),
        ], else_ = cls.lastname)
```

Using column_property

The `orm.column_property()` function can be used to map a SQL expression in a manner similar to a regularly mapped `Column`. With this technique, the attribute is loaded along with all other column-mapped attributes at load time. This is in some cases an advantage over the usage of hybrids, as the value can be loaded up front at the same time as the parent row of the object, particularly if the expression is one which links to other tables (typically as a correlated subquery) to access data that wouldn't normally be available on an already loaded object.

Disadvantages to using `orm.column_property()` for SQL expressions include that the expression must be compatible with the SELECT statement emitted for the class as a whole, and there are also some configurational quirks which can occur when using `orm.column_property()` from declarative mixins.

Our "fullname" example can be expressed using `orm.column_property()` as follows:

```
from sqlalchemy.orm import column_property

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))
    fullname = column_property(firstname + " " + lastname)
```

Correlated subqueries may be used as well. Below we use the `select()` construct to create a SELECT that links together the count of `Address` objects available for a particular `User`:

```
from sqlalchemy.orm import column_property
from sqlalchemy import select, func
from sqlalchemy import Column, Integer, String, ForeignKey

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

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

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    address_count = column_property(
        select([func.count(Address.id)]).\
            where(Address.user_id==id).\
            correlate_except(Address)
    )
```

In the above example, we define a `select()` construct like the following:

```
select([func.count(Address.id)]).\
    where(Address.user_id==id).\
    correlate_except(Address)
```

The meaning of the above statement is, select the count of `Address.id` rows where the `Address.user_id` column is equated to `id`, which in the context of the `User` class is the `Column` named `id` (note that `id` is also the name of a Python built in function, which is not what we want to use here - if we were outside of the `User` class definition, we'd use `User.id`).

The `select.correlate_except()` directive indicates that each element in the FROM clause of this `select()` may be omitted from the FROM list (that is, correlated to the enclosing SELECT statement against `User`) except for the one corresponding to `Address`. This isn't strictly necessary, but prevents `Address` from being inadvertently omitted from the FROM list in the case of a long string of joins between `User` and `Address` tables where SELECT statements against `Address` are nested.

If import issues prevent the `column_property()` from being defined inline with the class, it can be assigned to the class after both are configured. In Declarative this has the effect of calling `Mapper.add_property()` to add an additional property after the fact:

```
User.address_count = column_property(
        select([func.count(Address.id)]).\
            where(Address.user_id==User.id)
    )
```

For many-to-many relationships, use `and_()` to join the fields of the association table to both tables in a relation, illustrated here with a classical mapping:

```
from sqlalchemy import and_

mapper(Author, authors, properties={
    'book_count': column_property(
                        select([func.count(books.c.id)],
                            and_(
                                book_authors.c.author_id==authors.c.id,
                                book_authors.c.book_id==books.c.id
                            )))
    })
```

Using a plain descriptor

In cases where a SQL query more elaborate than what `orm.column_property()` or `hybrid_property` can provide must be emitted, a regular Python function accessed as an attribute can be used, assuming the expression only needs to be available on an already-loaded instance. The function is decorated with Python's own `@property` decorator to mark it as a read-only attribute. Within the function, `object_session()` is used to locate the `Session` corresponding to the current object, which is then used to emit a query:

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

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

    @property
    def address_count(self):
        return object_session(self).\
            scalar(
                select([func.count(Address.id)]).\
                    where(Address.user_id==self.id)
            )
```

The plain descriptor approach is useful as a last resort, but is less performant in the usual case than both the hybrid and column property approaches, in that it needs to emit a SQL query upon each access.

Query-time SQL expressions as mapped attributes

When using `Session.query()`, we have the option to specify not just mapped entities but ad-hoc SQL expressions as well. Suppose if a class `A` had integer attributes `.x` and `.y`, we could query for `A` objects, and additionally the sum of `.x` and `.y`, as follows:

```
q = session.query(A, A.x + A.y)
```

The above query returns tuples of the form `(A object, integer)`.

An option exists which can apply the ad-hoc `A.x + A.y` expression to the returned `A` objects instead of as a separate tuple entry; this is the `with_expression()` query option in conjunction with the `query_expression()` attribute mapping. The class is mapped to include a placeholder attribute where any particular SQL expression may be applied:

```
from sqlalchemy.orm import query_expression

class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    x = Column(Integer)
    y = Column(Integer)

    expr = query_expression()
```

We can then query for objects of type `A`, applying an arbitrary SQL expression to be populated into `A.expr`:

```
from sqlalchemy.orm import with_expression
q = session.query(A).options(
    with_expression(A.expr, A.x + A.y))
```

The `query_expression()` mapping has these caveats:

- On an object where `query_expression()` were not used to populate the attribute, the attribute on an object instance will have the value `None`.
- The query_expression value **does not refresh when the object is expired**. Once the object is expired, either via `Session.expire()` or via the expire_on_commit behavior of `Session.commit()`, the value is removed from the attribute and will return `None` on subsequent access.

Only by running a new `Query` that touches the object which includes a new `with_expression()` directive will the attribute be set to a non-None value.

- The mapped attribute currently **cannot** be applied to other parts of the query, such as the WHERE clause, the ORDER BY clause, and make use of the ad-hoc expression; that is, this won't work:

```
# wont work
q = session.query(A).options(
    with_expression(A.expr, A.x + A.y)
).filter(A.expr > 5).order_by(A.expr)
```

The `A.expr` expression will resolve to NULL in the above WHERE clause and ORDER BY clause. To use the expression throughout the query, assign to a variable and use that:

```
a_expr = A.x + A.y
q = session.query(A).options(
    with_expression(A.expr, a_expr)
).filter(a_expr > 5).order_by(a_expr)
```

New in version 1.2.

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

class EmailAddress(Base):
    __tablename__ = 'address'

    id = Column(Integer, primary_key=True)
    email = Column(String)

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

Changed in version 1.0.0: - validators are no longer triggered within the flush process when the newly fetched values for primary key columns as well as some python- or server-side defaults are fetched. Prior to 1.0, validators may be triggered in those cases as well.

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

```
from sqlalchemy.orm import validates

class User(Base):
    # ...

    addresses = relationship("Address")

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

The validation function by default does not get emitted for collection remove events, as the typical expectation is that a value being discarded doesn't require validation. However, `validates()` supports reception of these events by specifying `include_removes=True` to the decorator. When this flag is set, the validation function must receive an additional boolean argument which if `True` indicates that the operation is a removal:

```
from sqlalchemy.orm import validates

class User(Base):
    # ...

    addresses = relationship("Address")

    @validates('addresses', include_removes=True)
    def validate_address(self, key, address, is_remove):
        if is_remove:
            raise ValueError(
                    "not allowed to remove items from the collection")
        else:
            assert '@' in address.email
            return address
```

The case where mutually dependent validators are linked via a backref can also be tailored, using the `include_backrefs=False` option; this option, when set to `False`, prevents a validation function from emitting if the event occurs as a result of a backref:

```
from sqlalchemy.orm import validates

class User(Base):
    # ...

    addresses = relationship("Address", backref='user')

    @validates('addresses', include_backrefs=False)
    def validate_address(self, key, address):
        assert '@' in address.email
        return address
```

Above, if we were to assign to `Address.user` as in `some_address.user = some_user`, the `validate_address()` function would *not* be emitted, even though an append occurs to `some_user.addresses` - the event is caused by a backref.

Note that the `validates()` decorator is a convenience function built on top of attribute events. An application that requires more control over configuration of attribute change behavior can make use of this system, described at `AttributeEvents`.

`sqlalchemy.orm.validates`(**names*, ***kw*)

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, the value to be added to the collection. The function can then raise validation exceptions to halt the process from continuing (where Python's built-in `ValueError` and `AssertionError` exceptions are reasonable choices), 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.

Parameters

- **`*names`** – list of attribute names to be validated.

- **include_removes** – if True, "remove" events will be sent as well - the validation function must accept an additional argument "is_remove" which will be a boolean.

  New in version 0.7.7.

- **include_backrefs** – defaults to `True`; if `False`, the validation function will not emit if the originator is an attribute event related via a backref. This can be used for bi-directional `validates()` usage where only one validator should emit per attribute operation.

  New in version 0.9.0.

See also:

simple_validators - usage examples for `validates()`

Using Descriptors and Hybrids

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(Base):
    __tablename__ = 'email_address'

    id = Column(Integer, primary_key=True)

    # name the attribute with an underscore,
    # different from the column name
    _email = Column("email", String)

    # then create an ".email" attribute
    # to get/set "._email"
    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, email):
        self._email = 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 `hybrid` extension as follows:

```
from sqlalchemy.ext.hybrid import hybrid_property

class EmailAddress(Base):
    __tablename__ = 'email_address'

    id = Column(Integer, primary_key=True)

    _email = Column("email", String)

    @hybrid_property
    def email(self):
        return self._email

    @email.setter
```

```
    def email(self, email):
        self._email = email
```

The `.email` attribute, in addition to providing getter/setter behavior when we have an instance of `EmailAddress`, also provides a SQL expression when used at the class level, that is, from the `EmailAddress` class directly:

```
from sqlalchemy.orm import Session
session = Session()

{sql}address = session.query(EmailAddress).\
                filter(EmailAddress.email == 'address@example.com').\
                one()
SELECT address.email AS address_email, address.id AS address_id
FROM address
WHERE address.email = ?
('address@example.com',)
{stop}

address.email = 'otheraddress@example.com'
{sql}session.commit()
UPDATE address SET email=? WHERE address.id = ?
('otheraddress@example.com', 1)
COMMIT
{stop}
```

The `hybrid_property` also allows us to change the behavior of the attribute, including defining separate behaviors when the attribute is accessed at the instance level versus at the class/expression level, using the `hybrid_property.expression()` modifier. Such as, if we wanted to add a host name automatically, we might define two sets of string manipulation logic:

```
class EmailAddress(Base):
    __tablename__ = 'email_address'

    id = Column(Integer, primary_key=True)

    _email = Column("email", String)

    @hybrid_property
    def email(self):
        """Return the value of _email up until the last twelve
        characters."""

        return self._email[:-12]

    @email.setter
    def email(self, email):
        """Set the value of _email, tacking on the twelve character
        value @example.com."""

        self._email = email + "@example.com"

    @email.expression
    def email(cls):
        """Produce a SQL expression that represents the value
        of the _email column, minus the last twelve characters."""

        return func.substr(cls._email, 0, func.length(cls._email) - 12)
```

Above, accessing the `email` property of an instance of `EmailAddress` will return the value of the `_email` attribute, removing or adding the hostname `@example.com` from the value. When we query against the `email` attribute, a SQL function is rendered which produces the same effect:

```
{sql}address = session.query(EmailAddress).filter(EmailAddress.email == 'address').one()
SELECT address.email AS address_email, address.id AS address_id
FROM address
WHERE substr(address.email, ?, length(address.email) - ?) = ?
(0, 12, 'address')
{stop}
```

Read more about Hybrids at hybrids_toplevel.

Synonyms

Synonyms are a mapper-level construct that allow any attribute on a class to "mirror" another attribute that is mapped.

In the most basic sense, the synonym is an easy way to make a certain attribute available by an additional name:

```
class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    job_status = Column(String(50))

    status = synonym("job_status")
```

The above class `MyClass` has two attributes, `.job_status` and `.status` that will behave as one attribute, both at the expression level:

```
>>> print(MyClass.job_status == 'some_status')
my_table.job_status = :job_status_1

>>> print(MyClass.status == 'some_status')
my_table.job_status = :job_status_1
```

and at the instance level:

```
>>> m1 = MyClass(status='x')
>>> m1.status, m1.job_status
('x', 'x')

>>> m1.job_status = 'y'
>>> m1.status, m1.job_status
('y', 'y')
```

The `synonym()` can be used for any kind of mapped attribute that subclasses `MapperProperty`, including mapped columns and relationships, as well as synonyms themselves.

Beyond a simple mirror, `synonym()` can also be made to reference a user-defined descriptor. We can supply our `status` synonym with a `@property`:

```
class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    status = Column(String(50))

    @property
    def job_status(self):
        return "Status: " + self.status

    job_status = synonym("status", descriptor=job_status)
```

When using Declarative, the above pattern can be expressed more succinctly using the `synonym_for()` decorator:

```
from sqlalchemy.ext.declarative import synonym_for

class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    status = Column(String(50))

    @synonym_for("status")
    @property
    def job_status(self):
        return "Status: " + self.status
```

While the `synonym()` is useful for simple mirroring, the use case of augmenting attribute behavior with descriptors is better handled in modern usage using the hybrid attribute feature, which is more oriented towards Python descriptors. Technically, a `synonym()` can do everything that a `hybrid_property` can do, as it also supports injection of custom SQL capabilities, but the hybrid is more straightforward to use in more complex situations.

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

Denote an attribute name as a synonym to a mapped property, in that the attribute will mirror the value and expression behavior of another attribute.

e.g.:

```
class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    job_status = Column(String(50))

    status = synonym("job_status")
```

Parameters

- **name** – the name of the existing mapped property. This can refer to the string name ORM-mapped attribute configured on the class, including column-bound attributes and relationships.
- **descriptor** – a Python descriptor that will be used as a getter (and potentially a setter) when this attribute is accessed at the instance level.
- **map_column** – **For classical mappings and mappings against an existing Table object only**. if `True`, the `synonym()` construct will locate the `Column` object upon the mapped table that would normally be associated with the attribute name of this synonym, and produce a new `ColumnProperty` that instead maps this `Column` to the alternate name given as the "name" argument of the synonym; in this way, the usual step of redefining the mapping of the `Column` to be under a different name is unnecessary. This is usually intended to be used when a `Column` is to be replaced with an attribute that also uses a descriptor, that is, in conjunction with the `synonym.descriptor` parameter:

```
my_table = Table(
    "my_table", metadata,
    Column('id', Integer, primary_key=True),
    Column('job_status', String(50))
)

class MyClass(object):
```

```
    @property
    def _job_status_descriptor(self):
        return "Status: %s" % self._job_status


mapper(
    MyClass, my_table, properties={
        "job_status": synonym(
            "_job_status", map_column=True,
            descriptor=MyClass._job_status_descriptor)
    }
)
```

Above, the attribute named `_job_status` is automatically mapped to the `job_status` column:

```
>>> j1 = MyClass()
>>> j1._job_status = "employed"
>>> j1.job_status
Status: employed
```

When using Declarative, in order to provide a descriptor in conjunction with a synonym, use the `sqlalchemy.ext.declarative.synonym_for()` helper. However, note that the hybrid properties feature should usually be preferred, particularly when redefining attribute behavior.

- `info` – Optional data dictionary which will be populated into the `InspectionAttr.info` attribute of this object.

  New in version 1.0.0.

- `comparator_factory` – A subclass of `PropComparator` that will provide custom comparison behavior at the SQL expression level.

  Note: For the use case of providing an attribute which redefines both Python-level and SQL-expression level behavior of an attribute, please refer to the Hybrid attribute introduced at mapper_hybrids for a more effective technique.

See also:

synonyms - Overview of synonyms

`synonym_for()` - a helper oriented towards Declarative

mapper_hybrids - The Hybrid Attribute extension provides an updated approach to augmenting attribute behavior more flexibly than can be achieved with synonyms.

Operator Customization

The "operators" used by the SQLAlchemy ORM and Core expression language are fully customizable. For example, the comparison expression `User.name == 'ed'` makes usage of an operator built into Python itself called `operator.eq` - the actual SQL construct which SQLAlchemy associates with such an operator can be modified. New operations can be associated with column expressions as well. The operators which take place for column expressions are most directly redefined at the type level - see the section types_operators for a description.

ORM level functions like `column_property()`, `relationship()`, and `composite()` also provide for operator redefinition at the ORM level, by passing a `PropComparator` subclass to the `comparator_factory` argument of each function. Customization of operators at this level is a rare use case. See the documentation at `PropComparator` for an overview.

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. `Point` represents such a pair as `.x` and `.y`:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __composite_values__(self):
        return self.x, self.y

    def __repr__(self):
        return "Point(x=%r, y=%r)" % (self.x, self.y)

    def __eq__(self, other):
        return isinstance(other, Point) 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.

We will create a mapping to a table `vertices`, which represents two points as `x1/y1` and `x2/y2`. These are created normally as `Column` objects. Then, the `composite()` function is used to assign new attributes that will represent sets of columns via the `Point` class:

```
from sqlalchemy import Column, Integer
from sqlalchemy.orm import composite
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Vertex(Base):
    __tablename__ = 'vertices'

    id = Column(Integer, primary_key=True)
    x1 = Column(Integer)
    y1 = Column(Integer)
    x2 = Column(Integer)
    y2 = Column(Integer)

    start = composite(Point, x1, y1)
    end = composite(Point, x2, y2)
```

A classical mapping above would define each `composite()` against the existing table:

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

We can now persist and use `Vertex` instances, as well as query for them, using the `.start` and `.end` attributes against ad-hoc `Point` instances:

```
>>> v = Vertex(start=Point(3, 4), end=Point(5, 6))
>>> session.add(v)
>>> q = session.query(Vertex).filter(Vertex.start == Point(3, 4))
{sql}>>> print(q.first().start)
BEGIN (implicit)
INSERT INTO vertices (x1, y1, x2, y2) VALUES (?, ?, ?, ?)
(3, 4, 5, 6)
SELECT vertices.id AS vertices_id,
        vertices.x1 AS vertices_x1,
        vertices.y1 AS vertices_y1,
        vertices.x2 AS vertices_x2,
        vertices.y2 AS vertices_y2
FROM vertices
WHERE vertices.x1 = ? AND vertices.y1 = ?
 LIMIT ? OFFSET ?
(3, 4, 1, 0)
{stop}Point(x=3, y=4)
```

sqlalchemy.orm.**composite**(*class_*, **attrs*, ***kwargs*)

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

See the mapping documentation section mapper_composite for a full usage example.

The `MapperProperty` returned by `composite()` is the `CompositeProperty`.

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. See the same flag on `column_property()`.

  Changed in version 0.7: This flag specifically becomes meaningful - previously it was a placeholder.
- **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.
- **info** – Optional data dictionary which will be populated into the `MapperProperty.info` attribute of this object.

  New in version 0.8.
- **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. **Deprecated.** Please see `AttributeEvents`.

Tracking In-Place Mutations on Composites

In-place changes to an existing composite value are not tracked automatically. Instead, the composite class needs to provide events to its parent object explicitly. This task is largely automated via the usage of the `MutableComposite` mixin, which uses events to associate each user-defined composite object with all parent associations. Please see the example in mutable_composites.

Changed in version 0.7: In-place changes to an existing composite value are no longer tracked automatically; the functionality is superseded by the `MutableComposite` class.

Redefining Comparison Operations for Composites

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` argument to `composite()`, where we specify a custom `CompositeProperty.Comparator` class to define existing or new operations. 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__())])

class Vertex(Base):
    ___tablename__ = 'vertices'

    id = Column(Integer, primary_key=True)
    x1 = Column(Integer)
    y1 = Column(Integer)
    x2 = Column(Integer)
    y2 = Column(Integer)

    start = composite(Point, x1, y1,
                      comparator_factory=PointComparator)
    end = composite(Point, x2, y2,
                      comparator_factory=PointComparator)
```

2.2.3 Mapping Class Inheritance Hierarchies

SQLAlchemy supports three forms of inheritance: **single table inheritance**, where several types of classes are represented by a single table, **concrete table inheritance**, where each type of class is represented by independent tables, and **joined table inheritance**, where the class hierarchy is broken up among dependent tables, each class represented by its own table that only includes those attributes local to that class.

The most common forms of inheritance are single and joined table, while concrete inheritance presents more configurational challenges.

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.

See also:

examples_inheritance - complete exampes of joined, single and concrete inheritance

Joined Table Inheritance

In joined table inheritance, each class along a hierarchy of classes is represented by a distinct table. Querying for a particular subclass in the hierarchy will render as a SQL JOIN along all tables in its inheritance path - if the class is the base class, the default behavior is to include only the base table in

the SELECT. In all cases, the ultimate class to instantiate for a given row is determined by a discriminator column or expression that works against the base table. A subclass loaded against the base table only will have only base attributes populated at first; the additional attributes will lazy load when they are accessed. Options also exist to query for all columns across multiple tables/subclasses up front.

The base class in a joined inheritance hierarchy is configured with additional arguments that will refer to the polymorphic discriminator column as well as the identifier for the base class:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'employee',
        'polymorphic_on':type
    }
```

Above, an additional column `type` is established to act as the **discriminator**, configured as such using the `mapper.polymorphic_on` parameter. This column will store a value which indicates the type of object represented within the row. The column may be of any datatype, though string and integer are the most common.

While a polymorphic discriminator expression is not strictly necessary, it is required if polymorphic loading is desired. Establishing a simple column on the base table is the easiest way to achieve this, however very sophisticated inheritance mappings may even configure a SQL expression such as a CASE statement as the polymorphic discriminator.

Note: Currently, **only one discriminator column or SQL expression may be configured for the entire inheritance hierarchy**, typically on the base- most class in the hierarchy. "Cascading" polymorphic discriminator expressions are not yet supported.

We next define `Engineer` and `Manager` subclasses of `Employee`. Each contains columns that represent the attributes unique to the subclass they represent. Each table also must contain a primary key column (or columns), as well as a foreign key reference to the parent table:

```
class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    engineer_name = Column(String(30))

    __mapper_args__ = {
        'polymorphic_identity':'engineer',
    }

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_name = Column(String(30))

    __mapper_args__ = {
        'polymorphic_identity':'manager',
    }
```

It is most common that the foreign key constraint is established on the same column or columns as the primary key itself, however this is not required; a column distinct from the primary key may also be made to refer to the parent via foreign key. The way that a JOIN is constructed from the base table to subclasses is also directly customizable, however this is rarely necessary.

Joined inheritance primary keys

One natural effect of the joined table inheritance configuration is that the identity of any mapped object can be determined entirely from rows in the base table alone. 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. In other words, the `id` columns of both the `engineer` and `manager` tables are not used to locate `Engineer` or `Manager` objects - only the value in `employee.id` is considered. `engineer.id` and `manager.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 within a statement.

With the joined inheritance mapping complete, querying against `Employee` will return a combination of `Employee`, `Engineer` and `Manager` objects. Newly saved `Engineer`, `Manager`, and `Employee` objects will automatically populate the `employee.type` column with the correct "discriminator" value in this case `"engineer"`, `"manager"`, or `"employee"`, as appropriate.

Relationships with Joined Inheritance

Relationships are fully supported with joined table inheritance. The relationship involving a joined-inheritance class should target the class in the hierarchy that also corresponds to the foreign key constraint; below, as the `employee` table has a foreign key constraint back to the `company` table, the relationships are set up between `Company` and `Employee`:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    employees = relationship("Employee", back_populates="company")

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(50))
    company_id = Column(ForeignKey('company.id'))
    company = relationship("Company", back_populates="employees")

    __mapper_args__ = {
        'polymorphic_identity':'employee',
        'polymorphic_on':type
    }

class Manager(Employee):
    # ...

class Engineer(Employee):
    # ...
```

If the foreign key constraint is on a table corresponding to a subclass, the relationship should target that subclass instead. In the example below, there is a foreign key constraint from `manager` to `company`, so the relationships are established between the `Manager` and `Company` classes:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    managers = relationship("Manager", back_populates="company")

class Employee(Base):
    __tablename__ = 'employee'
```

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

    __mapper_args__ = {
        'polymorphic_identity':'employee',
        'polymorphic_on':type
    }

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_name = Column(String(30))

    company_id = Column(ForeignKey('company.id'))
    company = relationship("Company", back_populates="managers")

    __mapper_args__ = {
        'polymorphic_identity':'manager',
    }

class Engineer(Employee):
    # ...
```

Above, the `Manager` class will have a `Manager.company` attribute; `Company` will have a `Company.managers` attribute that always loads against a join of the `employee` and `manager` tables together.

Loading Joined Inheritance Mappings

See the sections inheritance_loading_toplevel and loading_joined_inheritance for background on inheritnce loading techniques, including configuration of tables to be queried both at mapper configuration time as well as query time.

Single Table Inheritance

Single table inheritance represents all attributes of all subclasses within a single table. A particular subclass that has attributes unique to that class will persist them within columns in the table that are otherwise NULL if the row refers to a different kind of object.

Querying for a particular subclass in the hierarchy will render as a SELECT against the base table, which will include a WHERE clause that limits rows to those with a particular value or values present in the discriminator column or expression.

Single table inheritance has the advantage of simplicity compared to joined table inheritance; queries are much more efficient as only one table needs to be involved in order to load objects of every represented class.

Single-table inheritance configuration looks much like joined-table inheritance, except only the base class specifies `__tablename__`. A discriminator column is also required on the base table so that classes can be differentiated from each other.

Even though subclasses share the base table for all of their attributes, when using Declarative, `Column` objects may still be specified on subclasses, indicating that the column is to be mapped only to that subclass; the `Column` will be applied to the same base `Table` object:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(20))
```

```
    __mapper_args__ = {
        'polymorphic_on':type,
        'polymorphic_identity':'employee'
    }

class Manager(Employee):
    manager_data = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'manager'
    }

class Engineer(Employee):
    engineer_info = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'engineer'
    }
```

Note that the mappers for the derived classes Manager and Engineer omit the `__tablename__`, indicating they do not have a mapped table of their own.

Relationships with Single Table Inheritance

Relationships are fully supported with single table inheritance. Configuration is done in the same manner as that of joined inheritance; a foreign key attribute should be on the same class that's the "foreign" side of the relationship:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    employees = relationship("Employee", back_populates="company")

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(50))
    company_id = Column(ForeignKey('company.id'))
    company = relationship("Company", back_populates="employees")

    __mapper_args__ = {
        'polymorphic_identity':'employee',
        'polymorphic_on':type
    }


class Manager(Employee):
    manager_data = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'manager'
    }

class Engineer(Employee):
    engineer_info = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'engineer'
```

```
    }
```

Also, like the case of joined inheritance, we can create relationships that involve a specific subclass. When queried, the SELECT statement will include a WHERE clause that limits the class selection to that subclass or subclasses:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    managers = relationship("Manager", back_populates="company")

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'employee',
        'polymorphic_on':type
    }


class Manager(Employee):
    manager_name = Column(String(30))

    company_id = Column(ForeignKey('company.id'))
    company = relationship("Company", back_populates="managers")

    __mapper_args__ = {
        'polymorphic_identity':'manager',
    }


class Engineer(Employee):
    engineer_info = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'engineer'
    }
```

Above, the `Manager` class will have a `Manager.company` attribute; `Company` will have a `Company.managers` attribute that always loads against the `employee` with an additional WHERE clause that limits rows to those with `type = 'manager'`.

Loading Single Inheritance Mappings

The loading techniques for single-table inheritance are mostly identical to those used for joined-table inheritance, and a high degree of abstraction is provided between these two mapping types such that it is easy to switch between them as well as to intermix them in a single hierarchy (just omit `__tablename__` from whichever subclasses are to be single-inheriting). See the sections inheritance_loading_toplevel and loading_single_inheritance for documentation on inheritance loading techniques, including configuration of classes to be queried both at mapper configuration time as well as query time.

Concrete Table Inheritance

Concrete inheritance maps each subclass to its own distinct table, each of which contains all columns necessary to produce an instance of that class. A concrete inheritance configuration by default queries

non-polymorphically; a query for a particular class will only query that class' table and only return instances of that class. Polymorphic loading of concrete classes is enabled by configuring within the mapper a special SELECT that typically is produced as a UNION of all the tables.

Whereas joined and single table inheritance are fluent in "polymorphic" loading, it is a more awkward affair in concrete inheritance. For this reason, concrete inheritance is more appropriate when polymorphic loading is not required. Establishing relationships that involve concrete inheritance classes is also more awkward.

To establish a class as using concrete inheritance, add the `mapper.concrete` parameter within the `__mapper_args__`. This indicates to Declarative as well as the mapping that the superclass table should not be considered as part of the mapping:

```
class Employee(Base):
    __tablename__ = 'employee'

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

class Manager(Employee):
    __tablename__ = 'manager'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(50))

    __mapper_args__ = {
        'concrete': True
    }

class Engineer(Employee):
    __tablename__ = 'engineer'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    engineer_info = Column(String(50))

    __mapper_args__ = {
        'concrete': True
    }
```

Two critical points should be noted:

- We must **define all columns explicitly** on each subclass, even those of the same name. A column such as `Employee.name` here is **not** copied out to the tables mapped by `Manager` or `Engineer` for us.
- while the `Engineer` and `Manager` classes are mapped in an inheritance relationship with `Employee`, they still **do not include polymorphic loading**. Meaning, if we query for `Employee` objects, the `manager` and `engineer` tables are not queried at all.

Concrete Polymorphic Loading Configuration

Polymorphic loading with concrete inheritance requires that a specialized SELECT is configured against each base class that should have polymorphic loading. This SELECT needs to be capable of accessing all the mapped tables individually, and is typically a UNION statement that is constructed using a SQLAlchemy helper `polymorphic_union()`.

As discussed in inheritance_loading_toplevel, mapper inheritance configurations of any type can be configured to load from a special selectable by default using the `mapper.with_polymorphic` argument. Current public API requires that this argument is set on a `Mapper` when it is first constructed.

However, in the case of Declarative, both the mapper and the `Table` that is mapped are created at once, the moment the mapped class is defined. This means that the `mapper.with_polymorphic` argument cannot be provided yet, since the `Table` objects that correspond to the subclasses haven't yet been defined.

There are a few strategies available to resolve this cycle, however Declarative provides helper classes `ConcreteBase` and `AbstractConcreteBase` which handle this issue behind the scenes.

Using `ConcreteBase`, we can set up our concrete mapping in almost the same way as we do other forms of inheritance mappings:

```
from sqlalchemy.ext.declarative import ConcreteBase

class Employee(ConcreteBase, Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity': 'employee',
        'concrete': True
    }

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity': 'manager',
        'concrete': True
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    engineer_info = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity': 'engineer',
        'concrete': True
    }
```

Above, Declarative sets up the polymorphic selectable for the `Employee` class at mapper "initialization" time; this is the late-configuration step for mappers that resolves other dependent mappers. The `ConcreteBase` helper uses the `polymorphic_union()` function to create a UNION of all concrete-mapped tables after all the other classes are set up, and then configures this statement with the already existing base-class mapper.

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

```
session.query(Employee).all()
{opensql}
SELECT
    pjoin.id AS pjoin_id,
    pjoin.name AS pjoin_name,
    pjoin.type AS pjoin_type,
    pjoin.manager_data AS pjoin_manager_data,
    pjoin.engineer_info AS pjoin_engineer_info
FROM (
    SELECT
```

```
        employee.id AS id,
        employee.name AS name,
        CAST(NULL AS VARCHAR(50)) AS manager_data,
        CAST(NULL AS VARCHAR(50)) AS engineer_info,
        'employee' AS type
    FROM employee
    UNION ALL
    SELECT
        manager.id AS id,
        manager.name AS name,
        manager.manager_data AS manager_data,
        CAST(NULL AS VARCHAR(50)) AS engineer_info,
        'manager' AS type
    FROM manager
    UNION ALL
    SELECT
        engineer.id AS id,
        engineer.name AS name,
        CAST(NULL AS VARCHAR(50)) AS manager_data,
        engineer.engineer_info AS engineer_info,
        'engineer' AS type
    FROM engineer
) AS pjoin
```

The above UNION query needs to manufacture "NULL" columns for each subtable in order to accommodate for those columns that aren't members of that particular subclass.

Abstract Concrete Classes

The concrete mappings illustrated thus far show both the subclasses as well as the base class mapped to individual tables. In the concrete inheritance use case, it is common that the base class is not represented within the database, only the subclasses. In other words, the base class is "abstract".

Normally, when one would like to map two different subclasses to individual tables, and leave the base class unmapped, this can be achieved very easily. When using Declarative, just declare the base class with the `__abstract__` indicator:

```
class Employee(Base):
    __abstract__ = True

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity': 'manager',
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    engineer_info = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity': 'engineer',
    }
```

Above, we are not actually making use of SQLAlchemy's inheritance mapping facilities; we can load and

persist instances of `Manager` and `Engineer` normally. The situation changes however when we need to **query polymorphically**, that is, we'd like to emit `session.query(Employee)` and get back a collection of `Manager` and `Engineer` instances. This brings us back into the domain of concrete inheritance, and we must build a special mapper against `Employee` in order to achieve this.

Mappers can always SELECT

In SQLAlchemy, a mapper for a class always has to refer to some "selectable", which is normally a `Table` but may also refer to any `select()` object as well. While it may appear that a "single table inheritance" mapper does not map to a table, these mappers in fact implicitly refer to the table that is mapped by a superclass.

To modify our concrete inheritance example to illustrate an "abstract" base that is capable of polymorphic loading, we will have only an `engineer` and a `manager` table and no `employee` table, however the `Employee` mapper will be mapped directly to the "polymorphic union", rather than specifying it locally to the `mapper.with_polymorphic` parameter.

To help with this, Declarative offers a variant of the `ConcreteBase` class called `AbstractConcreteBase` which achieves this automatically:

```
from sqlalchemy.ext.declarative import AbstractConcreteBase

class Employee(AbstractConcreteBase, Base):
    pass

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity': 'manager',
        'concrete': True
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    engineer_info = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity': 'engineer',
        'concrete': True
    }
```

The `AbstractConcreteBase` helper class has a more complex internal process than that of `ConcreteBase`, in that the entire mapping of the base class must be delayed until all the subclasses have been declared. With a mapping like the above, only instances of `Manager` and `Engineer` may be persised; querying against the `Employee` class will always produce `Manager` and `Engineer` objects.

See also:

declarative_concrete_table - in the Declarative reference documentation

Classical and Semi-Classical Concrete Polymorphic Configuration

The Declarative configurations illustrated with `ConcreteBase` and `AbstractConcreteBase` are equivalent to two other forms of configuration that make use of `polymorphic_union()` explicitly. These

configurational forms make use of the `Table` object explicitly so that the "polymorphic union" can be created first, then applied to the mappings. These are illustrated here to clarify the role of the `polymorphic_union()` function in terms of mapping.

A **semi-classical mapping** for example makes use of Declarative, but establishes the `Table` objects separately:

```
metadata = Base.metadata

employees_table = Table(
    'employee', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50)),
)

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

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

Next, the UNION is produced using `polymorphic_union()`:

```
from sqlalchemy.orm import polymorphic_union

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

With the above `Table` objects, the mappings can be produced using "semi-classical" style, where we use Declarative in conjunction with the `__table__` argument; our polymorphic union above is passed via `__mapper_args__` to the `mapper.with_polymorphic` parameter:

```
class Employee(Base):
    __table__ = employee_table
    __mapper_args__ = {
        'polymorphic_on': pjoin.c.type,
        'with_polymorphic': ('*', pjoin),
        'polymorphic_identity': 'employee'
    }

class Engineer(Employee):
    __table__ = engineer_table
    __mapper_args__ = {
        'polymorphic_identity': 'engineer',
        'concrete': True}

class Manager(Employee):
    __table__ = manager_table
    __mapper_args__ = {
        'polymorphic_identity': 'manager',
        'concrete': True}
```

Alternatvely, the same `Table` objects can be used in fully "classical" style, without using Declarative at all. A constructor similar to that supplied by Declarative is illustrated:

```
class Employee(object):
    def __init__(self, **kw):
        for k in kw:
            setattr(self, k, kw[k])

class Manager(Employee):
    pass

class Engineer(Employee):
    pass

employee_mapper = mapper(Employee, pjoin,
                                    with_polymorphic=('*', pjoin),
                                    polymorphic_on=pjoin.c.type)
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')
```

The "abstract" example can also be mapped using "semi-classical" or "classical" style. The difference is that instead of applying the "polymorphic union" to the `mapper.with_polymorphic` parameter, we apply it directly as the mapped selectable on our basemost mapper. The semi-classical mapping is illustrated below:

```
from sqlalchemy.orm import polymorphic_union

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

class Employee(Base):
    __table__ = pjoin
    __mapper_args__ = {
        'polymorphic_on': pjoin.c.type,
        'with_polymorphic': '*',
        'polymorphic_identity': 'employee'
    }

class Engineer(Employee):
    __table__ = engineer_table
    __mapper_args__ = {
        'polymorphic_identity': 'engineer',
        'concrete': True}

class Manager(Employee):
    __table__ = manager_table
    __mapper_args__ = {
        'polymorphic_identity': 'manager',
        'concrete': True}
```

Above, we use `polymorphic_union()` in the same manner as before, except that we omit the `employee` table.

See also:

classical_mapping - background information on "classical" mappings

Relationships with Concrete Inheritance

In a concrete inheritance scenario, mapping relationships is challenging since the distinct classes do not share a table. If the relationships only involve specific classes, such as a relationship between `Company` in our previous examples and `Manager`, special steps aren't needed as these are just two related tables.

However, if `Company` is to have a one-to-many relationship to `Employee`, indicating that the collection may include both `Engineer` and `Manager` objects, that implies that `Employee` must have polymorphic loading capabilities and also that each table to be related must have a foreign key back to the `company` table. An example of such a configuration is as follows:

```
from sqlalchemy.ext.declarative import ConcreteBase


class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    employees = relationship("Employee")


class Employee(ConcreteBase, Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    company_id = Column(ForeignKey('company.id'))

    __mapper_args__ = {
        'polymorphic_identity': 'employee',
        'concrete': True
    }


class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))
    company_id = Column(ForeignKey('company.id'))

    __mapper_args__ = {
        'polymorphic_identity': 'manager',
        'concrete': True
    }


class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    engineer_info = Column(String(40))
    company_id = Column(ForeignKey('company.id'))

    __mapper_args__ = {
        'polymorphic_identity': 'engineer',
        'concrete': True
    }
```

The next complexity with concrete inheritance and relationships involves when we'd like one or all of `Employee`, `Manager` and `Engineer` to themselves refer back to `Company`. For this case, SQLAlchemy has special behavior in that a `relationship()` placed on `Employee` which links to `Company` **does not work** against the `Manager` and `Engineer` classes, when exercised at the instance level. Instead, a distinct

`relationship()` must be applied to each class. In order to achieve bi-directional behavior in terms of three separate relationships which serve as the opposite of `Company.employees`, the `relationship.back_populates` parameter is used between each of the relationships:

```
from sqlalchemy.ext.declarative import ConcreteBase


class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    employees = relationship("Employee", back_populates="company")


class Employee(ConcreteBase, Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    company_id = Column(ForeignKey('company.id'))
    company = relationship("Company", back_populates="employees")

    __mapper_args__ = {
        'polymorphic_identity': 'employee',
        'concrete': True
    }


class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))
    company_id = Column(ForeignKey('company.id'))
    company = relationship("Company", back_populates="employees")

    __mapper_args__ = {
        'polymorphic_identity': 'manager',
        'concrete': True
    }


class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    engineer_info = Column(String(40))
    company_id = Column(ForeignKey('company.id'))
    company = relationship("Company", back_populates="employees")

    __mapper_args__ = {
        'polymorphic_identity': 'engineer',
        'concrete': True
    }
```

The above limitation is related to the current implementation, including that concrete inheriting classes do not share any of the attributes of the superclass and therefore need distinct relationships to be set up.

Loading Concrete Inheritance Mappings

The options for loading with concrete inheritance are limited; generally, if polymorphic loading is configured on the mapper using one of the declarative concrete mixins, it can't be modified at query time in current SQLAlchemy versions. Normally, the `orm.with_polymorphic()` function would be able to override the style of loading used by concrete, however due to current limitations this is not yet supported.

2.2.4 Non-Traditional Mappings

Mapping a Class against Multiple Tables

Mappers can be constructed against arbitrary relational units (called *selectables*) in addition to plain tables. For example, the `join()` function creates a selectable unit comprised of multiple tables, complete with its own composite primary key, which can be mapped in the same way as a `Table`:

```
from sqlalchemy import Table, Column, Integer, \
        String, MetaData, join, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import column_property

metadata = MetaData()

# define two Table objects
user_table = Table('user', metadata,
            Column('id', Integer, primary_key=True),
            Column('name', String),
        )

address_table = Table('address', metadata,
            Column('id', Integer, primary_key=True),
            Column('user_id', Integer, ForeignKey('user.id')),
            Column('email_address', String)
            )

# define a join between them.  This
# takes place across the user.id and address.user_id
# columns.
user_address_join = join(user_table, address_table)

Base = declarative_base()

# map to it
class AddressUser(Base):
    __table__ = user_address_join

    id = column_property(user_table.c.id, address_table.c.user_id)
    address_id = address_table.c.id
```

In the example above, the join expresses columns for both the `user` and the `address` table. The `user.id` and `address.user_id` columns are equated by foreign key, so in the mapping they are defined as one attribute, `AddressUser.id`, using `column_property()` to indicate a specialized column mapping. Based on this part of the configuration, the mapping will copy new primary key values from `user.id` into the `address.user_id` column when a flush occurs.

Additionally, the `address.id` column is mapped explicitly to an attribute named `address_id`. This is to **disambiguate** the mapping of the `address.id` column from the same-named `AddressUser.id` attribute, which here has been assigned to refer to the `user` table combined with the `address.user_id` foreign key.

The natural primary key of the above mapping is the composite of `(user.id, address.id)`, as these are the primary key columns of the `user` and `address` table combined together. The identity of an

AddressUser object will be in terms of these two values, and is represented from an AddressUser object as (AddressUser.id, AddressUser.address_id).

Mapping a Class against Arbitrary Selects

Similar to mapping against a join, a plain select() object can be used with a mapper as well. The example fragment below illustrates mapping a class called Customer to a select() which includes a join to a subquery:

```
from sqlalchemy import select, func

subq = select([
            func.count(orders.c.id).label('order_count'),
            func.max(orders.c.price).label('highest_order'),
            orders.c.customer_id
            ]).group_by(orders.c.customer_id).alias()

customer_select = select([customers, subq]).\
            select_from(
                join(customers, subq,
                        customers.c.id == subq.c.customer_id)
            ).alias()

class Customer(Base):
    __table__ = customer_select
```

Above, the full row represented by customer_select will be all the columns of the customers table, in addition to those columns exposed by the subq subquery, which are order_count, highest_order, and customer_id. Mapping the Customer class to this selectable then creates a class which will contain those attributes.

When the ORM persists new instances of Customer, only the customers table will actually receive an INSERT. This is because the primary key of the orders table is not represented in the mapping; the ORM will only emit an INSERT into a table for which it has mapped the primary key.

Note: The practice of mapping to arbitrary SELECT statements, especially complex ones as above, is almost never needed; it necessarily tends to produce complex queries which are often less efficient than that which would be produced by direct query construction. The practice is to some degree based on the very early history of SQLAlchemy where the mapper() construct was meant to represent the primary querying interface; in modern usage, the Query object can be used to construct virtually any SELECT statement, including complex composites, and should be favored over the "map-to-selectable" approach.

Multiple Mappers for One Class

In modern SQLAlchemy, a particular class is mapped by only one so-called **primary** mapper at a time. This mapper is involved in three main areas of functionality: querying, persistence, and instrumentation of the mapped class. The rationale of the primary mapper relates to the fact that the mapper() modifies the class itself, not only persisting it towards a particular Table, but also instrumenting attributes upon the class which are structured specifically according to the table metadata. It's not possible for more than one mapper to be associated with a class in equal measure, since only one mapper can actually instrument the class.

However, there is a class of mapper known as the **non primary** mapper with allows additional mappers to be associated with a class, but with a limited scope of use. This scope typically applies to being able to load rows from an alternate table or selectable unit, but still producing classes which are ultimately persisted using the primary mapping. The non-primary mapper is created using the classical style of mapping against a class that is already mapped with a primary mapper, and involves the use of the non_primary flag.

The non primary mapper is of very limited use in modern SQLAlchemy, as the task of being able to load classes from subqueries or other compound statements can be now accomplished using the `Query` object directly.

There is really only one use case for the non-primary mapper, which is that we wish to build a `relationship()` to such a mapper; this is useful in the rare and advanced case that our relationship is attempting to join two classes together using many tables and/or joins in between. An example of this pattern is at relationship_non_primary_mapper.

As far as the use case of a class that can actually be fully persisted to different tables under different scenarios, very early versions of SQLAlchemy offered a feature for this adapted from Hibernate, known as the "entity name" feature. However, this use case became infeasable within SQLAlchemy once the mapped class itself became the source of SQL expression construction; that is, the class' attributes themselves link directly to mapped table columns. The feature was removed and replaced with a simple recipe-oriented approach to accomplishing this task without any ambiguity of instrumentation - to create new subclasses, each mapped individually. This pattern is now available as a recipe at Entity Name.

2.2.5 Configuring a Version Counter

The `Mapper` supports management of a version id column, which is a single table column that increments or otherwise updates its value each time an `UPDATE` to the mapped table occurs. This value is checked each time the ORM emits an `UPDATE` or `DELETE` against the row to ensure that the value held in memory matches the database value.

Warning: Because the versioning feature relies upon comparison of the **in memory** record of an object, the feature only applies to the `Session.flush()` process, where the ORM flushes individual in-memory rows to the database. It does **not** take effect when performing a multirow UPDATE or DELETE using `Query.update()` or `Query.delete()` methods, as these methods only emit an UPDATE or DELETE statement but otherwise do not have direct access to the contents of those rows being affected.

The purpose of this feature is to detect when two concurrent transactions are modifying the same row at roughly the same time, or alternatively to provide a guard against the usage of a "stale" row in a system that might be re-using data from a previous transaction without refreshing (e.g. if one sets `expire_on_commit=False` with a `Session`, it is possible to re-use the data from a previous transaction).

Concurrent transaction updates

When detecting concurrent updates within transactions, it is typically the case that the database's transaction isolation level is below the level of repeatable read; otherwise, the transaction will not be exposed to a new row value created by a concurrent update which conflicts with the locally updated value. In this case, the SQLAlchemy versioning feature will typically not be useful for in-transaction conflict detection, though it still can be used for cross-transaction staleness detection.

The database that enforces repeatable reads will typically either have locked the target row against a concurrent update, or is employing some form of multi version concurrency control such that it will emit an error when the transaction is committed. SQLAlchemy's version_id_col is an alternative which allows version tracking to occur for specific tables within a transaction that otherwise might not have this isolation level set.

See also:

Repeatable Read Isolation Level - PostgreSQL's implementation of repeatable read, including a description of the error condition.

Simple Version Counting

The most straightforward way to track versions is to add an integer column to the mapped table, then establish it as the `version_id_col` within the mapper options:

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

    id = Column(Integer, primary_key=True)
    version_id = Column(Integer, nullable=False)
    name = Column(String(50), nullable=False)

    __mapper_args__ = {
        "version_id_col": version_id
    }
```

Note: It is **strongly recommended** that the `version_id` column be made NOT NULL. The versioning feature **does not support** a NULL value in the versioning column.

Above, the `User` mapping tracks integer versions using the column `version_id`. When an object of type `User` is first flushed, the `version_id` column will be given a value of "1". Then, an UPDATE of the table later on will always be emitted in a manner similar to the following:

```
UPDATE user SET version_id=:version_id, name=:name
WHERE user.id = :user_id AND user.version_id = :user_version_id
{"name": "new name", "version_id": 2, "user_id": 1, "user_version_id": 1}
```

The above UPDATE statement is updating the row that not only matches `user.id = 1`, it also is requiring that `user.version_id = 1`, where "1" is the last version identifier we've been known to use on this object. If a transaction elsewhere has modified the row independently, this version id will no longer match, and the UPDATE statement will report that no rows matched; this is the condition that SQLAlchemy tests, that exactly one row matched our UPDATE (or DELETE) statement. If zero rows match, that indicates our version of the data is stale, and a `StaleDataError` is raised.

Custom Version Counters / Types

Other kinds of values or counters can be used for versioning. Common types include dates and GUIDs. When using an alternate type or counter scheme, SQLAlchemy provides a hook for this scheme using the `version_id_generator` argument, which accepts a version generation callable. This callable is passed the value of the current known version, and is expected to return the subsequent version.

For example, if we wanted to track the versioning of our `User` class using a randomly generated GUID, we could do this (note that some backends support a native GUID type, but we illustrate here using a simple string):

```
import uuid

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    version_uuid = Column(String(32), nullable=False)
    name = Column(String(50), nullable=False)

    __mapper_args__ = {
        'version_id_col':version_uuid,
        'version_id_generator':lambda version: uuid.uuid4().hex
    }
```

The persistence engine will call upon `uuid.uuid4()` each time a `User` object is subject to an INSERT or an UPDATE. In this case, our version generation function can disregard the incoming value of `version`, as the `uuid4()` function generates identifiers without any prerequisite value. If we were using a sequential versioning scheme such as numeric or a special character system, we could make use of the given `version` in order to help determine the subsequent value.

See also:

custom_guid_type

Server Side Version Counters

The `version_id_generator` can also be configured to rely upon a value that is generated by the database. In this case, the database would need some means of generating new identifiers when a row is subject to an INSERT as well as with an UPDATE. For the UPDATE case, typically an update trigger is needed, unless the database in question supports some other native version identifier. The PostgreSQL database in particular supports a system column called xmin which provides UPDATE versioning. We can make use of the PostgreSQL `xmin` column to version our `User` class as follows:

```
from sqlalchemy import FetchedValue

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String(50), nullable=False)
    xmin = Column("xmin", Integer, system=True, server_default=FetchedValue())

    __mapper_args__ = {
        'version_id_col': xmin,
        'version_id_generator': False
    }
```

With the above mapping, the ORM will rely upon the `xmin` column for automatically providing the new value of the version id counter.

creating tables that refer to system columns

In the above scenario, as `xmin` is a system column provided by PostgreSQL, we use the `system=True` argument to mark it as a system-provided column, omitted from the `CREATE TABLE` statement.

The ORM typically does not actively fetch the values of database-generated values when it emits an INSERT or UPDATE, instead leaving these columns as "expired" and to be fetched when they are next accessed, unless the `eager_defaults mapper()` flag is set. However, when a server side version column is used, the ORM needs to actively fetch the newly generated value. This is so that the version counter is set up *before* any concurrent transaction may update it again. This fetching is also best done simultaneously within the INSERT or UPDATE statement using RETURNING, otherwise if emitting a SELECT statement afterwards, there is still a potential race condition where the version counter may change before it can be fetched.

When the target database supports RETURNING, an INSERT statement for our `User` class will look like this:

```
INSERT INTO "user" (name) VALUES (%(name)s) RETURNING "user".id, "user".xmin
{'name': 'ed'}
```

Where above, the ORM can acquire any newly generated primary key values along with server-generated version identifiers in one statement. When the backend does not support RETURNING, an additional SELECT must be emitted for **every** INSERT and UPDATE, which is much less efficient, and also introduces the possibility of missed version counters:

```
INSERT INTO "user" (name) VALUES (%(name)s)
{'name': 'ed'}

SELECT "user".version_id AS user_version_id FROM "user" where
"user".id = :param_1
{"param_1": 1}
```

It is *strongly recommended* that server side version counters only be used when absolutely necessary and only on backends that support RETURNING, e.g. PostgreSQL, Oracle, SQL Server (though SQL Server has major caveats when triggers are used), Firebird.

New in version 0.9.0: Support for server side version identifier tracking.

Programmatic or Conditional Version Counters

When `version_id_generator` is set to False, we can also programmatically (and conditionally) set the version identifier on our object in the same way we assign any other mapped attribute. Such as if we used our UUID example, but set `version_id_generator` to `False`, we can set the version identifier at our choosing:

```
import uuid

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    version_uuid = Column(String(32), nullable=False)
    name = Column(String(50), nullable=False)

    __mapper_args__ = {
        'version_id_col':version_uuid,
        'version_id_generator': False
    }

u1 = User(name='u1', version_uuid=uuid.uuid4())

session.add(u1)

session.commit()

u1.name = 'u2'
u1.version_uuid = uuid.uuid4()

session.commit()
```

We can update our `User` object without incrementing the version counter as well; the value of the counter will remain unchanged, and the UPDATE statement will still check against the previous value. This may be useful for schemes where only certain classes of UPDATE are sensitive to concurrency issues:

```
# will leave version_uuid unchanged
u1.name = 'u3'
session.commit()
```

New in version 0.9.0: Support for programmatic and conditional version identifier tracking.

2.2.6 Class Mapping API

sqlalchemy.orm.mapper(*class_*, *local_table=None*, *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*, *polymorphic_load=None*, *allow_partial_pks=True*, *batch=True*, *column_prefix=None*, *include_properties=None*, *exclude_properties=None*, *passive_updates=True*, *passive_deletes=False*, *confirm_deleted_rows=True*, *eager_defaults=False*, *legacy_is_orphan=False*, *_compiled_cache_size=100*)

Return a new `Mapper` object.

This function is typically used behind the scenes via the Declarative extension. When using Declarative, many of the usual `mapper()` arguments are handled by the Declarative extension itself, including `class_`, `local_table`, `properties`, and `inherits`. Other options are passed to `mapper()` using the `__mapper_args__` class variable:

```
class MyClass(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    type = Column(String(50))
    alt = Column("some_alt", Integer)

    __mapper_args__ = {
        'polymorphic_on' : type
    }
```

Explicit use of `mapper()` is often referred to as *classical mapping*. The above declarative example is equivalent in classical form to:

```
my_table = Table("my_table", metadata,
    Column('id', Integer, primary_key=True),
    Column('type', String(50)),
    Column("some_alt", Integer)
)

class MyClass(object):
    pass

mapper(MyClass, my_table,
    polymorphic_on=my_table.c.type,
    properties={
        'alt':my_table.c.some_alt
    })
```

See also:

classical_mapping - discussion of direct usage of `mapper()`

Parameters

- **`class_`** – The class to be mapped. When using Declarative, this argument is automatically passed as the declared class itself.
- **`local_table`** – The `Table` or other selectable to which the class is mapped. May be `None` if this mapper inherits from another mapper using single-table inheritance. When using Declarative, this argument is automatically passed by the extension, based on what is configured via the `__table__` argument or via

the `Table` produced as a result of the `__tablename__` and `Column` arguments present.

- **`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_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`** – Defaults to `True`, indicating 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. This is used in the extremely rare case that a `MapperEvents` listener requires being called in between individual row persistence operations.
- **`column_prefix`** – A string which will be prepended to the mapped attribute name when `Column` objects are automatically assigned as attributes to the mapped class. Does not affect explicitly specified column-based properties.

  See the section column_prefix for an example.
- **`concrete`** – If True, indicates this mapper should use concrete table inheritance with its parent mapper.

  See the section concrete_inheritance for an example.
- **`confirm_deleted_rows`** – defaults to True; when a DELETE occurs of one more rows based on specific primary keys, a warning is emitted when the number of rows matched does not equal the number of rows expected. This parameter may be set to False to handle the case where database ON DELETE CASCADE rules may be deleting some of those rows automatically. The warning may be changed to an exception in a future release.

  New in version 0.9.4: - added `mapper.confirm_deleted_rows` as well as conditional matched row checking on delete.
- **`eager_defaults`** – if True, the ORM will immediately fetch the value of server-generated default values after an INSERT or UPDATE, rather than leaving them as expired to be fetched on next access. This can be used for event schemes where the server-generated values are needed immediately before the flush completes. By default, this scheme will emit an individual `SELECT` statement per row inserted or updated, which note can add significant performance overhead. However, if the target database supports RETURNING, the default values will be returned inline with the INSERT or UPDATE statement, which can greatly enhance performance for an application that needs frequent access to just-generated server defaults.

  Changed in version 0.9.0: The `eager_defaults` option can now make use of RETURNING for backends which support it.
- **`exclude_properties`** – A list or set of string column names to be excluded from mapping.

  See include_exclude_cols for an example.
- **`extension`** – A `MapperExtension` instance or list of `MapperExtension` instances which will be applied to all operations by this `Mapper`. **Deprecated.** Please see `MapperEvents`.

- **include_properties** – An inclusive list or set of string column names to map.

  See include_exclude_cols for an example.

- **inherits** – A mapped class or the corresponding `Mapper` of one indicating a superclass to which this `Mapper` should *inherit* from. The mapped class here must be a subclass of the other mapper's class. When using Declarative, this argument is passed automatically as a result of the natural class hierarchy of the declared classes.

  See also:

  inheritance_toplevel

- **inherit_condition** – For joined table inheritance, a SQL expression 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 columns present are missing a `ForeignKey` configuration, this parameter can be used to specify which columns are "foreign". In most cases can be left as `None`.

- **legacy_is_orphan** – Boolean, defaults to `False`. When `True`, specifies that "legacy" orphan consideration is to be applied to objects mapped by this mapper, which means that a pending (that is, not persistent) object is auto-expunged from an owning `Session` only when it is de-associated from *all* parents that specify a `delete-orphan` cascade towards this mapper. The new default behavior is that the object is auto-expunged when it is de-associated with *any* of its parents that specify `delete-orphan` cascade. This behavior is more consistent with that of a persistent object, and allows behavior to be consistent in more scenarios independently of whether or not an orphanable object has been flushed yet or not.

  See the change note and example at legacy_is_orphan_addition for more detail on this change.

  New in version 0.8: - the consideration of a pending object as an "orphan" has been modified to more closely match the behavior as that of persistent objects, which is that the object is expunged from the `Session` as soon as it is de-associated from any of its orphan-enabled parents. Previously, the pending object would be expunged only if de-associated from all of its orphan-enabled parents. The new flag `legacy_is_orphan` is added to `orm.mapper()` which re-establishes the legacy behavior.

- **non_primary** – Specify that this `Mapper` is in addition to the "primary" mapper, that is, the one used for persistence. The `Mapper` created here may be used for ad-hoc mapping of the class to an alternate selectable, for loading only.

  `Mapper.non_primary` is not an often used option, but is useful in some specific `relationship()` cases.

  See also:

  relationship_non_primary_mapper

- **order_by** – A single `Column` or list of `Column` objects for which selection operations should use as the default ordering for entities. By default mappers have no pre-defined ordering.

  Deprecated since version 1.1: The `Mapper.order_by` parameter is deprecated. Use `Query.order_by()` to determine the ordering of a result set.

- **passive_deletes** – Indicates DELETE behavior of foreign key columns when a joined-table inheritance entity is being deleted. Defaults to `False` for a base mapper; for an inheriting mapper, defaults to `False` unless the value is set to `True` on the superclass mapper.

When `True`, it is assumed that ON DELETE CASCADE is configured on the foreign key relationships that link this mapper's table to its superclass table, so that when the unit of work attempts to delete the entity, it need only emit a DELETE statement for the superclass table, and not this table.

When `False`, a DELETE statement is emitted for this mapper's table individually. If the primary key attributes local to this table are unloaded, then a SELECT must be emitted in order to validate these attributes; note that the primary key columns of a joined-table subclass are not part of the "primary key" of the object as a whole.

Note that a value of `True` is **always** forced onto the subclass mappers; that is, it's not possible for a superclass to specify passive_deletes without this taking effect for all subclass mappers.

New in version 1.1.

See also:

passive_deletes - description of similar feature as used with `relationship()`

`mapper.passive_updates` - supporting ON UPDATE CASCADE for joined-table inheritance mappers

- **`passive_updates`** – Indicates UPDATE behavior of foreign key columns when a primary key column changes on a joined-table inheritance mapping. Defaults to `True`.

  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 columns on joined-table rows.

  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 unit of work process will emit an UPDATE statement for the dependent columns during a primary key change.

  See also:

  passive_updates - description of a similar feature as used with `relationship()`

  `mapper.passive_deletes` - supporting ON DELETE CASCADE for joined-table inheritance mappers

- **`polymorphic_load`** –

  Specifies "polymorphic loading" behavior for a subclass in an inheritance hierarchy (joined and single table inheritance only). Valid values are:

  – "'inline'" - specifies this class should be part of the "with_polymorphic" mappers, e.g. its columns will be included in a SELECT query against the base.

  – "'selectin'" - specifies that when instances of this class are loaded, an additional SELECT will be emitted to retrieve the columns specific to this subclass. The SELECT uses IN to fetch multiple subclasses at once.

  New in version 1.2.

  See also:

  with_polymorphic_mapper_config

  polymorphic_selectin

- **`polymorphic_on`** – Specifies the column, attribute, or SQL expression used to determine the target class for an incoming row, when inheriting classes are present.

  This value is commonly a `Column` object that's present in the mapped `Table`:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key=True)
    discriminator = Column(String(50))

    __mapper_args__ = {
        "polymorphic_on":discriminator,
        "polymorphic_identity":"employee"
    }
```

It may also be specified as a SQL expression, as in this example where we use the `case()` construct to provide a conditional approach:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key=True)
    discriminator = Column(String(50))

    __mapper_args__ = {
        "polymorphic_on":case([
            (discriminator == "EN", "engineer"),
            (discriminator == "MA", "manager"),
        ], else_="employee"),
        "polymorphic_identity":"employee"
    }
```

It may also refer to any attribute configured with `column_property()`, or to the string name of one:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key=True)
    discriminator = Column(String(50))
    employee_type = column_property(
        case([
            (discriminator == "EN", "engineer"),
            (discriminator == "MA", "manager"),
        ], else_="employee")
    )

    __mapper_args__ = {
        "polymorphic_on":employee_type,
        "polymorphic_identity":"employee"
    }
```

Changed in version 0.7.4: `polymorphic_on` may be specified as a SQL expression, or refer to any attribute configured with `column_property()`, or to the string name of one.

When setting `polymorphic_on` to reference an attribute or expression that's not present in the locally mapped `Table`, yet the value of the discriminator should be persisted to the database, the value of the discriminator is not automatically set on new instances; this must be handled by the user, either through manual means or via event listeners. A typical approach to establishing such a listener looks like:

```
from sqlalchemy import event
from sqlalchemy.orm import object_mapper
```

```
@event.listens_for(Employee, "init", propagate=True)
def set_identity(instance, *arg, **kw):
    mapper = object_mapper(instance)
    instance.discriminator = mapper.polymorphic_identity
```

Where above, we assign the value of `polymorphic_identity` for the mapped class to the `discriminator` attribute, thus persisting the value to the `discriminator` column in the database.

Warning: Currently, **only one discriminator column may be set**, typically on the base-most class in the hierarchy. "Cascading" polymorphic columns are not yet supported.

See also:

inheritance_toplevel

- **`polymorphic_identity`** – Specifies the value which identifies this particular class as returned by the column expression referred to by the `polymorphic_on` setting. As rows are received, the value corresponding to the `polymorphic_on` column expression is compared to this value, indicating which subclass should be used for the newly reconstructed object.
- **`properties`** – A dictionary mapping the string names of object attributes to `MapperProperty` instances, which define the persistence behavior of that attribute. Note that `Column` objects present in the mapped `Table` are automatically placed into `ColumnProperty` instances upon mapping, unless overridden. When using Declarative, this argument is passed automatically, based on all those `MapperProperty` instances declared in the declared class body.
- **`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` that will be used to keep a running version id of rows in the table. This is used to detect concurrent updates or the presence of stale data in a flush. The methodology is to detect if an UPDATE statement does not match the last known version id, a `StaleDataError` exception is thrown. By default, the column must be of `Integer` type, unless `version_id_generator` specifies an alternative version generator.

  See also:

  mapper_version_counter - discussion of version counting and rationale.

- **`version_id_generator`** – Define how new version ids should be generated. Defaults to `None`, which indicates that a simple integer counting scheme be employed. To provide a custom versioning scheme, provide a callable function of the form:

```
def generate_version(version):
    return next_version
```

  Alternatively, server-side versioning functions such as triggers, or programmatic versioning schemes outside of the version id generator may be used, by specifying the value `False`. Please see server_side_version_counter for a discussion of important points when using this option.

  New in version 0.9.0: `version_id_generator` supports server-side version number generation.

  See also:

custom_version_counter

server_side_version_counter

- **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.

  See also:

  with_polymorphic - discussion of polymorphic querying techniques.

sqlalchemy.orm.**object_mapper**(*instance*)

Given an object, return the primary Mapper associated with the object instance.

Raises sqlalchemy.orm.exc.UnmappedInstanceError if no mapping is configured.

This function is available via the inspection system as:

```
inspect(instance).mapper
```

Using the inspection system will raise sqlalchemy.exc.NoInspectionAvailable if the instance is not part of a mapping.

sqlalchemy.orm.**class_mapper**(*class_*, *configure=True*)

Given a class, return the primary Mapper associated with the key.

Raises UnmappedClassError if no mapping is configured on the given class, or ArgumentError if a non-class object is passed.

Equivalent functionality is available via the inspect() function as:

```
inspect(some_mapped_class)
```

Using the inspection system will raise sqlalchemy.exc.NoInspectionAvailable if the class is not mapped.

sqlalchemy.orm.**configure_mappers**()

Initialize the inter-mapper relationships of all mappers that have been constructed thus far.

This function can be called any number of times, but in most cases is invoked automatically, the first time mappings are used, as well as whenever mappings are used and additional not-yet-configured mappers have been constructed.

Points at which this occur include when a mapped class is instantiated into an instance, as well as when the Session.query() method is used.

The configure_mappers() function provides several event hooks that can be used to augment its functionality. These methods include:

- MapperEvents.before_configured() - called once before configure_mappers() does any work; this can be used to establish additional options, properties, or related mappings before the operation proceeds.
- MapperEvents.mapper_configured() - called as each indivudal Mapper is configured within the process; will include all mapper state except for backrefs set up by other mappers that are still to be configured.
- MapperEvents.after_configured() - called once after configure_mappers() is complete; at this stage, all Mapper objects that are known to SQLAlchemy will be fully configured. Note that the calling application may still have other mappings that haven't been produced yet, such as if they are in modules as yet unimported.

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)
Generate "identity key" tuples, as are used as keys in the Session.identity_map dictionary.

This function has several call styles:

- identity_key(class, ident, identity_token=token)

  This form receives a mapped class and a primary key scalar or tuple as an argument.

  E.g.:

```
>>> identity_key(MyClass, (1, 2))
(<class '__main__.MyClass'>, (1, 2), None)
```

  param class mapped class (must be a positional argument)

  param ident primary key, may be a scalar or tuple argument.

  ;param identity_token: optional identity token

  New in version 1.2: added identity_token

- identity_key(instance=instance)

  This form will produce the identity key for a given instance. The instance need not be persistent, only that its primary key attributes are populated (else the key will contain None for those missing values).

  E.g.:

```
>>> instance = MyClass(1, 2)
>>> identity_key(instance=instance)
(<class '__main__.MyClass'>, (1, 2), None)
```

  In this form, the given instance is ultimately run though Mapper.identity_key_from_instance(), which will have the effect of performing a database check for the corresponding row if the object is expired.

  param instance object instance (must be given as a keyword arg)

- identity_key(class, row=row, identity_token=token)

  This form is similar to the class/tuple form, except is passed a database result row as a RowProxy object.

  E.g.:

```
>>> row = engine.execute("select * from table where a=1 and b=2").first()
>>> identity_key(MyClass, row=row)
(<class '__main__.MyClass'>, (1, 2), None)
```

param class mapped class (must be a positional argument)

param row `RowProxy` row returned by a `ResultProxy` (must be given as a keyword arg)

;param identity_token: optional identity token

New in version 1.2: added identity_token

`sqlalchemy.orm.util.polymorphic_union`(*table_map*, *typecolname*, *aliasname='p_union'*, *cast_nulls=True*)

Create a `UNION` statement used by a polymorphic mapper.

See concrete_inheritance for an example of how this is used.

Parameters

- **`table_map`** – mapping of polymorphic identities to `Table` objects.
- **`typecolname`** – string name of a "discriminator" column, which will be derived from the query, producing the polymorphic identity for each row. If `None`, no polymorphic discriminator is generated.
- **`aliasname`** – name of the `alias()` construct generated.
- **`cast_nulls`** – if True, non-existent columns, which are represented as labeled NULLs, will be passed into CAST. This is a legacy behavior that is problematic on some backends such as Oracle - in which case it can be set to False.

`class sqlalchemy.orm.mapper.Mapper`(*class_*, *local_table=None*, *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*, *polymorphic_load=None*, *allow_partial_pks=True*, *batch=True*, *column_prefix=None*, *include_properties=None*, *exclude_properties=None*, *passive_updates=True*, *passive_deletes=False*, *confirm_deleted_rows=True*, *eager_defaults=False*, *legacy_is_orphan=False*, *_compiled_cache_size=100*)

Define the correlation of class attributes to database table columns.

The `Mapper` object is instantiated using the `mapper()` function. For information about instantiating new `Mapper` objects, see that function's documentation.

When `mapper()` is used explicitly to link a user defined class with table metadata, this is referred to as *classical mapping*. Modern SQLAlchemy usage tends to favor the `sqlalchemy.ext.declarative` extension for class configuration, which makes usage of `mapper()` behind the scenes.

Given a particular class known to be mapped by the ORM, the `Mapper` which maintains it can be acquired using the `inspect()` function:

```
from sqlalchemy import inspect

mapper = inspect(MyClass)
```

A class which was mapped by the `sqlalchemy.ext.declarative` extension will also have its mapper available via the `__mapper__` attribute.

`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 configured yet, just adds the property to the initial properties dictionary sent to the constructor. If this Mapper has already been configured, then the given MapperProperty is configured immediately.

`all_orm_descriptors`
A namespace of all `InspectionAttr` attributes associated with the mapped class.

These attributes are in all cases Python descriptors associated with the mapped class or its superclasses.

This namespace includes attributes that are mapped to the class as well as attributes declared by extension modules. It includes any Python descriptor type that inherits from `InspectionAttr`. This includes `QueryableAttribute`, as well as extension types such as `hybrid_property`, `hybrid_method` and `AssociationProxy`.

To distinguish between mapped attributes and extension attributes, the attribute `InspectionAttr.extension_type` will refer to a constant that distinguishes between different extension types.

When dealing with a `QueryableAttribute`, the `QueryableAttribute.property` attribute refers to the `MapperProperty` property, which is what you get when referring to the collection of mapped properties via `Mapper.attrs`.

> **Warning:** The `Mapper.all_orm_descriptors` accessor namespace is an instance of `OrderedProperties`. This is a dictionary-like object which includes a small number of named methods such as `OrderedProperties.items()` and `OrderedProperties.values()`. When accessing attributes dynamically, favor using the dict-access scheme, e.g. `mapper.all_orm_descriptors[somename]` over `getattr(mapper.all_orm_descriptors, somename)` to avoid name collisions.

New in version 0.8.0.

See also:

`Mapper.attrs`

`attrs`
A namespace of all `MapperProperty` objects associated this mapper.

This is an object that provides each property based on its key name. For instance, the mapper for a `User` class which has `User.name` attribute would provide `mapper.attrs.name`, which would be the `ColumnProperty` representing the `name` column. The namespace object can also be iterated, which would yield each `MapperProperty`.

`Mapper` has several pre-filtered views of this attribute which limit the types of properties returned, inclding `synonyms`, `column_attrs`, `relationships`, and `composites`.

> **Warning:** The `Mapper.attrs` accessor namespace is an instance of `OrderedProperties`. This is a dictionary-like object which includes a small number of named methods such as `OrderedProperties.items()` and `OrderedProperties.values()`. When accessing attributes dynamically, favor using the dict-access scheme, e.g. `mapper.attrs[somename]` over `getattr(mapper.attrs, somename)` to avoid name collisions.

See also:

`Mapper.all_orm_descriptors`

`base_mapper = None`
The base-most `Mapper` in an inheritance chain.

In a non-inheriting scenario, this attribute will always be this `Mapper`. In an inheritance scenario, it references the `Mapper` which is parent to all other `Mapper` objects in the inheritance chain.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`c = None`
A synonym for `columns`.

`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.).

  Note: the `"all"` cascade is not accepted here. For a generic object traversal function, see faq_walk_objects.

- `state` – The lead InstanceState. child items will be processed per the relationships defined for this object's mapper.

Returns the method yields individual object instances.

See also:

unitofwork_cascades

faq_walk_objects - illustrates a generic function to traverse all objects without relying on cascades.

`class_ = None`
The Python class which this `Mapper` maps.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`class_manager = None`
The `ClassManager` which maintains event listeners and class-bound descriptors for this `Mapper`.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`column_attrs`
Return a namespace of all `ColumnProperty` properties maintained by this `Mapper`.

See also:

`Mapper.attrs` - namespace of all `MapperProperty` objects.

`columns = None`
A collection of `Column` or other scalar expression objects maintained by this `Mapper`.

The collection behaves the same as that of the `c` attribute on any `Table` object, except that only those columns included in this mapping are present, and are keyed based on the attribute name defined in the mapping, not necessarily the `key` attribute of the `Column` itself. Additionally, scalar expressions mapped by `column_property()` are also present here.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`common_parent(`*other*`)`
Return true if the given mapper shares a common inherited parent as this mapper.

`composites`
Return a namespace of all `CompositeProperty` properties maintained by this `Mapper`.

See also:

`Mapper.attrs` - namespace of all `MapperProperty` objects.

`concrete = None`
Represent `True` if this `Mapper` is a concrete inheritance mapper.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`configured = None`
Represent `True` if this `Mapper` has been configured.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

See also:

`configure_mappers()`.

`entity`
Part of the inspection API.

Returns self.class_.

`get_property`(*key*, *_configure_mappers=True*)
return a MapperProperty associated with the given key.

`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.

If the instance's state is expired, calling this method will result in a database check to see if the object has been deleted. If the row no longer exists, `ObjectDeletedError` is raised.

This value is typically also found on the instance state under the attribute name *key*.

`identity_key_from_primary_key`(*primary_key*, *identity_token=None*)
Return an identity-map key for use in storing/retrieving an item from an identity map.

Parameters `primary_key` – A list of values indicating the identifier.

`identity_key_from_row`(*row*, *identity_token=None*, *adapter=None*)
Return an identity-map key for use in storing/retrieving an item from the identity map.

Parameters `row` – A `RowProxy` instance. The columns which are mapped by this `Mapper` should be locatable in the row, preferably via the `Column` object directly (as is the case when a `select()` construct is executed), or via string names of the form `<tablename>_<colname>`.

`inherits = None`
References the `Mapper` which this `Mapper` inherits from, if any.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`is_mapper = True`
Part of the inspection API.

`isa`(*other*)
Return True if the this mapper inherits from the given mapper.

`iterate_properties`
return an iterator of all MapperProperty objects.

`local_table = None`
The `Selectable` which this `Mapper` manages.

Typically is an instance of `Table` or `Alias`. May also be `None`.

The "local" table is the selectable that the `Mapper` is directly responsible for managing from an attribute access and flush perspective. For non-inheriting mappers, the local table is the same as the "mapped" table. For joined-table inheritance mappers, local_table will be the particular sub-table of the overall "join" which this `Mapper` represents. If this mapper is a single-table inheriting mapper, local_table will be `None`.

See also:

`mapped_table`.

`mapped_table = None`
The `Selectable` to which this `Mapper` is mapped.

Typically an instance of `Table`, `Join`, or `Alias`.

The "mapped" table is the selectable that the mapper selects from during queries. For non-inheriting mappers, the mapped table is the same as the "local" table. For joined-table inheritance mappers, mapped_table references the full `Join` representing full rows for this particular subclass. For single-table inheritance mappers, mapped_table references the base table.

See also:

`local_table`.

`mapper`
Part of the inspection API.

Returns self.

`non_primary = None`
Represent `True` if this `Mapper` is a "non-primary" mapper, e.g. a mapper that is used only to selet rows but not for persistence management.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`polymorphic_identity = None`
Represent an identifier which is matched against the `polymorphic_on` column during result row loading.

Used only with inheritance, this object can be of any type which is comparable to the type of column represented by `polymorphic_on`.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`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()`.

`polymorphic_map = None`
A mapping of "polymorphic identity" identifiers mapped to `Mapper` instances, within an inheritance scenario.

The identifiers can be of any type which is comparable to the type of column represented by `polymorphic_on`.

An inheritance chain of mappers will all reference the same polymorphic map object. The object is used to correlate incoming result rows to target mappers.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`polymorphic_on = None`
The `Column` or SQL expression specified as the `polymorphic_on` argument for this `Mapper`, within an inheritance scenario.

This attribute is normally a `Column` instance but may also be an expression, such as one derived from `cast()`.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`primary_key = None`
An iterable containing the collection of `Column` objects which comprise the 'primary key' of the mapped table, from the perspective of this `Mapper`.

This list is against the selectable in `mapped_table`. In the case of inheriting mappers, some columns may be managed by a superclass mapper. For example, in the case of a `Join`, the primary key is determined by all of the primary key columns across all tables referenced by the `Join`.

The list is also not necessarily the same as the primary key column collection associated with the underlying tables; the `Mapper` features a `primary_key` argument that can override what the `Mapper` considers as primary key columns.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`primary_key_from_instance`(*instance*)
Return the list of primary key values for the given instance.

If the instance's state is expired, calling this method will result in a database check to see if the object has been deleted. If the row no longer exists, `ObjectDeletedError` is raised.

`primary_mapper()`
Return the primary mapper corresponding to this mapper's class key (class).

`relationships`
A namespace of all `RelationshipProperty` properties maintained by this `Mapper`.

> **Warning:** the `Mapper.relationships` accessor namespace is an instance of `OrderedProperties`. This is a dictionary-like object which includes a small number of named methods such as `OrderedProperties.items()` and `OrderedProperties.values()`. When accessing attributes dynamically, favor using the dict-access scheme, e.g. `mapper.relationships[somename]` over `getattr(mapper.relationships, somename)` to avoid name collisions.

See also:

`Mapper.attrs` - namespace of all `MapperProperty` objects.

`selectable`
The `select()` construct this `Mapper` selects from by default.

Normally, this is equivalent to `mapped_table`, unless the `with_polymorphic` feature is in use, in which case the full "polymorphic" selectable is returned.

`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.

`single = None`
Represent `True` if this `Mapper` is a single table inheritance mapper.

`local_table` will be `None` if this flag is set.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`synonyms`
Return a namespace of all `SynonymProperty` properties maintained by this `Mapper`.

See also:

`Mapper.attrs` - namespace of all `MapperProperty` objects.

`tables = None`
An iterable containing the collection of `Table` objects which this `Mapper` is aware of.

If the mapper is mapped to a `Join`, or an `Alias` representing a `Select`, the individual `Table` objects that comprise the full construct will be represented here.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

`validators = None`
An immutable dictionary of attributes which have been decorated using the `validates()` decorator.

The dictionary contains string attribute names as keys mapped to the actual validation method.

`with_polymorphic_mappers`
The list of `Mapper` objects included in the default "polymorphic" query.

2.3 Relationship Configuration

This section describes the `relationship()` function and in depth discussion of its usage. For an introduction to relationships, start with the ormtutorial_toplevel and head into orm_tutorial_relationship.

2.3.1 Basic Relationship Patterns

A quick walkthrough of the basic relational patterns.

The imports used for each of the following sections is as follows:

```
from sqlalchemy import Table, Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

One To Many

A one to many relationship places a foreign key on the child table referencing the parent. `relationship()` is then specified on the parent, as referencing a collection of items represented by the child:

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

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

To establish a bidirectional relationship in one-to-many, where the "reverse" side is a many to one, specify an additional `relationship()` and connect the two using the `relationship.back_populates` parameter:

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

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

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

Alternatively, the `backref` option may be used on a single `relationship()` instead of using `back_populates`:

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

Many To One

Many to one places a foreign key in the parent table referencing the child. `relationship()` is declared on the parent, where a new scalar-holding attribute will be created:

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

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

Bidirectional behavior is achieved by adding a second `relationship()` and applying the `relationship.back_populates` parameter in both directions:

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

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parents = relationship("Parent", back_populates="child")
```

Alternatively, the `backref` parameter may be applied to a single `relationship()`, such as `Parent.child`:

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

One To One

One To One is essentially a bidirectional relationship with a scalar attribute on both sides. To achieve this, the `uselist` 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:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child = relationship("Child", uselist=False, back_populates="parent")

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

Or for many-to-one:

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

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent = relationship("Parent", back_populates="child", uselist=False)
```

As always, the `relationship.backref` and `backref()` functions may be used in lieu of the `relationship.back_populates` approach; to specify `uselist` on a backref, use the `backref()` function:

```
from sqlalchemy.orm import backref

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))
```

Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the `secondary` argument to `relationship()`. Usually, the `Table` uses the `MetaData` object associated with the declarative base class, so that the `ForeignKey` directives can locate the remote tables with which to link:

```
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)

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

For a bidirectional relationship, both sides of the relationship contain a collection. Specify using `relationship.back_populates`, and for each `relationship()` specify the common association table:

```
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,
        back_populates="parents")

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

When using the `backref` parameter instead of `relationship.back_populates`, the backref will automatically use the same `secondary` argument for the reverse relationship:

```
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)
```

The `secondary` argument of `relationship()` also accepts a callable that returns the ultimate argument, which is evaluated only when mappers are first used. Using this, we can define the `association_table` at a later point, as long as it's available to the callable after all module initialization is complete:

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

```
                    secondary=lambda: association_table,
                    backref="parents")
```

With the declarative extension in use, the traditional "string name of the table" is accepted as well, matching the name of the table as stored in `Base.metadata.tables`:

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

Deleting Rows from the Many to Many Table

A behavior which is unique to the `secondary` argument to `relationship()` is that the `Table` which is specified here is automatically subject to INSERT and DELETE statements, as objects are added or removed from the collection. There is **no need to delete from this table manually**. The act of removing a record from the collection will have the effect of the row being deleted on flush:

```
# row will be deleted from the "secondary" table
# automatically
myparent.children.remove(somechild)
```

A question which often arises is how the row in the "secondary" table can be deleted when the child object is handed directly to `Session.delete()`:

```
session.delete(somechild)
```

There are several possibilities here:

- If there is a `relationship()` from `Parent` to `Child`, but there is **not** a reverse-relationship that links a particular `Child` to each `Parent`, SQLAlchemy will not have any awareness that when deleting this particular `Child` object, it needs to maintain the "secondary" table that links it to the `Parent`. No delete of the "secondary" table will occur.
- If there is a relationship that links a particular `Child` to each `Parent`, suppose it's called `Child.parents`, SQLAlchemy by default will load in the `Child.parents` collection to locate all `Parent` objects, and remove each row from the "secondary" table which establishes this link. Note that this relationship does not need to be bidirectional; SQLAlchemy is strictly looking at every `relationship()` associated with the `Child` object being deleted.
- A higher performing option here is to use ON DELETE CASCADE directives with the foreign keys used by the database. Assuming the database supports this feature, the database itself can be made to automatically delete rows in the "secondary" table as referencing rows in "child" are deleted. SQLAlchemy can be instructed to forego actively loading in the `Child.parents` collection in this case using the `passive_deletes` directive on `relationship()`; see passive_deletes for more details on this.

Note again, these behaviors are *only* relevant to the `secondary` option used with `relationship()`. If dealing with association tables that are mapped explicitly and are *not* present in the `secondary` option of a relevant `relationship()`, cascade rules can be used instead to automatically delete entities in reaction to a related entity being deleted - see unitofwork_cascades for information on this feature.

Association Object

The association object pattern is a variant on many-to-many: it's 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. Below we illustrate an association table mapped to the `Association` class which includes a column called `extra_data`, which is a string value that is stored along with each association between `Parent` and `Child`:

```
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)
    extra_data = Column(String(50))
    child = relationship("Child")

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

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

As always, the bidirectional version makes use of `relationship.back_populates` or `relationship.backref`:

```
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)
    extra_data = Column(String(50))
    child = relationship("Child", back_populates="parents")
    parent = relationship("Parent", back_populates="children")

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

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
    parents = relationship("Association", back_populates="child")
```

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(extra_data="some data")
a.child = Child()
p.children.append(a)

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

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

Warning: The association object pattern **does not coordinate changes with a separate relationship that maps the association table as "secondary"**.

Below, changes made to `Parent.children` will not be coordinated with changes made to `Parent.child_associations` or `Child.parent_associations` in Python; while all of these relationships will continue to function normally by themselves, changes on one will not show up in another until the `Session` is expired, which normally occurs automatically after `Session.commit()`:

```
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)
    extra_data = Column(String(50))

    child = relationship("Child", backref="parent_associations")
    parent = relationship("Parent", backref="child_associations")

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)

    children = relationship("Child", secondary="association")

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

Additionally, just as changes to one relationship aren't reflected in the others automatically, writing the same data to both relationships will cause conflicting INSERT or DELETE statements as well, such as below where we establish the same relationship between a `Parent` and `Child` object twice:

```
p1 = Parent()
c1 = Child()
p1.children.append(c1)

# redundant, will cause a duplicate INSERT on Association
p1.parent_associations.append(Association(child=c1))
```

It's fine to use a mapping like the above if you know what you're doing, though it may be a good idea to apply the `viewonly=True` parameter to the "secondary" relationship to avoid the issue of redundant changes being logged. However, to get a foolproof pattern that allows a simple two-object `Parent->Child` relationship while still using the association object pattern, use the association proxy extension as documented at associationproxy_toplevel.

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 way to represent hierarchical data in flat tables. Other methods include **nested sets**, sometimes called "modified preorder", as well as **materialized path**. Despite the appeal that modified preorder has when evaluated for its fluency within SQL queries, 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.

In this example, we'll work with a single mapped class called `Node`, representing a tree structure:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
```

```
    data = Column(String(50))
    children = relationship("Node")
```

With this structure, 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
```

The `relationship()` configuration here works in the same way as a "normal" one-to-many relationship, with the exception that the "direction", i.e. whether the relationship is one-to-many or many-to-one, is assumed by default to be one-to-many. To establish the relationship as many-to-one, an extra directive is added known as `remote_side`, which is a `Column` or collection of `Column` objects that indicate those which should be considered to be "remote":

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    data = Column(String(50))
    parent = relationship("Node", remote_side=[id])
```

Where above, the `id` column is applied as the `remote_side` of the `parent` `relationship()`, thus establishing `parent_id` as the "local" side, and the relationship then behaves as a many-to-one.

As always, both directions can be combined into a bidirectional relationship using the `backref()` function:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.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 examples_adjacencylist and examples_xmlpersistence.

Composite Adjacency Lists

A sub-category of the adjacency list relationship is the rare case where a particular column is present on both the "local" and "remote" side of the join condition. An example is the `Folder` class below; using a composite primary key, the `account_id` column refers to itself, to indicate sub folders which are within the same account as that of the parent; while `folder_id` refers to a specific folder within that account:

```
class Folder(Base):
    __tablename__ = 'folder'
```

```
    __table_args__ = (
      ForeignKeyConstraint(
          ['account_id', 'parent_id'],
          ['folder.account_id', 'folder.folder_id']),
    )

    account_id = Column(Integer, primary_key=True)
    folder_id = Column(Integer, primary_key=True)
    parent_id = Column(Integer)
    name = Column(String)

    parent_folder = relationship("Folder",
                          backref="child_folders",
                          remote_side=[account_id, folder_id]
                    )
```

Above, we pass `account_id` into the `remote_side` list. `relationship()` recognizes that the `account_id` column here is on both sides, and aligns the "remote" column along with the `folder_id` column, which it recognizes as uniquely present on the "remote" side.

New in version 0.8: Support for self-referential composite keys in `relationship()` where a column points to itself.

Self-Referential Query Strategies

Querying of self-referential structures works like any other query:

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

However extra care is needed when attempting to join along the foreign key from one level of the tree to the next. In SQL, a join from a table to itself requires that at least one side of the expression be "aliased" so that it can be unambiguously referred to.

Recall from ormtutorial_aliases in the ORM tutorial that the `orm.aliased()` construct is normally used to provide an "alias" of an ORM entity. Joining from `Node` to itself using this technique looks like:

```
from sqlalchemy.orm import aliased

nodealias = aliased(Node)
{sql}session.query(Node).filter(Node.data=='subchild1').\
                join(nodealias, Node.parent).\
                filter(nodealias.data=="child2").\
                all()
SELECT node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node JOIN node AS node_1
    ON node.parent_id = node_1.id
WHERE node.data = ?
    AND node_1.data = ?
['subchild1', 'child2']
```

`Query.join()` also includes a feature known as `Query.join.aliased` that can shorten the verbosity self- referential joins, at the expense of query flexibility. This feature performs a similar "aliasing" step to that above, without the need for an explicit entity. Calls to `Query.filter()` and similar subsequent to the aliased join will **adapt** the `Node` entity to be that of the alias:

```
{sql}session.query(Node).filter(Node.data=='subchild1').\
        join(Node.parent, aliased=True).\
        filter(Node.data=='child2').\
```

```
        all()
SELECT node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node
    JOIN node AS node_1 ON node_1.id = node.parent_id
WHERE node.data = ? AND node_1.data = ?
['subchild1', 'child2']
```

To add criterion to multiple points along a longer join, add `Query.join.from_joinpoint` to the additional `join()` calls:

```
# get all nodes named 'subchild1' with a
# parent named 'child2' and a grandparent 'root'
{sql}session.query(Node).\
        filter(Node.data=='subchild1').\
        join(Node.parent, aliased=True).\
        filter(Node.data=='child2').\
        join(Node.parent, aliased=True, from_joinpoint=True).\
        filter(Node.data=='root').\
        all()
SELECT node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node
    JOIN node AS node_1 ON node_1.id = node.parent_id
    JOIN node AS node_2 ON node_2.id = node_1.parent_id
WHERE node.data = ?
    AND node_1.data = ?
    AND node_2.data = ?
['subchild1', 'child2', 'root']
```

`Query.reset_joinpoint()` will also remove the "aliasing" from filtering calls:

```
session.query(Node).\
        join(Node.children, aliased=True).\
        filter(Node.data == 'foo').\
        reset_joinpoint().\
        filter(Node.data == 'bar')
```

For an example of using `Query.join.aliased` to arbitrarily join along a chain of self-referential nodes, see examples_xmlpersistence.

Configuring Self-Referential 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 immediate child collection or reference can be populated from a single SQL statement, or a second statement for all immediate child collections. SQLAlchemy's joined and subquery eager loading use aliased tables in all cases when joining to related items, so are 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 and/or query; otherwise the eager load will not take place at all. This depth setting is configured via `join_depth`:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    data = Column(String(50))
    children = relationship("Node",
                    lazy="joined",
                    join_depth=2)
```

```
{sql}session.query(Node).all()
SELECT node_1.id AS node_1_id,
        node_1.parent_id AS node_1_parent_id,
        node_1.data AS node_1_data,
        node_2.id AS node_2_id,
        node_2.parent_id AS node_2_parent_id,
        node_2.data AS node_2_data,
        node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node
    LEFT OUTER JOIN node AS node_2
        ON node.id = node_2.parent_id
    LEFT OUTER JOIN node AS node_1
        ON node_2.id = node_1.parent_id
[]
```

2.3.3 Linking Relationships with Backref

The `backref` keyword argument was first introduced in ormtutorial_toplevel, and has been mentioned throughout many of the examples here. What does it actually do ? Let's start with the canonical `User` and `Address` scenario:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

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

    addresses = relationship("Address", backref="user")

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

The above configuration establishes a collection of `Address` objects on `User` called `User.addresses`. It also establishes a `.user` attribute on `Address` which will refer to the parent `User` object.

In fact, the `backref` keyword is only a common shortcut for placing a second `relationship()` onto the `Address` mapping, including the establishment of an event listener on both sides which will mirror attribute operations in both directions. The above configuration is equivalent to:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

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

```
    addresses = relationship("Address", back_populates="user")

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

    user = relationship("User", back_populates="addresses")
```

Above, we add a `.user` relationship to `Address` explicitly. On both relationships, the `back_populates` directive tells each relationship about the other one, indicating that they should establish "bidirectional" behavior between each other. The primary effect of this configuration is that the relationship adds event handlers to both attributes which have the behavior of "when an append or set event occurs here, set ourselves onto the incoming attribute using this particular attribute name". The behavior is illustrated as follows. Start with a `User` and an `Address` instance. The `.addresses` collection is empty, and the `.user` attribute is `None`:

```
>>> u1 = User()
>>> a1 = Address()
>>> u1.addresses
[]
>>> print(a1.user)
None
```

However, once the `Address` is appended to the `u1.addresses` collection, both the collection and the scalar attribute have been populated:

```
>>> u1.addresses.append(a1)
>>> u1.addresses
[<__main__.Address object at 0x12a6ed0>]
>>> a1.user
<__main__.User object at 0x12a6590>
```

This behavior of course works in reverse for removal operations as well, as well as for equivalent operations on both sides. Such as when `.user` is set again to `None`, the `Address` object is removed from the reverse collection:

```
>>> a1.user = None
>>> u1.addresses
[]
```

The manipulation of the `.addresses` collection and the `.user` attribute occurs entirely in Python without any interaction with the SQL database. Without this behavior, the proper state would be apparent on both sides once the data has been flushed to the database, and later reloaded after a commit or expiration operation occurs. The `backref`/`back_populates` behavior has the advantage that common bidirectional operations can reflect the correct state without requiring a database round trip.

Remember, when the `backref` keyword is used on a single relationship, it's exactly the same as if the above two relationships were created individually using `back_populates` on each.

Backref Arguments

We've established that the `backref` keyword is merely a shortcut for building two individual `relationship()` constructs that refer to each other. Part of the behavior of this shortcut is that certain configurational arguments applied to the `relationship()` will also be applied to the other direction - namely those arguments that describe the relationship at a schema level, and are unlikely to be different in the reverse direction. The usual case here is a many-to-many `relationship()` that has a `secondary` argument, or a one-to-many or many-to-one which has a `primaryjoin` argument (the `primaryjoin` ar-

gument is discussed in relationship_primaryjoin). Such as if we limited the list of `Address` objects to those which start with "tony":

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

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

    addresses = relationship("Address",
                    primaryjoin="and_(User.id==Address.user_id, "
                        "Address.email.startswith('tony'))",
                    backref="user")

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

We can observe, by inspecting the resulting property, that both sides of the relationship have this join condition applied:

```
>>> print(User.addresses.property.primaryjoin)
"user".id = address.user_id AND address.email LIKE :email_1 || '%%'
>>>
>>> print(Address.user.property.primaryjoin)
"user".id = address.user_id AND address.email LIKE :email_1 || '%%'
>>>
```

This reuse of arguments should pretty much do the "right thing" - it uses only arguments that are applicable, and in the case of a many-to- many relationship, will reverse the usage of `primaryjoin` and `secondaryjoin` to correspond to the other direction (see the example in self_referential_many_to_many for this).

It's very often the case however that we'd like to specify arguments that are specific to just the side where we happened to place the "backref". This includes `relationship()` arguments like `lazy`, `remote_side`, `cascade` and `cascade_backrefs`. For this case we use the `backref()` function in place of a string:

```
# <other imports>
from sqlalchemy.orm import backref

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

    addresses = relationship("Address",
                    backref=backref("user", lazy="joined"))
```

Where above, we placed a `lazy="joined"` directive only on the `Address.user` side, indicating that when a query against `Address` is made, a join to the `User` entity should be made automatically which will populate the `.user` attribute of each returned `Address`. The `backref()` function formatted the arguments we gave it into a form that is interpreted by the receiving `relationship()` as additional arguments to be applied to the new relationship it creates.

One Way Backrefs

An unusual case is that of the "one way backref". This is where the "back-populating" behavior of the backref is only desirable in one direction. An example of this is a collection which contains a filtering `primaryjoin` condition. We'd like to append items to this collection as needed, and have them populate the "parent" object on the incoming object. However, we'd also like to have items that are not part of the collection, but still have the same "parent" association - these items should never be in the collection.

Taking our previous example, where we established a `primaryjoin` that limited the collection only to `Address` objects whose email address started with the word `tony`, the usual backref behavior is that all items populate in both directions. We wouldn't want this behavior for a case like the following:

```
>>> u1 = User()
>>> a1 = Address(email='mary')
>>> a1.user = u1
>>> u1.addresses
[<__main__.Address object at 0x1411910>]
```

Above, the `Address` object that doesn't match the criterion of "starts with 'tony'" is present in the `addresses` collection of `u1`. After these objects are flushed, the transaction committed and their attributes expired for a re-load, the `addresses` collection will hit the database on next access and no longer have this `Address` object present, due to the filtering condition. But we can do away with this unwanted side of the "backref" behavior on the Python side by using two separate `relationship()` constructs, placing `back_populates` only on one side:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    addresses = relationship("Address",
                    primaryjoin="and_(User.id==Address.user_id, "
                        "Address.email.startswith('tony'))",
                    back_populates="user")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(Integer, ForeignKey('user.id'))
    user = relationship("User")
```

With the above scenario, appending an `Address` object to the `.addresses` collection of a `User` will always establish the `.user` attribute on that `Address`:

```
>>> u1 = User()
>>> a1 = Address(email='tony')
>>> u1.addresses.append(a1)
>>> a1.user
<__main__.User object at 0x1411850>
```

However, applying a `User` to the `.user` attribute of an `Address`, will not append the `Address` object to the collection:

```
>>> a2 = Address(email='mary')
>>> a2.user = u1
```

```
>>> a2 in u1.addresses
False
```

Of course, we've disabled some of the usefulness of `backref` here, in that when we do append an `Address` that corresponds to the criteria of `email.startswith('tony')`, it won't show up in the `User.addresses` collection until the session is flushed, and the attributes reloaded after a commit or expire operation. While we could consider an attribute event that checks this criterion in Python, this starts to cross the line of duplicating too much SQL behavior in Python. The backref behavior itself is only a slight transgression of this philosophy - SQLAlchemy tries to keep these to a minimum overall.

2.3.4 Configuring how Relationship Joins

`relationship()` will normally create a join between two tables by examining the foreign key relationship between the two tables to determine which columns should be compared. There are a variety of situations where this behavior needs to be customized.

Handling Multiple Join Paths

One of the most common situations to deal with is when there are more than one foreign key path between two tables.

Consider a `Customer` class that contains two foreign keys to an `Address` class:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Customer(Base):
    __tablename__ = 'customer'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    billing_address_id = Column(Integer, ForeignKey("address.id"))
    shipping_address_id = Column(Integer, ForeignKey("address.id"))

    billing_address = relationship("Address")
    shipping_address = relationship("Address")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    street = Column(String)
    city = Column(String)
    state = Column(String)
    zip = Column(String)
```

The above mapping, when we attempt to use it, will produce the error:

```
sqlalchemy.exc.AmbiguousForeignKeysError: Could not determine join
condition between parent/child tables on relationship
Customer.billing_address - there are multiple foreign key
paths linking the tables.  Specify the 'foreign_keys' argument,
providing a list of those columns which should be
counted as containing a foreign key reference to the parent table.
```

The above message is pretty long. There are many potential messages that `relationship()` can return, which have been carefully tailored to detect a variety of common configurational issues; most will suggest the additional configuration that's needed to resolve the ambiguity or other missing information.

In this case, the message wants us to qualify each `relationship()` by instructing for each one which foreign key column should be considered, and the appropriate form is as follows:

```
class Customer(Base):
    __tablename__ = 'customer'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    billing_address_id = Column(Integer, ForeignKey("address.id"))
    shipping_address_id = Column(Integer, ForeignKey("address.id"))

    billing_address = relationship("Address", foreign_keys=[billing_address_id])
    shipping_address = relationship("Address", foreign_keys=[shipping_address_id])
```

Above, we specify the `foreign_keys` argument, which is a `Column` or list of `Column` objects which indicate those columns to be considered "foreign", or in other words, the columns that contain a value referring to a parent table. Loading the `Customer.billing_address` relationship from a `Customer` object will use the value present in `billing_address_id` in order to identify the row in `Address` to be loaded; similarly, `shipping_address_id` is used for the `shipping_address` relationship. The linkage of the two columns also plays a role during persistence; the newly generated primary key of a just-inserted `Address` object will be copied into the appropriate foreign key column of an associated `Customer` object during a flush.

When specifying `foreign_keys` with Declarative, we can also use string names to specify, however it is important that if using a list, the **list is part of the string**:

```
billing_address = relationship("Address", foreign_keys="[Customer.billing_address_id]")
```

In this specific example, the list is not necessary in any case as there's only one `Column` we need:

```
billing_address = relationship("Address", foreign_keys="Customer.billing_address_id")
```

Changed in version 0.8: `relationship()` can resolve ambiguity between foreign key targets on the basis of the `foreign_keys` argument alone; the `primaryjoin` argument is no longer needed in this situation.

Specifying Alternate Join Conditions

The default behavior of `relationship()` when constructing a join is that it equates the value of primary key columns on one side to that of foreign-key-referring columns on the other. We can change this criterion to be anything we'd like using the `primaryjoin` argument, as well as the `secondaryjoin` argument in the case when a "secondary" table is used.

In the example below, using the `User` class as well as an `Address` class which stores a street address, we create a relationship `boston_addresses` which will only load those `Address` objects which specify a city of "Boston":

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    boston_addresses = relationship("Address",
                    primaryjoin="and_(User.id==Address.user_id, "
                        "Address.city=='Boston')")

class Address(Base):
    __tablename__ = 'address'
```

```
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'))

    street = Column(String)
    city = Column(String)
    state = Column(String)
    zip = Column(String)
```

Within this string SQL expression, we made use of the `and_()` conjunction construct to establish two distinct predicates for the join condition - joining both the `User.id` and `Address.user_id` columns to each other, as well as limiting rows in `Address` to just `city='Boston'`. When using Declarative, rudimentary SQL functions like `and_()` are automatically available in the evaluated namespace of a string `relationship()` argument.

The custom criteria we use in a `primaryjoin` is generally only significant when SQLAlchemy is rendering SQL in order to load or represent this relationship. That is, it's used in the SQL statement that's emitted in order to perform a per-attribute lazy load, or when a join is constructed at query time, such as via `Query.join()`, or via the eager "joined" or "subquery" styles of loading. When in-memory objects are being manipulated, we can place any `Address` object we'd like into the `boston_addresses` collection, regardless of what the value of the `.city` attribute is. The objects will remain present in the collection until the attribute is expired and re-loaded from the database where the criterion is applied. When a flush occurs, the objects inside of `boston_addresses` will be flushed unconditionally, assigning value of the primary key `user.id` column onto the foreign-key-holding `address.user_id` column for each row. The `city` criteria has no effect here, as the flush process only cares about synchronizing primary key values into referencing foreign key values.

Creating Custom Foreign Conditions

Another element of the primary join condition is how those columns considered "foreign" are determined. Usually, some subset of `Column` objects will specify `ForeignKey`, or otherwise be part of a `ForeignKeyConstraint` that's relevant to the join condition. `relationship()` looks to this foreign key status as it decides how it should load and persist data for this relationship. However, the `primaryjoin` argument can be used to create a join condition that doesn't involve any "schema" level foreign keys. We can combine `primaryjoin` along with `foreign_keys` and `remote_side` explicitly in order to establish such a join.

Below, a class `HostEntry` joins to itself, equating the string `content` column to the `ip_address` column, which is a PostgreSQL type called `INET`. We need to use `cast()` in order to cast one side of the join to the type of the other:

```
from sqlalchemy import cast, String, Column, Integer
from sqlalchemy.orm import relationship
from sqlalchemy.dialects.postgresql import INET

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class HostEntry(Base):
    __tablename__ = 'host_entry'

    id = Column(Integer, primary_key=True)
    ip_address = Column(INET)
    content = Column(String(50))

    # relationship() using explicit foreign_keys, remote_side
    parent_host = relationship("HostEntry",
                        primaryjoin=ip_address == cast(content, INET),
                        foreign_keys=content,
```

```
                        remote_side=ip_address
                    )
```

The above relationship will produce a join like:

```
SELECT host_entry.id, host_entry.ip_address, host_entry.content
FROM host_entry JOIN host_entry AS host_entry_1
ON host_entry_1.ip_address = CAST(host_entry.content AS INET)
```

An alternative syntax to the above is to use the `foreign()` and `remote()` annotations, inline within the `primaryjoin` expression. This syntax represents the annotations that `relationship()` normally applies by itself to the join condition given the `foreign_keys` and `remote_side` arguments. These functions may be more succinct when an explicit join condition is present, and additionally serve to mark exactly the column that is "foreign" or "remote" independent of whether that column is stated multiple times or within complex SQL expressions:

```
from sqlalchemy.orm import foreign, remote

class HostEntry(Base):
    __tablename__ = 'host_entry'

    id = Column(Integer, primary_key=True)
    ip_address = Column(INET)
    content = Column(String(50))

    # relationship() using explicit foreign() and remote() annotations
    # in lieu of separate arguments
    parent_host = relationship("HostEntry",
                        primaryjoin=remote(ip_address) == \
                                cast(foreign(content), INET),
                    )
```

Using custom operators in join conditions

Another use case for relationships is the use of custom operators, such as PostgreSQL's "is contained within" `<<` operator when joining with types such as *postgresql.INET* and *postgresql.CIDR*. For custom operators we use the `Operators.op()` function:

```
inet_column.op("<<")(cidr_column)
```

However, if we construct a `primaryjoin` using this operator, `relationship()` will still need more information. This is because when it examines our primaryjoin condition, it specifically looks for operators used for **comparisons**, and this is typically a fixed list containing known comparison operators such as `==`, `<`, etc. So for our custom operator to participate in this system, we need it to register as a comparison operator using the `is_comparison` parameter:

```
inet_column.op("<<", is_comparison=True)(cidr_column)
```

A complete example:

```
class IPA(Base):
    __tablename__ = 'ip_address'

    id = Column(Integer, primary_key=True)
    v4address = Column(INET)

    network = relationship("Network",
                        primaryjoin="IPA.v4address.op('<<', is_comparison=True)"
                            "(foreign(Network.v4representation))",
                        viewonly=True
```

```
                    )
class Network(Base):
    __tablename__ = 'network'

    id = Column(Integer, primary_key=True)
    v4representation = Column(CIDR)
```

Above, a query such as:

```
session.query(IPA).join(IPA.network)
```

Will render as:

```
SELECT ip_address.id AS ip_address_id, ip_address.v4address AS ip_address_v4address
FROM ip_address JOIN network ON ip_address.v4address << network.v4representation
```

New in version 0.9.2: - Added the `Operators.op.is_comparison` flag to assist in the creation of `relationship()` constructs using custom operators.

Overlapping Foreign Keys

A rare scenario can arise when composite foreign keys are used, such that a single column may be the subject of more than one column referred to via foreign key constraint.

Consider an (admittedly complex) mapping such as the `Magazine` object, referred to both by the `Writer` object and the `Article` object using a composite primary key scheme that includes `magazine_id` for both; then to make `Article` refer to `Writer` as well, `Article.magazine_id` is involved in two separate relationships; `Article.magazine` and `Article.writer`:

```
class Magazine(Base):
    __tablename__ = 'magazine'

    id = Column(Integer, primary_key=True)


class Article(Base):
    __tablename__ = 'article'

    article_id = Column(Integer)
    magazine_id = Column(ForeignKey('magazine.id'))
    writer_id = Column()

    magazine = relationship("Magazine")
    writer = relationship("Writer")

    __table_args__ = (
        PrimaryKeyConstraint('article_id', 'magazine_id'),
        ForeignKeyConstraint(
            ['writer_id', 'magazine_id'],
            ['writer.id', 'writer.magazine_id']
        ),
    )


class Writer(Base):
    __tablename__ = 'writer'

    id = Column(Integer, primary_key=True)
    magazine_id = Column(ForeignKey('magazine.id'), primary_key=True)
    magazine = relationship("Magazine")
```

When the above mapping is configured, we will see this warning emitted:

```
SAWarning: relationship 'Article.writer' will copy column
writer.magazine_id to column article.magazine_id,
which conflicts with relationship(s): 'Article.magazine'
(copies magazine.id to article.magazine_id). Consider applying
viewonly=True to read-only relationships, or provide a primaryjoin
condition marking writable columns with the foreign() annotation.
```

What this refers to originates from the fact that `Article.magazine_id` is the subject of two different foreign key constraints; it refers to `Magazine.id` directly as a source column, but also refers to `Writer.magazine_id` as a source column in the context of the composite key to `Writer`. If we associate an `Article` with a particular `Magazine`, but then associate the `Article` with a `Writer` that's associated with a *different* `Magazine`, the ORM will overwrite `Article.magazine_id` non-deterministically, silently changing which magazine we refer towards; it may also attempt to place NULL into this columnn if we de-associate a `Writer` from an `Article`. The warning lets us know this is the case.

To solve this, we need to break out the behavior of `Article` to include all three of the following features:

1. `Article` first and foremost writes to `Article.magazine_id` based on data persisted in the `Article.magazine` relationship only, that is a value copied from `Magazine.id`.
2. `Article` can write to `Article.writer_id` on behalf of data persisted in the `Article.writer` relationship, but only the `Writer.id` column; the `Writer.magazine_id` column should not be written into `Article.magazine_id` as it ultimately is sourced from `Magazine.id`.
3. `Article` takes `Article.magazine_id` into account when loading `Article.writer`, even though it *doesn't* write to it on behalf of this relationship.

To get just #1 and #2, we could specify only `Article.writer_id` as the "foreign keys" for `Article.writer`:

```
class Article(Base):
    # ...

    writer = relationship("Writer", foreign_keys='Article.writer_id')
```

However, this has the effect of `Article.writer` not taking `Article.magazine_id` into account when querying against `Writer`:

```
SELECT article.article_id AS article_article_id,
    article.magazine_id AS article_magazine_id,
    article.writer_id AS article_writer_id
FROM article
JOIN writer ON writer.id = article.writer_id
```

Therefore, to get at all of #1, #2, and #3, we express the join condition as well as which columns to be written by combining `primaryjoin` fully, along with either the `foreign_keys` argument, or more succinctly by annotating with `foreign()`:

```
class Article(Base):
    # ...

    writer = relationship(
        "Writer",
        primaryjoin="and_(Writer.id == foreign(Article.writer_id), "
                    "Writer.magazine_id == Article.magazine_id)")
```

Changed in version 1.0.0: the ORM will attempt to warn when a column is used as the synchronization target from more than one relationship simultaneously.

Non-relational Comparisons / Materialized Path

> **Warning:** this section details an experimental feature.

Using custom expressions means we can produce unorthodox join conditions that don't obey the usual primary/foreign key model. One such example is the materialized path pattern, where we compare strings for overlapping path tokens in order to produce a tree structure.

Through careful use of `foreign()` and `remote()`, we can build a relationship that effectively produces a rudimentary materialized path system. Essentially, when `foreign()` and `remote()` are on the *same* side of the comparison expression, the relationship is considered to be "one to many"; when they are on *different* sides, the relationship is considered to be "many to one". For the comparison we'll use here, we'll be dealing with collections so we keep things configured as "one to many":

```
class Element(Base):
    __tablename__ = 'element'

    path = Column(String, primary_key=True)

    descendants = relationship('Element',
                               primaryjoin=
                                    remote(foreign(path)).like(
                                            path.concat('/%')),
                               viewonly=True,
                               order_by=path)
```

Above, if given an `Element` object with a path attribute of `"/foo/bar2"`, we seek for a load of `Element.descendants` to look like:

```
SELECT element.path AS element_path
FROM element
WHERE element.path LIKE ('/foo/bar2' || '/%') ORDER BY element.path
```

New in version 0.9.5: Support has been added to allow a single-column comparison to itself within a primaryjoin condition, as well as for primaryjoin conditions that use `ColumnOperators.like()` as the comparison operator.

Self-Referential Many-to-Many Relationship

Many to many relationships can be customized by one or both of `primaryjoin` and `secondaryjoin` - the latter is significant for a relationship that specifies a many-to-many reference using the `secondary` argument. A common situation which involves the usage of `primaryjoin` and `secondaryjoin` is when establishing a many-to-many relationship from a class to itself, as shown below:

```
from sqlalchemy import Integer, ForeignKey, String, Column, Table
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

node_to_node = Table("node_to_node", Base.metadata,
    Column("left_node_id", Integer, ForeignKey("node.id"), primary_key=True),
    Column("right_node_id", Integer, ForeignKey("node.id"), primary_key=True)
)

class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    label = Column(String)
```

```
    right_nodes = relationship("Node",
                        secondary=node_to_node,
                        primaryjoin=id==node_to_node.c.left_node_id,
                        secondaryjoin=id==node_to_node.c.right_node_id,
                        backref="left_nodes"
    )
```

Where above, SQLAlchemy can't know automatically which columns should connect to which for the `right_nodes` and `left_nodes` relationships. The `primaryjoin` and `secondaryjoin` arguments establish how we'd like to join to the association table. In the Declarative form above, as we are declaring these conditions within the Python block that corresponds to the `Node` class, the `id` variable is available directly as the `Column` object we wish to join with.

Alternatively, we can define the `primaryjoin` and `secondaryjoin` arguments using strings, which is suitable in the case that our configuration does not have either the `Node.id` column object available yet or the `node_to_node` table perhaps isn't yet available. When referring to a plain `Table` object in a declarative string, we use the string name of the table as it is present in the `MetaData`:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    label = Column(String)
    right_nodes = relationship("Node",
                        secondary="node_to_node",
                        primaryjoin="Node.id==node_to_node.c.left_node_id",
                        secondaryjoin="Node.id==node_to_node.c.right_node_id",
                        backref="left_nodes"
    )
```

A classical mapping situation here is similar, where `node_to_node` can be joined to `node.c.id`:

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

metadata = MetaData()

node_to_node = Table("node_to_node", metadata,
    Column("left_node_id", Integer, ForeignKey("node.id"), primary_key=True),
    Column("right_node_id", Integer, ForeignKey("node.id"), primary_key=True)
)

node = Table("node", metadata,
    Column('id', Integer, primary_key=True),
    Column('label', String)
)
class Node(object):
    pass

mapper(Node, node, properties={
    'right_nodes':relationship(Node,
                        secondary=node_to_node,
                        primaryjoin=node.c.id==node_to_node.c.left_node_id,
                        secondaryjoin=node.c.id==node_to_node.c.right_node_id,
                        backref="left_nodes"
                    )})
```

Note that in both examples, the `backref` keyword specifies a `left_nodes` backref - when `relationship()` creates the second relationship in the reverse direction, it's smart enough to reverse the `primaryjoin` and `secondaryjoin` arguments.

Composite "Secondary" Joins

Note: This section features some new and experimental features of SQLAlchemy.

Sometimes, when one seeks to build a `relationship()` between two tables there is a need for more than just two or three tables to be involved in order to join them. This is an area of `relationship()` where one seeks to push the boundaries of what's possible, and often the ultimate solution to many of these exotic use cases needs to be hammered out on the SQLAlchemy mailing list.

In more recent versions of SQLAlchemy, the `secondary` parameter can be used in some of these cases in order to provide a composite target consisting of multiple tables. Below is an example of such a join condition (requires version 0.9.2 at least to function as is):

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)
    b_id = Column(ForeignKey('b.id'))

    d = relationship("D",
                secondary="join(B, D, B.d_id == D.id)."
                            "join(C, C.d_id == D.id)",
                primaryjoin="and_(A.b_id == B.id, A.id == C.a_id)",
                secondaryjoin="D.id == B.d_id",
                uselist=False
                )

class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    d_id = Column(ForeignKey('d.id'))

class C(Base):
    __tablename__ = 'c'

    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))
    d_id = Column(ForeignKey('d.id'))

class D(Base):
    __tablename__ = 'd'

    id = Column(Integer, primary_key=True)
```

In the above example, we provide all three of `secondary`, `primaryjoin`, and `secondaryjoin`, in the declarative style referring to the named tables `a`, `b`, `c`, `d` directly. A query from `A` to `D` looks like:

```
sess.query(A).join(A.d).all()

{opensql}SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a JOIN (
    b AS b_1 JOIN d AS d_1 ON b_1.d_id = d_1.id
        JOIN c AS c_1 ON c_1.d_id = d_1.id)
    ON a.b_id = b_1.id AND a.id = c_1.a_id JOIN d ON d.id = b_1.d_id
```

In the above example, we take advantage of being able to stuff multiple tables into a "secondary" container, so that we can join across many tables while still keeping things "simple" for `relationship()`, in that there's just "one" table on both the "left" and the "right" side; the complexity is kept within the middle.

New in version 0.9.2: Support is improved for allowing a `join()` construct to be used directly as the target of the `secondary` argument, including support for joins, eager joins and lazy loading, as well as support within declarative to specify complex conditions such as joins involving class names as targets.

Relationship to Non Primary Mapper

In the previous section, we illustrated a technique where we used `secondary` in order to place additional tables within a join condition. There is one complex join case where even this technique is not sufficient; when we seek to join from `A` to `B`, making use of any number of `C`, `D`, etc. in between, however there are also join conditions between `A` and `B` *directly*. In this case, the join from `A` to `B` may be difficult to express with just a complex `primaryjoin` condition, as the intermediary tables may need special handling, and it is also not expressable with a `secondary` object, since the `A->secondary->B` pattern does not support any references between `A` and `B` directly. When this **extremely advanced** case arises, we can resort to creating a second mapping as a target for the relationship. This is where we use `mapper()` in order to make a mapping to a class that includes all the additional tables we need for this join. In order to produce this mapper as an "alternative" mapping for our class, we use the `non_primary` flag.

Below illustrates a `relationship()` with a simple join from `A` to `B`, however the primaryjoin condition is augmented with two additional entities `C` and `D`, which also must have rows that line up with the rows in both `A` and `B` simultaneously:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)
    b_id = Column(ForeignKey('b.id'))

class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)

class C(Base):
    __tablename__ = 'c'

    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))

class D(Base):
    __tablename__ = 'd'

    id = Column(Integer, primary_key=True)
    c_id = Column(ForeignKey('c.id'))
    b_id = Column(ForeignKey('b.id'))

# 1. set up the join() as a variable, so we can refer
# to it in the mapping multiple times.
j = join(B, D, D.b_id == B.id).join(C, C.id == D.c_id)

# 2. Create a new mapper() to B, with non_primary=True.
# Columns in the join with the same name must be
# disambiguated within the mapping, using named properties.
B_viacd = mapper(B, j, non_primary=True, properties={
    "b_id": [j.c.b_id, j.c.d_b_id],
    "d_id": j.c.d_id
    })

A.b = relationship(B_viacd, primaryjoin=A.b_id == B_viacd.c.b_id)
```

In the above case, our non-primary mapper for `B` will emit for additional columns when we query; these can be ignored:

```
sess.query(A).join(A.b).all()

{opensql}SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a JOIN (b JOIN d ON d.b_id = b.id JOIN c ON c.id = d.c_id) ON a.b_id = b.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` 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(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)

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

2.3.5 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.

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

A key feature to enable management of a large collection is the so-called "dynamic" relationship. This is an optional form 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:

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

    posts = relationship(Post, lazy="dynamic")

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 the `backref()` function in conjunction with `lazy='dynamic'`:

```
class Post(Base):
    __table__ = posts_table

    user = relationship(User,
                backref=backref('posts', lazy='dynamic')
            )
```

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

Note: The `dynamic_loader()` function is essentially the same as `relationship()` with the `lazy='dynamic'` argument specified.

Warning: The "dynamic" loader applies to **collections only**. It is not valid to use "dynamic" loaders with many-to-one, one-to-one, or uselist=False relationships. Newer versions of SQLAlchemy emit warnings or exceptions in these cases.

Setting Noload, RaiseLoad

A "noload" relationship never loads from the database, even when accessed. It is configured using `lazy='noload'`:

```
class MyClass(Base):
    __tablename__ = 'some_table'

    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. The noload strategy is also available on a query option basis using the `orm.noload()` loader option.

Alternatively, a "raise"-loaded relationship will raise an `InvalidRequestError` where the attribute would normally emit a lazy load:

```
class MyClass(Base):
    __tablename__ = 'some_table'

    children = relationship(MyOtherClass, lazy='raise')
```

Above, attribute access on the `children` collection will raise an exception if it was not previously eagerloaded. This includes read access but for collections will also affect write access, as collections can't be mutated without first loading them. The rationale for this is to ensure that an application is not emitting any unexpected lazy loads within a certain context. Rather than having to read through SQL

logs to determine that all necessary attributes were eager loaded, the "raise" strategy will cause unloaded attributes to raise immediately if accessed. The raise strategy is also available on a query option basis using the `orm.raiseload()` loader option.

New in version 1.1: added the "raise" loader strategy.

See also:

prevent_lazy_with_raiseload

Using Passive Deletes

Use `passive_deletes` 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:

```
class MyClass(Base):
    __tablename__ = 'mytable'
    id = Column(Integer, primary_key=True)
    children = relationship("MyOtherClass",
                    cascade="all, delete-orphan",
                    passive_deletes=True)

class MyOtherClass(Base):
    __tablename__ = 'myothertable'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer,
                ForeignKey('mytable.id', ondelete='CASCADE')
                    )
```

Note: To use "ON DELETE CASCADE", the underlying database engine must support foreign keys.

- When using MySQL, an appropriate storage engine must be selected. See *CREATE TABLE arguments including Storage Engines* for details.
- When using SQLite, foreign key support must be enabled explicitly. See *Foreign Key Support* for details.

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.

See also:

`orm.mapper.passive_deletes` - similar feature on `mapper()`

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`:

```
class Parent(Base):
    __tablename__ = 'parent'
    parent_id = Column(Integer, primary_key=True)

    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()`:

```
class Parent(Base):
    __tablename__ = 'parent'
    parent_id = Column(Integer, primary_key=True)

    # use a set
    children = relationship(Child, collection_class=set)

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

Dictionary Collections

A little extra detail is needed when using a dictionary as a collection. This because objects are always loaded from the database as lists, and a key-generation strategy must be available to populate the dictionary correctly. The `attribute_mapped_collection()` function is by far the most common way to achieve a simple dictionary collection. It produces a dictionary class that will apply a particular attribute of the mapped class as a key. Below we map an `Item` class containing a dictionary of `Note` items keyed to the `Note.keyword` attribute:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.orm.collections import attribute_mapped_collection
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Item(Base):
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
    notes = relationship("Note",
                collection_class=attribute_mapped_collection('keyword'),
                cascade="all, delete-orphan")

class Note(Base):
    __tablename__ = 'note'
    id = Column(Integer, primary_key=True)
    item_id = Column(Integer, ForeignKey('item.id'), nullable=False)
    keyword = Column(String)
    text = Column(String)

    def __init__(self, keyword, text):
        self.keyword = keyword
        self.text = text
```

`Item.notes` is then a dictionary:

```
>>> item = Item()
>>> item.notes['a'] = Note('a', 'atext')
>>> item.notes.items()
{'a': <__main__.Note object at 0x2eaaf0>}
```

`attribute_mapped_collection()` will ensure that the `.keyword` attribute of each `Note` complies with the key in the dictionary. Such as, when assigning to `Item.notes`, the dictionary key we supply must

match that of the actual `Note` object:

```
item = Item()
item.notes = {
            'a': Note('a', 'atext'),
            'b': Note('b', 'btext')
        }
```

The attribute which `attribute_mapped_collection()` uses as a key does not need to be mapped at all! Using a regular Python `@property` allows virtually any detail or combination of details about the object to be used as the key, as below when we establish it as a tuple of `Note.keyword` and the first ten letters of the `Note.text` field:

```
class Item(Base):
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
    notes = relationship("Note",
                collection_class=attribute_mapped_collection('note_key'),
                backref="item",
                cascade="all, delete-orphan")

class Note(Base):
    __tablename__ = 'note'
    id = Column(Integer, primary_key=True)
    item_id = Column(Integer, ForeignKey('item.id'), nullable=False)
    keyword = Column(String)
    text = Column(String)

    @property
    def note_key(self):
        return (self.keyword, self.text[0:10])

    def __init__(self, keyword, text):
        self.keyword = keyword
        self.text = text
```

Above we added a `Note.item` backref. Assigning to this reverse relationship, the `Note` is added to the `Item.notes` dictionary and the key is generated for us automatically:

```
>>> item = Item()
>>> n1 = Note("a", "atext")
>>> n1.item = item
>>> item.notes
{('a', 'atext'): <__main__.Note object at 0x2eaaf0>}
```

Other built-in dictionary types include `column_mapped_collection()`, which is almost like `attribute_mapped_collection()` except given the `Column` object directly:

```
from sqlalchemy.orm.collections import column_mapped_collection

class Item(Base):
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
    notes = relationship("Note",
                collection_class=column_mapped_collection(Note.__table__.c.keyword),
                cascade="all, delete-orphan")
```

as well as `mapped_collection()` which is passed any callable function. Note that it's usually easier to use `attribute_mapped_collection()` along with a `@property` as mentioned earlier:

```
from sqlalchemy.orm.collections import mapped_collection

class Item(Base):
```

```
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
    notes = relationship("Note",
                collection_class=mapped_collection(lambda note: note.text[0:10]),
                cascade="all, delete-orphan")
```

Dictionary mappings are often combined with the "Association Proxy" extension to produce streamlined dictionary views. See proxying_dictionaries and composite_association_proxy for examples.

`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, where `attr_name` is the string name of the attribute.

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.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.

Custom Collection Implementations

You can use your own types for collections as well. In simple cases, inherting from `list` or `set`, adding custom behavior, is all that's needed. In other cases, special decorators are needed to tell SQLAlchemy more detail about how the collection operates.

Do I need a custom collection implementation?

In most cases not at all! The most common use cases for a "custom" collection is one that validates or marshals incoming values into a new form, such as a string that becomes a class instance, or one which goes a step beyond and represents the data internally in some fashion, presenting a "view" of that data on the outside of a different form.

For the first use case, the `orm.validates()` decorator is by far the simplest way to intercept incoming values in all cases for the purposes of validation and simple marshaling. See simple_validators for an example of this.

For the second use case, the associationproxy_toplevel extension is a well-tested, widely used system that provides a read/write "view" of a collection in terms of some attribute present on the target object. As the target attribute can be a `@property` that returns virtually anything, a wide array of "alternative" views of a collection can be constructed with just a few functions. This approach leaves the underlying mapped collection unaffected and avoids the need to carefully tailor collection behavior on a method-by-method basis.

Customized collections are useful when the collection needs to have special behaviors upon access or mutation operations that can't otherwise be modeled externally to the collection. They can of course be combined with the above two approaches.

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 when you otherwise

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.

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, linker, converter, internally_instrumented) 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): ...
```

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, its 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 link(*fn*)
deprecated; synonym for `collection.linker()`.

static linker(*fn*)
Tag the method as a "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.

Deprecated since version 1.0.0: - the `collection.linker()` handler is superseded by the `AttributeEvents.init_collection()` and `AttributeEvents.dispose_collection()` handlers.

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): ...
```

Custom Dictionary-Based Collections

The `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.

Note: Due to a bug in MappedCollection prior to version 0.7.6, this workaround usually needs to be called before a custom subclass of `MappedCollection` which uses `collection.internally_instrumented()` can be used:

```
from sqlalchemy.orm.collections import _instrument_class, MappedCollection
_instrument_class(MappedCollection)
```

This will ensure that the `MappedCollection` has been properly initialized with custom `__setitem__()` and `__delitem__()` methods before used in a custom subclass.

`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.

`clear()` → None. Remove all items from D.

`pop(`*k*[, *d*]`)` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`popitem()` → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

`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.

`setdefault(`*k*[, *d*]`)` → D.get(k,d), also set D[k]=d if k not in D

`update(`[*E*], ***F*`)` → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

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.

Collection Internals

Various internal methods.

`sqlalchemy.orm.collections.bulk_replace(`*values*, *existing_adapter*, *new_adapter*, *initiator=None*`)`

Load a new collection, firing events based on prior like membership.

Appends instances in `values` onto the `new_adapter`. Events will be fired for any instance not present in the `existing_adapter`. Any instances in `existing_adapter` not present in `values` will have remove events fired upon them.

Parameters

- **values** – An iterable of collection member instances
- **existing_adapter** – A `CollectionAdapter` of instances to be replaced
- **new_adapter** – An empty `CollectionAdapter` to load with `values`

`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, linker, converter, internally_instrumented) 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): ...
```

`sqlalchemy.orm.collections.collection_adapter = operator.attrgetter('_sa_adapter')`

Fetch the `CollectionAdapter` for a collection.

`class sqlalchemy.orm.collections.CollectionAdapter(`*attr*, *owner_state*, *data*`)`

Bridges between the ORM and arbitrary Python collections.

Proxies base-level collection operations (append, remove, iterate) to the underlying Python collection, and emits add/remove events for entities entering or leaving the collection.

The ORM uses `CollectionAdapter` exclusively for interaction with entity collections.

class sqlalchemy.orm.collections.InstrumentedDict
An instrumented version of the built-in dict.

class sqlalchemy.orm.collections.InstrumentedList
An instrumented version of the built-in list.

class sqlalchemy.orm.collections.InstrumentedSet
An instrumented version of the built-in set.

sqlalchemy.orm.collections.prepare_instrumentation(*factory*)
Prepare a callable for future use as a collection class factory.

Given a collection class factory (either a type or no-arg callable), return another factory that will produce compatible instances when called.

This function is responsible for converting collection_class=list into the run-time behavior of collection_class=InstrumentedList.

2.3.6 Special Relationship Persistence Patterns

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 usage of a supplementary UPDATE statement, we use the `post_update` option of `relationship()`. This specifies that the linkage between the two rows should be created using an UPDATE statement after both rows have been INSERTED; it also causes the rows to be de-associated with each other via UPDATE before a DELETE is emitted. The flag should be placed on just *one* of the relationships, preferably the many-to-one side. Below we illustrate a complete example, including two `ForeignKey` constructs:

```
from sqlalchemy import Integer, ForeignKey, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Entry(Base):
    __tablename__ = 'entry'
    entry_id = Column(Integer, primary_key=True)
    widget_id = Column(Integer, ForeignKey('widget.widget_id'))
    name = Column(String(50))

class Widget(Base):
    __tablename__ = 'widget'

    widget_id = Column(Integer, primary_key=True)
    favorite_entry_id = Column(Integer,
                               ForeignKey('entry.entry_id',
                               name="fk_favorite_entry"))
    name = Column(String(50))

    entries = relationship(Entry, primaryjoin=
                                    widget_id==Entry.widget_id)
    favorite_entry = relationship(Entry,
                                  primaryjoin=
                                      favorite_entry_id==Entry.entry_id,
                                  post_update=True)
```

When a structure against the above configuration 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):

```
>>> w1 = Widget(name='somewidget')
>>> e1 = Entry(name='someentry')
>>> w1.favorite_entry = e1
>>> w1.entries = [e1]
>>> session.add_all([w1, e1])
{sql}>>> session.commit()
BEGIN (implicit)
INSERT INTO widget (favorite_entry_id, name) VALUES (?, ?)
(None, 'somewidget')
INSERT INTO entry (widget_id, name) VALUES (?, ?)
(1, 'someentry')
UPDATE widget SET favorite_entry_id=? WHERE widget.widget_id = ?
(1, 1)
COMMIT
```

An additional configuration we can specify is to supply a more comprehensive foreign key constraint on `Widget`, such that it's guaranteed that `favorite_entry_id` refers to an `Entry` that also refers to this `Widget`. We can use a composite foreign key, as illustrated below:

```
from sqlalchemy import Integer, ForeignKey, String, \
        Column, UniqueConstraint, ForeignKeyConstraint
from sqlalchemy.ext.declarative import declarative_base
```

```
from sqlalchemy.orm import relationship

Base = declarative_base()

class Entry(Base):
    __tablename__ = 'entry'
    entry_id = Column(Integer, primary_key=True)
    widget_id = Column(Integer, ForeignKey('widget.widget_id'))
    name = Column(String(50))
    __table_args__ = (
        UniqueConstraint("entry_id", "widget_id"),
    )

class Widget(Base):
    __tablename__ = 'widget'

    widget_id = Column(Integer, autoincrement='ignore_fk', primary_key=True)
    favorite_entry_id = Column(Integer)

    name = Column(String(50))

    __table_args__ = (
        ForeignKeyConstraint(
            ["widget_id", "favorite_entry_id"],
            ["entry.widget_id", "entry.entry_id"],
            name="fk_favorite_entry"
        ),
    )

    entries = relationship(Entry, primaryjoin=
                                    widget_id==Entry.widget_id,
                                    foreign_keys=Entry.widget_id)
    favorite_entry = relationship(Entry,
                                  primaryjoin=
                                      favorite_entry_id==Entry.entry_id,
                                  foreign_keys=favorite_entry_id,
                                  post_update=True)
```

The above mapping features a composite `ForeignKeyConstraint` bridging the `widget_id` and `favorite_entry_id` columns. To ensure that `Widget.widget_id` remains an "autoincrementing" column we specify `autoincrement` to the value `"ignore_fk"` on `Column`, and additionally on each `relationship()` we must limit those columns considered as part of the foreign key for the purposes of joining and cross-population.

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, the best strategy is 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 unless the constraints are marked as "deferrable", that is, not enforced until the transaction completes.

It is **highly recommended** that an application which seeks to employ natural primary keys with mutable values to use the `ON UPDATE CASCADE` capabilities of the database. An example mapping which illustrates this is:

```
class User(Base):
    __tablename__ = 'user'
    __table_args__ = {'mysql_engine': 'InnoDB'}

    username = Column(String(50), primary_key=True)
```

```
    fullname = Column(String(100))

    addresses = relationship("Address")


class Address(Base):
    __tablename__ = 'address'
    __table_args__ = {'mysql_engine': 'InnoDB'}

    email = Column(String(50), primary_key=True)
    username = Column(String(50),
                ForeignKey('user.username', onupdate="cascade")
            )
```

Above, we illustrate `onupdate="cascade"` on the `ForeignKey` object, and we also illustrate the `mysql_engine='InnoDB'` setting which, on a MySQL backend, ensures that the `InnoDB` engine supporting referential integrity is used. When using SQLite, referential integrity should be enabled, using the configuration described at *Foreign Key Support*.

See also:

passive_deletes - supporting ON DELETE CASCADE with relationships

`orm.mapper.passive_updates` - similar feature on `mapper()`

Simulating limited ON UPDATE CASCADE without foreign key support

In those cases when a database that does not support referential integrity is used, and natural primary keys with mutable values are in play, SQLAlchemy offers a feature in order to allow propagation of primary key values to already-referenced foreign keys to a **limited** extent, by emitting an UPDATE statement against foreign key columns that immediately reference a primary key column whose value has changed. The primary platforms without referential integrity features are MySQL when the `MyISAM` storage engine is used, and SQLite when the `PRAGMA foreign_keys=ON` pragma is not used. The Oracle database also has no support for `ON UPDATE CASCADE`, but because it still enforces referential integrity, needs constraints to be marked as deferrable so that SQLAlchemy can emit UPDATE statements.

The feature is enabled by setting the `passive_updates` flag to `False`, most preferably on a one-to-many or many-to-many `relationship()`. When "updates" are no longer "passive" this indicates that SQLAlchemy will issue UPDATE statements individually for objects referenced in the collection referred to by the parent object with a changing primary key value. This also implies that collections will be fully loaded into memory if not already locally present.

Our previous mapping using `passive_updates=False` looks like:

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

    username = Column(String(50), primary_key=True)
    fullname = Column(String(100))

    # passive_updates=False *only* needed if the database
    # does not implement ON UPDATE CASCADE
    addresses = relationship("Address", passive_updates=False)

class Address(Base):
    __tablename__ = 'address'

    email = Column(String(50), primary_key=True)
    username = Column(String(50), ForeignKey('user.username'))
```

Key limitations of `passive_updates=False` include:

- it performs much more poorly than direct database ON UPDATE CASCADE, because it needs to fully pre-load affected collections using SELECT and also must emit UPDATE statements against those values, which it will attempt to run in "batches" but still runs on a per-row basis at the DBAPI level.
- the feature cannot "cascade" more than one level. That is, if mapping X has a foreign key which refers to the primary key of mapping Y, but then mapping Y's primary key is itself a foreign key to mapping Z, `passive_updates=False` cannot cascade a change in primary key value from `Z` to `X`.
- Configuring `passive_updates=False` only on the many-to-one side of a relationship will not have a full 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.

As virtually all databases other than Oracle now support `ON UPDATE CASCADE`, it is highly recommended that traditional `ON UPDATE CASCADE` support be used in the case that natural and mutable primary key values are in use. Functional constructs for ORM configuration.

See the SQLAlchemy object relational tutorial and mapper configuration documentation for an overview of how this module is used.

2.3.7 Relationships API

`sqlalchemy.orm.relationship`(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign_keys=None*, *uselist=None*, *order_by=False*, *backref=None*, *back_populates=None*, *post_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy='select'*, *collection_class=None*, *passive_deletes=False*, *passive_updates=True*, *remote_side=None*, *enable_typechecks=True*, *join_depth=None*, *comparator_factory=None*, *single_parent=False*, *innerjoin=False*, *distinct_target_key=None*, *doc=None*, *active_history=False*, *cascade_backrefs=True*, *load_on_pending=False*, *bake_queries=True*, *_local_remote_pairs=None*, *query_class=None*, *info=None*)

Provide a relationship between two mapped classes.

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

A typical `relationship()`, used in a classical mapping:

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

Some arguments accepted by `relationship()` optionally accept a callable function, which when called produces the desired value. The callable is invoked by the parent `Mapper` at "mapper initialization" time, which happens only when mappers are first used, and is assumed to be after all mappings have been constructed. This can be used to resolve order-of-declaration and other dependency issues, such as if `Child` is declared below `Parent` in the same file:

```
mapper(Parent, properties={
    "children":relationship(lambda: Child,
                        order_by=lambda: Child.id)
})
```

When using the declarative_toplevel extension, the Declarative initializer allows string arguments to be passed to `relationship()`. These string arguments are converted into callables that evaluate the string as Python code, using the Declarative class-registry as a namespace. This allows the

lookup of related classes to be automatic via their string name, and removes the need to import related classes at all into the local module space:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

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

See also:

relationship_config_toplevel - Full introductory and reference documentation for `relationship()`.

orm_tutorial_relationship - ORM tutorial introduction.

Parameters

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

  `argument` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  declarative_configuring_relationships - further detail on relationship configuration when using Declarative.

- **`secondary`** – for a many-to-many relationship, specifies the intermediary table, and is typically an instance of `Table`. In less common circumstances, the argument may also be specified as an `Alias` construct, or even a `Join` construct.

  `secondary` may also be passed as a callable function which is evaluated at mapper initialization time. When using Declarative, it may also be a string argument noting the name of a `Table` that is present in the `MetaData` collection associated with the parent-mapped `Table`.

  The `secondary` keyword argument is typically applied in the case where the intermediary `Table` is not otherwise expressed in any direct class mapping. If the "secondary" table is also explicitly mapped elsewhere (e.g. as in association_pattern), one should consider applying the `viewonly` flag so that this `relationship()` is not used for persistence operations which may conflict with those of the association object pattern.

  See also:

  relationships_many_to_many - Reference example of "many to many".

  orm_tutorial_many_to_many - ORM tutorial introduction to many-to-many relationships.

  self_referential_many_to_many - Specifics on using many-to-many in a self-referential case.

  declarative_many_to_many - Additional options when using Declarative.

  association_pattern - an alternative to `secondary` when composing association table relationships, allowing additional attributes to be specified on the association table.

  composite_secondary_join - a lesser-used pattern which in some cases can enable complex `relationship()` SQL conditions to be used.

New in version 0.9.2: `secondary` works more effectively when referring to a `Join` instance.

- `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.
- `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.

  See also:

  relationships_backref - Introductory documentation and examples.

  `back_populates` - alternative form of backref specification.

  `backref()` - allows control over `relationship()` configuration when using `backref`.
- `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.

  See also:

  relationships_backref - Introductory documentation and examples.

  `backref` - alternative form of backref specification.
- `bake_queries=True` – Use the `BakedQuery` cache to cache the construction of SQL used in lazy loads. True by default. Set to False if the join condition of the relationship has unusual features that might not respond well to statement caching.

  Changed in version 1.2: "Baked" loading is the default implementation for the "select", a.k.a. "lazy" loading strategy for relationships.

  New in version 1.0.0.

  See also:

  baked_toplevel
- `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 - this default cascade is `"save-update, merge"`.

  The available cascades are `save-update`, `merge`, `expunge`, `delete`, `delete-orphan`, and `refresh-expire`. An additional option, `all` indicates shorthand for `"save-update, merge, refresh-expire, expunge, delete"`, and is often used as in `"all, delete-orphan"` to indicate that related objects should follow along with the parent object in all cases, and be deleted when de-associated.

  See also:

  unitofwork_cascades - Full detail on each of the available cascade options.

  tutorial_delete_cascade - Tutorial example describing a delete cascade.

- **cascade_backrefs=True** – a boolean value indicating if the `save-update` cascade should operate along an assignment event intercepted by a backref. When set to `False`, the attribute managed by this relationship will not cascade an incoming transient object into the session of a persistent parent, if the event is received via backref.

  See also:

  backref_cascade - Full discussion and examples on how the `cascade_backrefs` option is used.

- **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.

  See also:

  custom_collections - Introductory documentation and examples.

- **comparator_factory** – a class which extends `RelationshipProperty.Comparator` which provides custom SQL clause generation for comparison operations.

  See also:

  `PropComparator` - some detail on redefining comparators at this level.

  custom_comparators - Brief intro to this feature.

- **distinct_target_key=None** – Indicate if a "subquery" eager load should apply the DISTINCT keyword to the innermost SELECT statement. When left as `None`, the DISTINCT keyword will be applied in those cases when the target columns do not comprise the full primary key of the target table. When set to `True`, the DISTINCT keyword is applied to the innermost SELECT unconditionally.

  It may be desirable to set this flag to False when the DISTINCT is reducing performance of the innermost subquery beyond that of what duplicate innermost rows may be causing.

  New in version 0.8.3: - `distinct_target_key` allows the subquery eager loader to apply a DISTINCT modifier to the innermost SELECT.

  Changed in version 0.9.0: - `distinct_target_key` now defaults to `None`, so that the feature enables itself automatically for those cases where the innermost query targets a non-unique key.

  See also:

  loading_toplevel - includes an introduction to subquery eager loading.

- **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.

  Deprecated since version 0.7: Please see `AttributeEvents`.

- **foreign_keys** – a list of columns which are to be used as "foreign key" columns, or columns which refer to the value in a remote column, within the context of this `relationship()` object's `primaryjoin` condition. That is, if the `primaryjoin` condition of this `relationship()` is `a.id == b.a_id`, and the values in `b.a_id` are required to be present in `a.id`, then the "foreign key" column of this `relationship()` is `b.a_id`.

  In normal cases, the `foreign_keys` parameter is **not required.** `relationship()` will automatically determine which columns in the `primaryjoin` conditition are to be considered "foreign key" columns based

on those `Column` objects that specify `ForeignKey`, or are otherwise listed as referencing columns in a `ForeignKeyConstraint` construct. `foreign_keys` is only needed when:

1. There is more than one way to construct a join from the local table to the remote table, as there are multiple foreign key references present. Setting `foreign_keys` will limit the `relationship()` to consider just those columns specified here as "foreign".

   Changed in version 0.8: A multiple-foreign key join ambiguity can be resolved by setting the `foreign_keys` parameter alone, without the need to explicitly set `primaryjoin` as well.

2. The `Table` being mapped does not actually have `ForeignKey` or `ForeignKeyConstraint` constructs present, often because the table was reflected from a database that does not support foreign key reflection (MySQL MyISAM).

3. The `primaryjoin` argument is used to construct a non-standard join condition, which makes use of columns or expressions that do not normally refer to their "parent" column, such as a join condition expressed by a complex comparison using a SQL function.

The `relationship()` construct will raise informative error messages that suggest the use of the `foreign_keys` parameter when presented with an ambiguous condition. In typical cases, if `relationship()` doesn't raise any exceptions, the `foreign_keys` parameter is usually not needed.

`foreign_keys` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

relationship_foreign_keys

relationship_custom_foreign

`foreign()` - allows direct annotation of the "foreign" columns within a `primaryjoin` condition.

New in version 0.8: The `foreign()` annotation can also be applied directly to the `primaryjoin` expression, which is an alternate, more specific system of describing which columns in a particular `primaryjoin` should be considered "foreign".

- **`info`** – Optional data dictionary which will be populated into the `MapperProperty.info` attribute of this object.

  New in version 0.8.

- **`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 generally 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.

  The option supports the same "nested" and "unnested" options as that of `joinedload.innerjoin`. See that flag for details on nested / unnested behaviors.

  See also:

  `joinedload.innerjoin` - the option as specified by loader option, including detail on nesting behavior.

  what_kind_of_loading - Discussion of some details of various loader options.

- `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.

  See also:

  self_referential_eager_loading - Introductory documentation and examples.

- `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.

  – `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" as the parents are loaded, using one additional SQL statement, which issues a JOIN to a subquery of the original statement, for each collection requested.

  – `selectin` - items should be loaded "eagerly" as the parents are loaded, using one or more additional SQL statements, which issues a JOIN to the immediate parent object, specifying primary key identifiers using an IN clause.

    New in version 1.2.

  – `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.

  – `raise` - lazy loading is disallowed; accessing the attribute, if its value were not already loaded via eager loading, will raise an `InvalidRequestError`. This strategy can be used when objects are to be detached from their attached `Session` after they are loaded.

    New in version 1.1.

  – `raise_on_sql` - lazy loading that emits SQL is disallowed; accessing the attribute, if its value were not already loaded via eager loading, will raise an `InvalidRequestError`, **if the lazy load needs to emit SQL**. If the lazy load can pull the related value from the identity map or determine that it should be None, the value is loaded. This strategy can be used when objects will remain associated with the attached `Session`, however additional SELECT statements should be blocked.

    New in version 1.1.

  – `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. See the section dynamic_relationship for more details.

  – True - a synonym for 'select'

  – False - a synonym for 'joined'

  – None - a synonym for 'noload'

See also:

Relationship Loading Techniques - Full documentation on relationship loader configuration.

dynamic_relationship - detail on the `dynamic` option.

collections_noload_raiseload - notes on "noload" and "raise"

- **`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.

  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 a flush proceeds. This flag is not not intended for general use.

  See also:

  `Session.enable_relationship_loading()` - this method establishes "load on pending" behavior for the whole object, and also allows loading on objects that remain transient or detached.

- **`order_by`** – indicates the ordering that should be applied when loading these items. `order_by` is expected to refer to one of the `Column` objects to which the target class is mapped, or the attribute itself bound to the target class which refers to the column.

  `order_by` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

- **`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 the parent object is deleted and 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. Additionally, the "nulling out" will still occur if the child object is de-associated with the parent.

  See also:

  passive_deletes - Introductory documentation and examples.

- **`passive_updates=True`** – Indicates the persistence behavior to take when a referenced primary key value changes in place, indicating that the referencing foreign key columns will also need their value changed.

  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. When False, the SQLAlchemy `relationship()` construct will attempt to emit its own UPDATE statements to modify related targets. However note that SQLAlchemy **cannot** emit an

UPDATE for more than one level of cascade. Also, setting this flag to False is not compatible in the case where the database is in fact enforcing referential integrity, unless those constraints are explicitly "deferred", if the target backend supports it.

It is highly advised that an application which is employing mutable primary keys keeps `passive_updates` set to True, and instead uses the referential integrity features of the database itself in order to handle the change efficiently and fully.

See also:

passive_updates - Introductory documentation and examples.

`mapper.passive_updates` - a similar flag which takes effect for joined-table inheritance mappings.

- `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.

  See also:

  post_update - Introductory documentation and examples.

- `primaryjoin` – a SQL expression 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).

  `primaryjoin` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  relationship_primaryjoin

- `remote_side` – used for self-referential relationships, indicates the column or list of columns that form the "remote side" of the relationship.

  `relationship.remote_side` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  Changed in version 0.8: The `remote()` annotation can also be applied directly to the `primaryjoin` expression, which is an alternate, more specific system of describing which columns in a particular `primaryjoin` should be considered "remote".

  See also:

  self_referential - in-depth explanation of how `remote_side` is used to configure self-referential relationships.

  `remote()` - an annotation function that accomplishes the same purpose as `remote_side`, typically when a custom `primaryjoin` condition is used.

- **query_class** – a Query subclass that will be used as the base of the "appender query" returned by a "dynamic" relationship, that is, a relationship that specifies lazy="dynamic" or was otherwise constructed using the orm.dynamic_loader() function.

  See also:

  dynamic_relationship - Introduction to "dynamic" relationship loaders.

- **secondaryjoin** – a SQL expression 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.

  secondaryjoin may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  relationship_primaryjoin

- **single_parent** – 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, except for relationship() constructs which are many-to-one or many-to-many and also specify the delete-orphan cascade option. The relationship() construct itself will raise an error instructing when this option is required.

  See also:

  unitofwork_cascades - includes detail on when the single_parent flag may be appropriate.

- **uselist** – 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() at mapper configuration time, 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.

  The uselist flag is also available on an existing relationship() construct as a read-only attribute, which can be used to determine if this relationship() deals with collections or scalar attributes:

  ```
  >>> User.addresses.property.uselist
  True
  ```

  See also:

  relationships_one_to_one - Introduction to the "one to one" relationship pattern, which is typically when the uselist flag is needed.

- **viewonly=False** – when set to True, the relationship is used only for loading objects, and not for any persistence operation. A relationship() which specifies viewonly can work with a wider range of SQL operations within the primaryjoin condition, including operations that feature the use of a variety of comparison operators as well as SQL functions such as cast(). The viewonly flag is also of general use when defining any kind of relationship() that doesn't represent the full set of related objects, to prevent modifications of the collection from resulting in persistence operations.

sqlalchemy.orm.**backref**(*name*, ***kwargs*)

Create a back reference with explicit keyword 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, e.g.:

```
'items':relationship(
    SomeItem, backref=backref('parent', lazy='subquery'))
```

See also:

relationships_backref

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

`sqlalchemy.orm.dynamic_loader(argument, **kw)`
Construct a dynamically-loading mapper property.

This is essentially the same as using the `lazy='dynamic'` argument with `relationship()`:

```
dynamic_loader(SomeClass)

# is the same as

relationship(SomeClass, lazy="dynamic")
```

See the section dynamic_relationship for more details on dynamic loading.

`sqlalchemy.orm.foreign(expr)`
Annotate a portion of a primaryjoin expression with a 'foreign' annotation.

See the section relationship_custom_foreign for a description of use.

New in version 0.8.

See also:

relationship_custom_foreign

`remote()`

`sqlalchemy.orm.remote(expr)`
Annotate a portion of a primaryjoin expression with a 'remote' annotation.

See the section relationship_custom_foreign for a description of use.

New in version 0.8.

See also:

relationship_custom_foreign

`foreign()`

2.4 Loading Objects

Notes and features regarding the general loading of mapped objects.

For an in-depth introduction to querying with the SQLAlchemy ORM, please see the ormtutorial_toplevel.

2.4.1 Loading Columns

This section presents additional options regarding the loading of columns.

Deferred Column Loading

This feature allows particular columns of a table be loaded only upon direct access, instead of when the entity is queried using `Query`. 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, using the `orm.deferred()` function to mark them as "deferred". In the example below, we define a mapping 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:

```
from sqlalchemy.orm import deferred
from sqlalchemy import Integer, String, Text, Binary, Column

class Book(Base):
    __tablename__ = 'book'

    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))
```

Classical mappings as always place the usage of `orm.deferred()` in the `properties` dictionary against the table-bound `Column`:

```
mapper(Book, book_table, properties={
    'photo':deferred(book_table.c.photo)
})
```

Deferred columns can be associated with a "group" name, so that they load together when any of them are first accessed. The example below defines a mapping with a `photos` deferred group. When one `.photo` is accessed, all three photos will be loaded in one SELECT statement. The `.excerpt` will be loaded separately when it is accessed:

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

    book_id = Column(Integer, primary_key=True)
    title = Column(String(200), nullable=False)
    summary = Column(String(2000))
    excerpt = deferred(Column(Text))
    photo1 = deferred(Column(Binary), group='photos')
    photo2 = deferred(Column(Binary), group='photos')
    photo3 = deferred(Column(Binary), group='photos')
```

You can defer or undefer columns at the `Query` level using options, including `orm.defer()` and `orm.undefer()`:

```
from sqlalchemy.orm import defer, undefer

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

`orm.deferred()` attributes which are marked with a "group" can be undeferred using `orm.undefer_group()`, sending in the group name:

```
from sqlalchemy.orm import undefer_group

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

Load Only Cols

An arbitrary set of columns can be selected as "load only" columns, which will be loaded while deferring all other columns on a given entity, using `orm.load_only()`:

```
from sqlalchemy.orm import load_only

session.query(Book).options(load_only("summary", "excerpt"))
```

New in version 0.9.0.

Deferred Loading with Multiple Entities

To specify column deferral options within a `Query` that loads multiple types of entity, the `Load` object can specify which parent entity to start with:

```
from sqlalchemy.orm import Load

query = session.query(Book, Author).join(Book.author)
query = query.options(
            Load(Book).load_only("summary", "excerpt"),
            Load(Author).defer("bio")
        )
```

To specify column deferral options along the path of various relationships, the options support chaining, where the loading style of each relationship is specified first, then is chained to the deferral options. Such as, to load `Book` instances, then joined-eager-load the `Author`, then apply deferral options to the `Author` entity:

```
from sqlalchemy.orm import joinedload

query = session.query(Book)
query = query.options(
            joinedload(Book.author).load_only("summary", "excerpt"),
        )
```

In the case where the loading style of parent relationships should be left unchanged, use `orm.defaultload()`:

```
from sqlalchemy.orm import defaultload

query = session.query(Book)
query = query.options(
            defaultload(Book.author).load_only("summary", "excerpt"),
        )
```

New in version 0.9.0: support for `Load` and other options which allow for better targeting of deferral options.

Column Deferral API

`sqlalchemy.orm.defer`(*key*, **addl_attrs*)

Indicate that the given column-oriented attribute should be deferred, e.g. not loaded until accessed.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

e.g.:

```
from sqlalchemy.orm import defer

session.query(MyClass).options(
                    defer("attribute_one"),
                    defer("attribute_two"))

session.query(MyClass).options(
                    defer(MyClass.attribute_one),
                    defer(MyClass.attribute_two))
```

To specify a deferred load of an attribute on a related class, the path can be specified one token at a time, specifying the loading style for each link along the chain. To leave the loading style for a link unchanged, use `orm.defaultload()`:

```
session.query(MyClass).options(defaultload("someattr").defer("some_column"))
```

A `Load` object that is present on a certain path can have `Load.defer()` called multiple times, each will operate on the same parent entity:

```
session.query(MyClass).options(
                defaultload("someattr").
                    defer("some_column").
                    defer("some_other_column").
                    defer("another_column")
    )
```

Parameters

- **key** – Attribute to be deferred.
- ***addl_attrs** – Deprecated; this option supports the old 0.8 style of specifying a path as a series of attributes, which is now superseded by the method-chained style.

See also:

deferred

`orm.undefer()`

`sqlalchemy.orm.deferred(*columns, **kw)`

Indicate a column-based mapped attribute that by default will not load unless accessed.

Parameters

- ***columns** – columns to be mapped. This is typically a single `Column` object, however a collection is supported in order to support multiple columns mapped under the same attribute.
- ****kw** – additional keyword arguments passed to `ColumnProperty`.

See also:

deferred

`sqlalchemy.orm.query_expression()`

Indicate an attribute that populates from a query-time SQL expression.

New in version 1.2.

See also:

mapper_query_expression

sqlalchemy.orm.**load_only**(**attrs*)

Indicate that for a particular entity, only the given list of column-based attribute names should be loaded; all others will be deferred.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

Example - given a class `User`, load only the `name` and `fullname` attributes:

```
session.query(User).options(load_only("name", "fullname"))
```

Example - given a relationship `User.addresses -> Address`, specify subquery loading for the `User.addresses` collection, but on each `Address` object load only the `email_address` attribute:

```
session.query(User).options(
        subqueryload("addresses").load_only("email_address")
)
```

For a `Query` that has multiple entities, the lead entity can be specifically referred to using the `Load` constructor:

```
session.query(User, Address).join(User.addresses).options(
            Load(User).load_only("name", "fullname"),
            Load(Address).load_only("email_addres")
        )
```

New in version 0.9.0.

sqlalchemy.orm.**undefer**(*key*, **addl_attrs*)

Indicate that the given column-oriented attribute should be undeferred, e.g. specified within the SELECT statement of the entity as a whole.

The column being undeferred is typically set up on the mapping as a `deferred()` attribute.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

Examples:

```
# undefer two columns
session.query(MyClass).options(undefer("col1"), undefer("col2"))

# undefer all columns specific to a single class using Load + *
session.query(MyClass, MyOtherClass).options(
    Load(MyClass).undefer("*"))
```

Parameters

- **key** – Attribute to be undeferred.
- ***addl_attrs** – Deprecated; this option supports the old 0.8 style of specifying a path as a series of attributes, which is now superseded by the method-chained style.

See also:

deferred

`orm.defer()`

`orm.undefer_group()`

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

Indicate that columns within the given deferred group name should be undeferred.

The columns being undeferred are set up on the mapping as `deferred()` attributes and include a "group" name.

E.g:

```
session.query(MyClass).options(undefer_group("large_attrs"))
```

To undefer a group of attributes on a related entity, the path can be spelled out using relationship loader options, such as `orm.defaultload()`:

```
session.query(MyClass).options(
    defaultload("someattr").undefer_group("large_attrs"))
```

Changed in version 0.9.0: `orm.undefer_group()` is now specific to a particiular entity load path.

See also:

deferred

`orm.defer()`

`orm.undefer()`

`sqlalchemy.orm.with_expression`(*key*, *expression*)

Apply an ad-hoc SQL expression to a "deferred expression" attribute.

This option is used in conjunction with the `orm.query_expression()` mapper-level construct that indicates an attribute which should be the target of an ad-hoc SQL expression.

E.g.:

```
sess.query(SomeClass).options(
    with_expression(SomeClass.x_y_expr, SomeClass.x + SomeClass.y)
)
```

New in version 1.2.

Parameters

- **key** – Attribute to be undeferred.
- **expr** – SQL expression to be applied to the attribute.

See also:

mapper_query_expression

Column Bundles

The `Bundle` may be used to query for groups of columns under one namespace.

New in version 0.9.0.

The bundle allows columns to be grouped together:

```
from sqlalchemy.orm import Bundle

bn = Bundle('mybundle', MyClass.data1, MyClass.data2)
for row in session.query(bn).filter(bn.c.data1 == 'd1'):
    print(row.mybundle.data1, row.mybundle.data2)
```

The bundle can be subclassed to provide custom behaviors when results are fetched. The method `Bundle.create_row_processor()` is given the `Query` and a set of "row processor" functions at query execution time; these processor functions when given a result row will return the individual attribute value, which can then be adapted into any kind of return data structure. Below illustrates replacing the usual `KeyedTuple` return structure with a straight Python dictionary:

```
from sqlalchemy.orm import Bundle

class DictBundle(Bundle):
    def create_row_processor(self, query, procs, labels):
        """Override create_row_processor to return values as dictionaries"""
        def proc(row):
            return dict(
                        zip(labels, (proc(row) for proc in procs))
                    )
        return proc
```

Changed in version 1.0: The `proc()` callable passed to the `create_row_processor()` method of custom `Bundle` classes now accepts only a single "row" argument.

A result from the above bundle will return dictionary values:

```
bn = DictBundle('mybundle', MyClass.data1, MyClass.data2)
for row in session.query(bn).filter(bn.c.data1 == 'd1'):
    print(row.mybundle['data1'], row.mybundle['data2'])
```

The `Bundle` construct is also integrated into the behavior of `composite()`, where it is used to return composite attributes as objects when queried as individual attributes.

2.4.2 Relationship Loading Techniques

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

The loading of relationships falls into three categories; **lazy** loading, **eager** loading, and **no** loading. Lazy loading refers to objects are returned from a query without the related objects loaded at first. When the given collection or reference is first accessed on a particular object, an additional SELECT statement is emitted such that the requested collection is loaded.

Eager loading refers to objects returned from a query with the related collection or scalar reference already loaded up front. The `Query` achieves this either by augmenting the SELECT statement it would normally emit with a JOIN to load in related rows simultaneously, or by emitting additional SELECT statements after the primary one to load collections or scalar references at once.

"No" loading refers to the disabling of loading on a given relationship, either that the attribute is empty and is just never loaded, or that it raises an error when it is accessed, in order to guard against unwanted lazy loads.

The primary forms of relationship loading are:

- **lazy loading** - available via `lazy='select'` or the `lazyload()` option, this is the form of loading that emits a SELECT statement at attribute access time to lazily load a related reference on a single object at a time. Lazy loading is detailed at lazy_loading.
- **joined loading** - available via `lazy='joined'` or the `joinedload()` option, this form of loading applies a JOIN to the given SELECT statement so that related rows are loaded in the same result set. Joined eager loading is detailed at joined_eager_loading.
- **subquery loading** - available via `lazy='subquery'` or the `subqueryload()` option, this form of loading emits a second SELECT statement which re-states the original query embedded inside of a subquery, then JOINs that subquery to the related table to be loaded to load all members of related collections / scalar references at once. Subquery eager loading is detailed at subquery_eager_loading.
- **select IN loading** - available via `lazy='selectin'` or the `selectinload()` option, this form of loading emits a second (or more) SELECT statement which assembles the primary key identifiers

of the parent objects into an IN clause, so that all members of related collections / scalar references are loaded at once by primary key. Select IN loading is detailed at selectin_eager_loading.

- **raise loading** - available via `lazy='raise'`, `lazy='raise_sql'`, or the `raiseload()` option, this form of loading is triggered at the same time a lazy load would normally occur, except it raises an ORM exception in order to guard against the application making unwanted lazy loads. An introduction to raise loading is at prevent_lazy_with_raiseload.
- **no loading** - available via `lazy='noload'`, or the `noload()` option; this loading style turns the attribute into an empty attribute that will never load or have any loading effect. "noload" is a fairly uncommon loader option.

Configuring Loader Strategies at Mapping Time

The loader strategy for a particular relationship can be configured at mapping time to take place in all cases where an object of the mapped type is loaded, in the absense of any query-level options that modify it. This is configured using the `relationship.lazy` parameter to `relationship()`; common values for this parameter include `select`, `joined`, `subquery` and `selectin`.

For example, to configure a relationship to use joined eager loading when the parent object is queried:

```
class Parent(Base):
    __tablename__ = 'parent'

    id = Column(Integer, primary_key=True)
    children = relationship("Child", lazy='joined')
```

Above, whenever a collection of `Parent` objects are loaded, each `Parent` will also have its `children` collection populated, using rows fetched by adding a JOIN to the query for `Parent` objects. See joined_eager_loading for background on this style of loading.

The default value of the `relationship.lazy` argument is `"select"`, which indicates lazy loading. See lazy_loading for further background.

Controlling Loading via Options

The other, and possibly more common way to configure loading strategies is to set them up on a per-query basis against specific attributes. Very detailed control over relationship loading is available using loader options; the most common are `joinedload()`, `subqueryload()`, `selectinload()` and `lazyload()`. The option accepts either the string name of an attribute against a parent, or for greater specificity can accommodate a class-bound attribute directly:

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

# same, using class-bound attribute
session.query(Parent).options(lazyload(Parent.children)).all()

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

The loader options can also be "chained" using **method chaining** to specify how loading should occur further levels deep:

```
session.query(Parent).options(
    joinedload(Parent.children).
    subqueryload(Child.subelements)).all()
```

Chained loader options can be applied against a "lazy" loaded collection. This means that when a collection or association is lazily loaded upon access, the specified option will then take effect:

```
session.query(Parent).options(
    lazyload(Parent.children).
    subqueryload(Child.subelements)).all()
```

Above, the query will return `Parent` objects without the `children` collections loaded. When the `children` collection on a particular `Parent` object is first accessed, it will lazy load the related objects, but additionally apply eager loading to the `subelements` collection on each member of `children`.

Using method chaining, the loader style of each link in the path is explicitly stated. To navigate along a path without changing the existing loader style of a particular attribute, the `defaultload()` method/function may be used:

```
session.query(A).options(
    defaultload("atob").
    joinedload("btoc")).all()
```

Lazy 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 typically issues a SQL call at the point of access in order to load the related object or objects:

```
>>> jack.addresses
{opensql}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]
{stop}[<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`. For this reason, while lazy loading can be expensive for related collections, in the case that one is loading lots of objects with simple many-to-ones against a relatively small set of possible target objects, lazy loading may be able to refer to these objects locally without emitting as many SELECT statements as there are parent objects.

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.

Lazy loading can be enabled for a given attribute that is normally configured in some other way using the `lazyload()` loader option:

```
from sqlalchemy.orm import lazyload

# force lazy loading for an attribute that is set to
# load some other way normally
session.query(User).options(lazyload(User.addresses))
```

Preventing unwanted lazy loads using raiseload

The `lazyload()` strategy produces an effect that is one of the most common issues referred to in object relational mapping; the N plus one problem, which states that for any N objects loaded, accessing their lazy-loaded attributes means there will be N+1 SELECT statements emitted. In SQLAlchemy, the usual mitigation for the N+1 problem is to make use of its very capable eager load system. However, eager loading requires that the attributes which are to be loaded be specified with the `Query` up front. The problem of code that may access other attributes that were not eagerly loaded, where lazy loading is not

desired, may be addressed using the `raiseload()` strategy; this loader strategy replaces the behavior of lazy loading with an informative error being raised:

```
from sqlalchemy.orm import raiseload
session.query(User).options(raiseload(User.addresses))
```

Above, a `User` object loaded from the above query will not have the `.addresses` collection loaded; if some code later on attempts to access this attribute, an ORM exception is raised.

`raiseload()` may be used with a so-called "wildcard" specifier to indicate that all relationships should use this strategy. For example, to set up only one attribute as eager loading, and all the rest as raise:

```
session.query(Order).options(
    joinedload(Order.items), raiseload('*'))
```

The above wildcard will apply to **all** relationships not just on `Order` besides `items`, but all those on the `Item` objects as well. To set up `raiseload()` for only the `Order` objects, specify a full path with `orm.Load`:

```
from sqlalchemy.orm import Load

session.query(Order).options(
    joinedload(Order.items), Load(Order).raiseload('*'))
```

Conversely, to set up the raise for just the `Item` objects:

```
session.query(Order).options(
    joinedload(Order.items).raiseload('*'))
```

See also:

wildcard_loader_strategies

Joined Eager Loading

Joined eager loading is the most fundamental style of eager loading in the ORM. It works by connecting a JOIN (by default a LEFT OUTER join) to the SELECT statement emitted by a `Query` and populates the target scalar/collection from the same result set as that of the parent.

At the mapping level, this looks like:

```
class Address(Base):
    # ...

    user = relationship(User, lazy="joined")
```

Joined eager loading is usually applied as an option to a query, rather than as a default loading option on the mapping, in particular when used for collections rather than many-to-one-references. This is achieved using the `joinedload()` loader option:

```
>>> jack = session.query(User).\
... options(joinedload(User.addresses)).\
... filter_by(name='jack').all()
{opensql}SELECT
    addresses_1.id AS addresses_1_id,
    addresses_1.email_address AS addresses_1_email_address,
    addresses_1.user_id AS addresses_1_user_id,
    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']
```

The JOIN emitted by default is a LEFT OUTER JOIN, to allow for a lead object that does not refer to a related row. For an attribute that is guaranteed to have an element, such as a many-to-one reference to a related object where the referencing foreign key is NOT NULL, the query can be made more efficient by using an inner join; this is available at the mapping level via the `relationship.innerjoin` flag:

```
class Address(Base):
    # ...

    user_id = Column(ForeignKey('users.id'), nullable=False)
    user = relationship(User, lazy="joined", innerjoin=True)
```

At the query option level, via the `joinedload.innerjoin` flag:

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

The JOIN will right-nest itself when applied in a chain that includes an OUTER JOIN:

```
>>> session.query(User).options(
...     joinedload(User.addresses).
...     joinedload(Address.widgets, innerjoin=True)).all()
{opensql}SELECT
    widgets_1.id AS widgets_1_id,
    widgets_1.name AS widgets_1_name,
    addresses_1.id AS addresses_1_id,
    addresses_1.email_address AS addresses_1_email_address,
    addresses_1.user_id AS addresses_1_user_id,
    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 JOIN widgets AS widgets_1 ON
    addresses_1.widget_id = widgets_1.id
) ON users.id = addresses_1.user_id
```

On older versions of SQLite, the above nested right JOIN may be re-rendered as a nested subquery. Older versions of SQLAlchemy would convert right-nested joins into subuqeries in all cases.

Joined eager loading and result set batching

A central concept of joined eager loading when applied to collections is that the `Query` object must de-duplicate rows against the leading entity being queried. Such as above, if the `User` object we loaded referred to three `Address` objects, the result of the SQL statement would have had three rows; yet the `Query` returns only one `User` object. As additional rows are received for a `User` object just loaded in a previous row, the additional columns that refer to new `Address` objects are directed into additional results within the `User.addresses` collection of that particular object.

This process is very transparent, however does imply that joined eager loading is incompatible with "batched" query results, provided by the `Query.yield_per()` method, when used for collection loading. Joined eager loading used for scalar references is however compatible with `Query.yield_per()`. The `Query.yield_per()` method will result in an exception thrown if a collection based joined eager loader is in play.

To "batch" queries with arbitrarily large sets of result data while maintaining compatibility with collection-based joined eager loading, emit multiple SELECT statements, each referring to a subset

of rows using the WHERE clause, e.g. windowing. Alternatively, consider using "select IN" eager loading which is **potentially** compatible with `Query.yield_per()`, provided that the database driver in use supports multiple, simultaneous cursors (SQLite, Postgresql drivers, not MySQL drivers or SQL Server ODBC drivers).

The Zen of Joined Eager Loading

Since joined eager loading seems to have many resemblances to the use of `Query.join()`, it often produces confusion as to when and how it should be used. It is critical to understand the distinction that while `Query.join()` is used to alter the results of a query, `joinedload()` goes through great lengths to **not** alter the results of the query, and instead hide the effects of the rendered join to only allow for related objects to be present.

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.

How `joinedload()` in particular achieves this result of not impacting entity rows returned in any way is that it creates an anonymous alias of the joins it adds to your query, so that they can't be referenced by other parts of the query. For example, the query below uses `joinedload()` to create a LEFT OUTER JOIN from `users` to `addresses`, however the `ORDER BY` added against `Address.email_address` is not valid - the `Address` entity is not named in the query:

```
>>> jack = session.query(User).\
... options(joinedload(User.addresses)).\
... filter(User.name=='jack').\
... order_by(Address.email_address).all()
{opensql}SELECT
    addresses_1.id AS addresses_1_id,
    addresses_1.email_address AS addresses_1_email_address,
    addresses_1.user_id AS addresses_1_user_id,
    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 = ?
ORDER BY addresses.email_address   <-- this part is wrong !
['jack']
```

Above, `ORDER BY addresses.email_address` is not valid since `addresses` is not in the FROM list. The correct way to load the `User` records and order by email address is to use `Query.join()`:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... filter(User.name=='jack').\
... order_by(Address.email_address).all()
{opensql}
SELECT
    users.id AS users_id,
    users.name AS users_name,
    users.fullname AS users_fullname,
    users.password AS users_password
FROM users
```

```
JOIN addresses ON users.id = addresses.user_id
WHERE users.name = ?
ORDER BY addresses.email_address
['jack']
```

The statement above is of course not the same as the previous one, in that the columns from `addresses` are not included in the result at all. We can add `joinedload()` back in, so that there are two joins - one is that which we are ordering on, the other is used anonymously to load the contents of the `User.addresses` collection:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... options(joinedload(User.addresses)).\
... filter(User.name=='jack').\
... order_by(Address.email_address).all()
{opensql}SELECT
    addresses_1.id AS addresses_1_id,
    addresses_1.email_address AS addresses_1_email_address,
    addresses_1.user_id AS addresses_1_user_id,
    users.id AS users_id, users.name AS users_name,
    users.fullname AS users_fullname,
    users.password AS users_password
FROM users JOIN addresses
    ON users.id = addresses.user_id
LEFT OUTER JOIN addresses AS addresses_1
    ON users.id = addresses_1.user_id
WHERE users.name = ?
ORDER BY addresses.email_address
['jack']
```

What we see above is that our usage of `Query.join()` is to supply JOIN clauses we'd like to use in subsequent query criterion, whereas our usage of `joinedload()` only concerns itself with the loading of the `User.addresses` collection, for each `User` in the result. In this case, the two joins most probably appear redundant - which they are. If we wanted to use just one JOIN for collection loading as well as ordering, we use the `contains_eager()` option, described in contains_eager below. But to see why `joinedload()` does what it does, consider if we were **filtering** on a particular `Address`:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... options(joinedload(User.addresses)).\
... filter(User.name=='jack').\
... filter(Address.email_address=='someaddress@foo.com').\
... all()
{opensql}SELECT
    addresses_1.id AS addresses_1_id,
    addresses_1.email_address AS addresses_1_email_address,
    addresses_1.user_id AS addresses_1_user_id,
    users.id AS users_id, users.name AS users_name,
    users.fullname AS users_fullname,
    users.password AS users_password
FROM users JOIN addresses
    ON users.id = addresses.user_id
LEFT OUTER JOIN addresses AS addresses_1
    ON users.id = addresses_1.user_id
WHERE users.name = ? AND addresses.email_address = ?
['jack', 'someaddress@foo.com']
```

Above, we can see that the two JOINs have very different roles. One will match exactly one row, that of the join of `User` and `Address` where `Address.email_address=='someaddress@foo.com'`. The other LEFT OUTER JOIN will match *all* `Address` rows related to `User`, and is only used to populate the `User.addresses` collection, for those `User` objects that are returned.

By changing the usage of `joinedload()` to another style of loading, we can change how the collection is loaded completely independently of SQL used to retrieve the actual `User` rows we want. Below we change `joinedload()` into `subqueryload()`:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... options(subqueryload(User.addresses)).\
... filter(User.name=='jack').\
... filter(Address.email_address=='someaddress@foo.com').\
... all()
{opensql}SELECT
    users.id AS users_id,
    users.name AS users_name,
    users.fullname AS users_fullname,
    users.password AS users_password
FROM users
JOIN addresses ON users.id = addresses.user_id
WHERE
    users.name = ?
    AND addresses.email_address = ?
['jack', 'someaddress@foo.com']

# ... subqueryload() emits a SELECT in order
# to load all address records ...
```

When using joined eager loading, if the query contains a modifier that impacts the rows returned externally to the joins, such as when using DISTINCT, LIMIT, OFFSET or equivalent, the completed statement is first wrapped inside a subquery, and the joins used specifically for joined eager loading are applied to the subquery. SQLAlchemy's joined eager loading goes the extra mile, and then ten miles further, to absolutely ensure that it does not affect the end result of the query, only the way collections and related objects are loaded, no matter what the format of the query is.

See also:

contains_eager - using `contains_eager()`

Subquery Eager Loading

Subqueryload eager loading is configured in the same manner as that of joined eager loading; for the `relationship.lazy` parameter, we would specify `"subquery"` rather than `"joined"`, and for the option we use the `subqueryload()` option rather than the `joinedload()` option.

The operation of subquery eager loading is to emit a second SELECT statement for each relationship to be loaded, across all result objects at once. This SELECT statement refers to the original SELECT statement, wrapped inside of a subquery, so that we retrieve the same list of primary keys for the primary object being returned, then link that to the sum of all the collection members to load them at once:

```
>>> jack = session.query(User).\
... options(subqueryload(User.addresses)).\
... filter_by(name='jack').all()
{opensql}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 subqueryload strategy has many advantages over joined eager loading in the area of loading collections. First, it allows the original query to proceed without changing it at all, not introducing in particular a LEFT OUTER JOIN that may make it less efficient. Secondly, it allows for many collections to be eagerly loaded without producing a single query that has many JOINs in it, which can be even less efficient; each relationship is loaded in a fully separate query. Finally, because the additional query only needs to load the collection items and not the lead object, it can use an inner JOIN in all cases for greater query efficiency.

Disadvantages of subqueryload include that the complexity of the original query is transferred to the relationship queries, which when combined with the use of a subquery, can on some backends in some cases (notably MySQL) produce significantly slow queries. Additionally, the subqueryload strategy can only load the full contents of all collections at once, is therefore incompatible with "batched" loading supplied by `Query.yield_per()`, both for collection and scalar relationships.

The newer style of loading provided by `selectinload()` solves these limitations of `subqueryload()`.

See also:

selectin_eager_loading

The Importance of Ordering

A query which makes use of `subqueryload()` in conjunction with a limiting modifier such as `Query.first()`, `Query.limit()`, or `Query.offset()` should **always** include `Query.order_by()` against unique column(s) such as the primary key, so that the additional queries emitted by `subqueryload()` include the same ordering as used by the parent query. Without it, there is a chance that the inner query could return the wrong rows:

```
# incorrect, no ORDER BY
session.query(User).options(
    subqueryload(User.addresses)).first()

# incorrect if User.name is not unique
session.query(User).options(
    subqueryload(User.addresses)
).order_by(User.name).first()

# correct
session.query(User).options(
    subqueryload(User.addresses)
).order_by(User.name, User.id).first()
```

See also:

faq_subqueryload_limit_sort - detailed example

Select IN loading

Select IN loading is similar in operation to subquery eager loading, however the SELECT statement which is emitted has a much simpler structure than that of subquery eager loading. Additionally, select IN loading applies itself to subsets of the load result at a time, so unlike joined and subquery eager

loading, is compatible with batching of results using `Query.yield_per()`, provided the database driver supports simultaneous cursors.

New in version 1.2.

"Select IN" eager loading is provided using the `"selectin"` argument to `relationship.lazy` or by using the `selectinload()` loader option. This style of loading emits a SELECT that refers to the primary key values of the parent object inside of an IN clause, in order to load related associations:

```
>>> jack = session.query(User).\
... options(selectinload('addresses')).\
... filter(or_(User.name == 'jack', User.name == 'ed')).all()
{opensql}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 = ? OR users.name = ?
('jack', 'ed')
SELECT
    users_1.id AS users_1_id,
    addresses.id AS addresses_id,
    addresses.email_address AS addresses_email_address,
    addresses.user_id AS addresses_user_id
FROM users AS users_1
JOIN addresses ON users_1.id = addresses.user_id
WHERE users_1.id IN (?, ?)
ORDER BY users_1.id, addresses.id
(5, 7)
```

Above, the second SELECT refers to `users_1.id IN (5, 7)`, where the "5" and "7" are the primary key values for the previous two `User` objects loaded; after a batch of objects are completely loaded, their primary key values are injected into the `IN` clause for the second SELECT.

"Select IN" loading is the newest form of eager loading added to SQLAlchemy as of the 1.2 series. Things to know about this kind of loading include:

- The SELECT statement emitted by the "selectin" loader strategy, unlike that of "subquery", does not require a subquery nor does it inherit any of the performance limitations of the original query; the lookup is a simple primary key lookup and should have high performance.
- The special ordering requirements of subqueryload described at subqueryload_ordering also don't apply to selectin loading; selectin is always linking directly to a parent primary key and can't really return the wrong result.
- "selectin" loading, unlike joined or subquery eager loading, always emits its SELECT in terms of the immediate parent objects just loaded, and not the original type of object at the top of the chain. So if eager loading many levels deep, "selectin" loading still uses exactly one JOIN in the statement. joined and subquery eager loading always refer to multiple JOINs up to the original parent.
- "selectin" loading produces a SELECT statement of a predictable structure, independent of that of the original query. As such, taking advantage of a new feature with `ColumnOperators.in_()` that allows it to work with cached queries, the selectin loader makes full use of the `sqlalchemy.ext.baked` extension to cache generated SQL and greatly cut down on internal function call overhead.
- The strategy will only query for at most 500 parent primary key values at a time, as the primary keys are rendered into a large IN expression in the SQL statement. Some databases like Oracle have a hard limit on how large an IN expression can be, and overall the size of the SQL string shouldn't be arbitrarily large. So for large result sets, "selectin" loading will emit a SELECT per 500 parent rows returned. These SELECT statements emit with minimal Python overhead due to the "baked" queries and also minimal SQL overhead as they query against primary key directly.

- "selectin" loading is the only eager loading that can work in conjunction with the "batching" feature provided by `Query.yield_per()`, provided the database driver supports simultaneous cursors. As it only queries for related items against specific result objects, "selectin" loading allows for eagerly loaded collections against arbitrarily large result sets with a top limit on memory use when used with `Query.yield_per()`.

  Current database drivers that support simultaneous cursors include SQLite, Postgresql. The MySQL drivers mysqlclient and pymysql currently **do not** support simultaneous cursors, nor do the ODBC drivers for SQL Server.

- As "selectin" loading relies upon IN, for a mapping with composite primary keys, it must use the "tuple" form of IN, which looks like `WHERE (table.column_a, table.column_b) IN ((?, ?), (?, ?), (?, ?))`. This syntax is not supported on every database; currently it is known to be only supported by modern Postgresql and MySQL versions. Therefore **selectin loading is not platform-agnostic for composite primary keys**. There is no special logic in SQLAlchemy to check ahead of time which platforms support this syntax or not; if run against a non-supporting platform (such as SQLite), the database will return an error immediately. An advantage to SQLAlchemy just running the SQL out for it to fail is that if a database like SQLite does start supporting this syntax, it will work without any changes to SQLAlchemy.

In general, "selectin" loading is probably superior to "subquery" eager loading in most ways, save for the syntax requirement with composite primary keys and possibly that it may emit many SELECT statements for larger result sets. As always, developers should spend time looking at the statements and results generated by their applications in development to check that things are working efficiently.

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 joined and subquery eager loading always join from the original parent class; if loading a collection four levels deep, there will be four JOINs out to the parent. selectin loading on the other hand will always have exactly one JOIN to the immediate parent table.
- Using selectin loading, the load of 100 objects will also emit two SQL statements, the second of which refers to the 100 primary keys of the objects loaded. selectin loading will however render at

most 500 primary key values into a single SELECT statement; so for a lead collection larger than 500, there will be a SELECT statement emitted for each batch of 500 objects selected.

- Using multiple levels of depth with selectin loading does not incur the "cartesian" issue that joined and subquery eager loading have; the queries for selectin loading have the best performance characteristics and the fewest number of rows. The only caveat is that there might be more than one SELECT emitted depending on the size of the lead result.
- selectin loading, unlike joined (when using collections) and subquery eager loading (all kinds of relationships), is potentially compatible with result set batching provided by `Query.yield_per()` assuming an appropriate database driver, so may be able to allow batching for large result sets.
- 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` set to `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.
- Selectin loading will also issue a second load for all the child objects (and as stated before, for larger results it will emit a SELECT per 500 rows), so for a load of 100 objects there would be two SQL statements emitted. The query itself still has to JOIN to the parent table, so again there's not too much advantage to selectin loading for many-to-one vs. joined eager loading save for the use of INNER JOIN in all cases.

Polymorphic Eager Loading

Specification of polymorpic options on a per-eager-load basis is supported. See the section eagerloading_polymorphic_subtypes for examples of the `PropComparator.of_type()` method in conjunction with the `orm.with_polymorphic()` function.

Wildcard Loading Strategies

Each of `joinedload()`, `subqueryload()`, `lazyload()`, `selectinload()`, `noload()`, and `raiseload()` can be used to set the default style of `relationship()` loading for a particular query, affecting all `relationship()` -mapped attributes not otherwise specified in the `Query`. This feature is available by passing the string `'*'` as the argument to any of these options:

```
session.query(MyClass).options(lazyload('*'))
```

Above, the `lazyload('*')` option will supersede the `lazy` setting of all `relationship()` constructs in use for that query, except for those which use the `'dynamic'` style of loading. If some relationships specify `lazy='joined'` or `lazy='subquery'`, for example, using `lazyload('*')` will unilaterally cause all those relationships to use `'select'` loading, e.g. emit a SELECT statement when each attribute is accessed.

The option does not supersede loader options stated in the query, such as `eagerload()`, `subqueryload()`, etc. The query below will still use joined loading for the `widget` relationship:

```
session.query(MyClass).options(
    lazyload('*'),
    joinedload(MyClass.widget)
)
```

If multiple `'*'` options are passed, the last one overrides those previously passed.

Per-Entity Wildcard Loading Strategies

A variant of the wildcard loader strategy is the ability to set the strategy on a per-entity basis. For example, if querying for `User` and `Address`, we can instruct all relationships on `Address` only to use lazy loading by first applying the `Load` object, then specifying the `*` as a chained option:

```
session.query(User, Address).options(
    Load(Address).lazyload('*'))
```

Above, all relationships on `Address` will be set to a lazy load.

Routing Explicit Joins/Statements into Eagerly Loaded Collections

The behavior of `joinedload()` is such that joins are created automatically, using anonymous aliases as targets, 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, we specify a join between `User` and `Address` and additionally establish this as the basis for eager loading of `User.addresses`:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    addresses = relationship("Address")

class Address(Base):
    __tablename__ = 'address'

    # ...

q = session.query(User).join(User.addresses).\
            options(contains_eager(User.addresses))
```

If the "eager" portion of the statement is "aliased", the `alias` keyword argument to `contains_eager()` may be used to indicate it. This is sent as a reference to an `aliased()` or `Alias` construct:

```
# 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()
```

```
{opensql}SELECT
    users.user_id AS users_user_id,
    users.user_name AS users_user_name,
    adalias.address_id AS adalias_address_id,
    adalias.user_id AS adalias_user_id,
    adalias.email_address AS adalias_email_address,
    (...other columns...)
FROM users
LEFT OUTER JOIN email_addresses AS email_addresses_1
ON users.user_id = email_addresses_1.user_id
```

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').
    contains_eager('items'))
```

Or using the class-bound descriptor:

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

Using contains_eager() to load a custom-filtered collection result

When we use `contains_eager()`, *we* are constructing ourselves the SQL that will be used to populate collections. From this, it naturally follows that we can opt to **modify** what values the collection is intended to store, by writing our SQL to load a subset of elements for collections or scalar attributes.

As an example, we can load a `User` object and eagerly load only particular addresses into its `.addresses` collection just by filtering:

```
q = session.query(User).join(User.addresses).\
            filter(Address.email.like('%ed%')).\
            options(contains_eager(User.addresses))
```

The above query will load only `User` objects which contain at least `Address` object that contains the substring `'ed'` in its `email` field; the `User.addresses` collection will contain **only** these `Address` entries, and *not* any other `Address` entries that are in fact associated with the collection.

Warning: Keep in mind that when we load only a subset of objects into a collection, that collection no longer represents what's actually in the database. If we attempted to add entries to this collection, we might find ourselves conflicting with entries that are already in the database but not locally loaded.

In addition, the **collection will fully reload normally** once the object or attribute is expired. This expiration occurs whenever the `Session.commit()`, `Session.rollback()` methods are used assuming default session settings, or the `Session.expire_all()` or `Session.expire()` methods are used.

For these reasons, prefer returning separate fields in a tuple rather than artificially altering a collection, when an object plus a custom set of related objects is desired:

```
q = session.query(User, Address).join(User.addresses).\
            filter(Address.email.like('%ed%'))
```

Advanced Usage with Arbitrary Statements

The `alias` argument can be more creatively used, in that it can be made to represent any set of arbitrary names to match up into a statement. Below it is linked to a `select()` which links a set of column objects to a string SQL statement:

```
# 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))
```

Creating Custom Load Rules

Warning: This is an advanced technique! Great care and testing should be applied.

The ORM has various edge cases where the value of an attribute is locally available, however the ORM itself doesn't have awareness of this. There are also cases when a user-defined system of loading attributes is desirable. To support the use case of user-defined loading systems, a key function `attributes.set_committed_value()` is provided. This function is basically equivalent to Python's own `setattr()` function, except that when applied to a target object, SQLAlchemy's "attribute history" system which is used to determine flush-time changes is bypassed; the attribute is assigned in the same way as if the ORM loaded it that way from the database.

The use of `attributes.set_committed_value()` can be combined with another key event known as `InstanceEvents.load()` to produce attribute-population behaviors when an object is loaded. One such example is the bi-directional "one-to-one" case, where loading the "many-to-one" side of a one-to-one should also imply the value of the "one-to-many" side. The SQLAlchemy ORM does not consider backrefs when loading related objects, and it views a "one-to-one" as just another "one-to-many", that just happens to be one row.

Given the following mapping:

```
from sqlalchemy import Integer, ForeignKey, Column
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    b_id = Column(ForeignKey('b.id'))
    b = relationship(
        "B",
        backref=backref("a", uselist=False),
```

```
        lazy='joined')


class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
```

If we query for an `A` row, and then ask it for `a.b.a`, we will get an extra SELECT:

```
>>> a1.b.a
SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a
WHERE ? = a.b_id
```

This SELECT is redundant because `b.a` is the same value as `a1`. We can create an on-load rule to populate this for us:

```
from sqlalchemy import event
from sqlalchemy.orm import attributes

@event.listens_for(A, "load")
def load_b(target, context):
    if 'b' in target.__dict__:
        attributes.set_committed_value(target.b, 'a', target)
```

Now when we query for `A`, we will get `A.b` from the joined eager load, and `A.b.a` from our event:

```
a1 = s.query(A).first()
{opensql}SELECT
    a.id AS a_id,
    a.b_id AS a_b_id,
    b_1.id AS b_1_id
FROM a
LEFT OUTER JOIN b AS b_1 ON b_1.id = a.b_id
 LIMIT ? OFFSET ?
(1, 0)
{stop}assert a1.b.a is a1
```

Relationship Loader API

sqlalchemy.orm.**contains_alias**(*alias*)

Return a `MapperOption` that will indicate to the `Query` that the main table has been aliased.

This is a seldom-used option to suit the very rare case that `contains_eager()` is being used in conjunction with a user-defined SELECT statement that aliases the parent table. E.g.:

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

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

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

```
# then get results via the statement
results = query.from_statement(statement).all()
```

Parameters **alias** – is the string name of an alias, or a `Alias` object representing the alias.

`sqlalchemy.orm.contains_eager(*keys, **kw)`

Indicate that the given attribute should be eagerly loaded from columns stated manually in the query.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

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))
```

When using `contains_eager()` in conjunction with inherited subclasses, the `RelationshipProperty.of_type()` modifier should also be used in order to set up the pathing properly:

```
sess.query(Company).\
    outerjoin(Company.employees.of_type(Manager)).\
    options(
        contains_eager(
            Company.employees.of_type(Manager),
            alias=Manager)
    )
```

See also:

loading_toplevel

contains_eager

`sqlalchemy.orm.defaultload(*keys)`

Indicate an attribute should load using its default loader style.

This method is used to link to other loader options further into a chain of attributes without altering the loader style of the links along the chain. For example, to set joined eager loading for an element of an element:

```
session.query(MyClass).options(
    defaultload(MyClass.someattribute).
    joinedload(MyOtherClass.someotherattribute)
)
```

defaultload() is also useful for setting column-level options on a related class, namely that of defer() and undefer():

```
session.query(MyClass).options(
    defaultload(MyClass.someattribute).
    defer("some_column").
    undefer("some_other_column")
)
```

See also:

relationship_loader_options

deferred_loading_w_multiple

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

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

sqlalchemy.orm.**immediateload**(**keys*)
Indicate that the given attribute should be loaded using an immediate load with a per-attribute SELECT statement.

The immediateload() option is superseded in general by the selectinload() option, which performs the same task more efficiently by emitting a SELECT for all loaded objects.

This function is part of the Load interface and supports both method-chained and standalone operation.

See also:

loading_toplevel

selectin_eager_loading

sqlalchemy.orm.**joinedload**(**keys*, ***kw*)
Indicate that the given attribute should be loaded using joined eager loading.

This function is part of the Load interface and supports both method-chained and standalone operation.

examples:

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

# joined-load Order.items and then Item.keywords
query(Order).options(
    joinedload(Order.items).joinedload(Item.keywords))

# lazily load Order.items, but when Items are loaded,
# joined-load the keywords collection
query(Order).options(
    lazyload(Order.items).joinedload(Item.keywords))
```

Parameters **innerjoin** – if True, indicates that the joined eager load should use an inner join instead of the default of left outer join:

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

In order to chain multiple eager joins together where some may be OUTER and others INNER, right-nested joins are used to link them:

```
query(A).options(
    joinedload(A.bs, innerjoin=False).
        joinedload(B.cs, innerjoin=True)
)
```

The above query, linking A.bs via "outer" join and B.cs via "inner" join would render the joins as "a LEFT OUTER JOIN (b JOIN c)". When using older versions of SQLite (< 3.7.16), this form of JOIN is translated to use full subqueries as this syntax is otherwise not directly supported.

The `innerjoin` flag can also be stated with the term `"unnested"`. This indicates that an INNER JOIN should be used, *unless* the join is linked to a LEFT OUTER JOIN to the left, in which case it will render as LEFT OUTER JOIN. For example, supposing `A.bs` is an outerjoin:

```
query(A).options(
    joinedload(A.bs).
        joinedload(B.cs, innerjoin="unnested")
)
```

The above join will render as "a LEFT OUTER JOIN b LEFT OUTER JOIN c", rather than as "a LEFT OUTER JOIN (b JOIN c)".

Note: The "unnested" flag does **not** affect the JOIN rendered from a many-to-many association table, e.g. a table configured as `relationship.secondary`, to the target table; for correctness of results, these joins are always INNER and are therefore right-nested if linked to an OUTER join.

Changed in version 1.0.0: `innerjoin=True` now implies `innerjoin="nested"`, whereas in 0.9 it implied `innerjoin="unnested"`. In order to achieve the pre-1.0 "unnested" inner join behavior, use the value `innerjoin="unnested"`. See migration_3008.

Note: The joins produced by `orm.joinedload()` are **anonymously aliased**. The criteria by which the join proceeds cannot be modified, nor can the `Query` refer to these joins in any way, including ordering. See zen_of_eager_loading for further detail.

To produce a specific SQL JOIN which is explicitly available, use `Query.join()`. To combine explicit JOINs with eager loading of collections, use `orm.contains_eager()`; see contains_eager.

See also:

loading_toplevel

joined_eager_loading

`sqlalchemy.orm.joinedload_all(*keys, **kw)`

Produce a standalone "all" option for `orm.joinedload()`.

Deprecated since version 0.9.0: The "_all()" style is replaced by method chaining, e.g.:

```
session.query(MyClass).options(
    joinedload("someattribute").joinedload("anotherattribute")
)
```

`sqlalchemy.orm.lazyload(*keys)`

Indicate that the given attribute should be loaded using "lazy" loading.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

See also:

loading_toplevel

lazy_loading

class sqlalchemy.orm.Load(*entity*)

Represents loader options which modify the state of a Query in order to affect how various mapped attributes are loaded.

The Load object is in most cases used implicitly behind the scenes when one makes use of a query option like joinedload(), defer(), or similar. However, the Load object can also be used directly, and in some cases can be useful.

To use Load directly, instantiate it with the target mapped class as the argument. This style of usage is useful when dealing with a Query that has multiple entities:

```
myopt = Load(MyClass).joinedload("widgets")
```

The above myopt can now be used with Query.options(), where it will only take effect for the MyClass entity:

```
session.query(MyClass, MyOtherClass).options(myopt)
```

One case where Load is useful as public API is when specifying "wildcard" options that only take effect for a certain class:

```
session.query(Order).options(Load(Order).lazyload('*'))
```

Above, all relationships on Order will be lazy-loaded, but other attributes on those descendant objects will load using their normal loader strategy.

See also:

loading_toplevel

sqlalchemy.orm.noload(**keys*)

Indicate that the given relationship attribute should remain unloaded.

This function is part of the Load interface and supports both method-chained and standalone operation.

orm.noload() applies to relationship() attributes; for column-based attributes, see orm.defer().

See also:

loading_toplevel

sqlalchemy.orm.raiseload(**keys*, ***kw*)

Indicate that the given relationship attribute should disallow lazy loads.

A relationship attribute configured with orm.raiseload() will raise an InvalidRequestError upon access. The typical way this is useful is when an application is attempting to ensure that all relationship attributes that are accessed in a particular context would have been already loaded via eager loading. Instead of having to read through SQL logs to ensure lazy loads aren't occurring, this strategy will cause them to raise immediately.

Parameters **sql_only** – if True, raise only if the lazy load would emit SQL, but not if it is only checking the identity map, or determining that the related value should just be None due to missing keys. When False, the strategy will raise for all varieties of lazyload.

This function is part of the Load interface and supports both method-chained and standalone operation.

orm.raiseload() applies to relationship() attributes only.

New in version 1.1.

See also:

loading_toplevel

prevent_lazy_with_raiseload

sqlalchemy.orm.**selectinload**(**keys*)

Indicate that the given attribute should be loaded using SELECT IN eager loading.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

examples:

```
# selectin-load the "orders" collection on "User"
query(User).options(selectinload(User.orders))

# selectin-load Order.items and then Item.keywords
query(Order).options(
    selectinload(Order.items).selectinload(Item.keywords))

# lazily load Order.items, but when Items are loaded,
# selectin-load the keywords collection
query(Order).options(
    lazyload(Order.items).selectinload(Item.keywords))
```

New in version 1.2.

See also:

loading_toplevel

selectin_eager_loading

sqlalchemy.orm.**selectinload_all**(**keys*)

Produce a standalone "all" option for `orm.selectinload()`.

Deprecated since version 0.9.0: The "_all()" style is replaced by method chaining, e.g.:

```
session.query(MyClass).options(
    selectinload("someattribute").selectinload("anotherattribute")
)
```

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

Indicate that the given attribute should be loaded using subquery eager loading.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

examples:

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

# subquery-load Order.items and then Item.keywords
query(Order).options(
    subqueryload(Order.items).subqueryload(Item.keywords))

# lazily load Order.items, but when Items are loaded,
# subquery-load the keywords collection
query(Order).options(
    lazyload(Order.items).subqueryload(Item.keywords))
```

See also:

loading_toplevel

subquery_eager_loading

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

Produce a standalone "all" option for `orm.subqueryload()`.

Deprecated since version 0.9.0: The "_all()" style is replaced by method chaining, e.g.:

```
session.query(MyClass).options(
    subqueryload("someattribute").subqueryload("anotherattribute")
)
```

2.4.3 Loading Inheritance Hierarchies

When classes are mapped in inheritance hierarchies using the "joined", "single", or "concrete" table inheritance styles as described at inheritance_toplevel, the usual behavior is that a query for a particular base class will also yield objects corresponding to subclasses as well. When a single query is capable of returning a result with a different class or subclasses per result row, we use the term "polymorphic loading".

Within the realm of polymorphic loading, specifically with joined and single table inheritance, there is an additional problem of which subclass attributes are to be queried up front, and which are to be loaded later. When an attribute of a particular subclass is queried up front, we can use it in our query as something to filter on, and it also will be loaded when we get our objects back. If it's not queried up front, it gets loaded later when we first need to access it. Basic control of this behavior is provided using the `orm.with_polymorphic()` function, as well as two variants, the mapper configuration `mapper.with_polymorphic` in conjunction with the `mapper.polymorphic_load` option, and the `Query` -level `Query.with_polymorphic()` method. The "with_polymorphic" family each provide a means of specifying which specific subclasses of a particular base class should be included within a query, which implies what columns and tables will be available in the SELECT.

Using with_polymorphic

For the following sections, assume the `Employee` / `Engineer` / `Manager` examples introduced in inheritance_toplevel.

Normally, when a `Query` specifies the base class of an inheritance hierarchy, only the columns that are local to that base class are queried:

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

Above, for both single and joined table inheritance, only the columns local to `Employee` will be present in the SELECT. We may get back instances of `Engineer` or `Manager`, however they will not have the additional attributes loaded until we first access them, at which point a lazy load is emitted.

Similarly, if we wanted to refer to columns mapped to `Engineer` or `Manager` in our query that's against `Employee`, these columns aren't available directly in either the single or joined table inheritance case, since the `Employee` entity does not refer to these columns (note that for single-table inheritance, this is common if Declarative is used, but not for a classical mapping).

To solve both of these issues, the `orm.with_polymorphic()` function provides a special `AliasedClass` that represents a range of columns across subclasses. This object can be used in a `Query` like any other alias. When queried, it represents all the columns present in the classes given:

```
from sqlalchemy.orm import with_polymorphic

eng_plus_manager = with_polymorphic(Employee, [Engineer, Manager])

query = session.query(eng_plus_manager)
```

If the above mapping were using joined table inheritance, the SELECT statement for the above would be:

```
query.all()
{opensql}
SELECT employee.id AS employee_id,
    engineer.id AS engineer_id,
    manager.id AS manager_id,
    employee.name AS employee_name,
    employee.type AS employee_type,
    engineer.engineer_info AS engineer_engineer_info,
    manager.manager_data AS manager_manager_data
FROM employee
    LEFT OUTER JOIN engineer
    ON employee.id = engineer.id
    LEFT OUTER JOIN manager
    ON employee.id = manager.id
[]
```

Where above, the additional tables / columns for "engineer" and "manager" are included. Similar behavior occurs in the case of single table inheritance.

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

```
# include columns for Engineer
entity = with_polymorphic(Employee, Engineer)

# include columns for Engineer, Manager
entity = with_polymorphic(Employee, [Engineer, Manager])

# include columns for all mapped subclasses
entity = with_polymorphic(Employee, '*')
```

Using aliasing with with_polymorphic

The `orm.with_polymorphic()` function also provides "aliasing" of the polymorphic selectable itself, meaning, two different `orm.with_polymorphic()` entities, referring to the same class hierarchy, can be used together. This is available using the `orm.with_polymorphic.aliased` flag. For a polymorphic selectable that is across multiple tables, the default behavior is to wrap the selectable into a subquery. Below we emit a query that will select for "employee or manager" paired with "employee or engineer" on employees with the same name:

```
engineer_employee = with_polymorphic(
    Employee, [Engineer], aliased=True)
manager_employee = with_polymorphic(
    Employee, [Manager], aliased=True)

q = s.query(engineer_employee, manager_employee).\
    join(
        manager_employee,
        and_(
            engineer_employee.id > manager_employee.id,
            engineer_employee.name == manager_employee.name
        )
)
q.all()
{opensql}
SELECT
    anon_1.employee_id AS anon_1_employee_id,
    anon_1.employee_name AS anon_1_employee_name,
```

```
    anon_1.employee_type AS anon_1_employee_type,
    anon_1.engineer_id AS anon_1_engineer_id,
    anon_1.engineer_engineer_name AS anon_1_engineer_engineer_name,
    anon_2.employee_id AS anon_2_employee_id,
    anon_2.employee_name AS anon_2_employee_name,
    anon_2.employee_type AS anon_2_employee_type,
    anon_2.manager_id AS anon_2_manager_id,
    anon_2.manager_manager_name AS anon_2_manager_manager_name
FROM (
    SELECT
        employee.id AS employee_id,
        employee.name AS employee_name,
        employee.type AS employee_type,
        engineer.id AS engineer_id,
        engineer.engineer_name AS engineer_engineer_name
    FROM employee
    LEFT OUTER JOIN engineer ON employee.id = engineer.id
) AS anon_1
JOIN (
    SELECT
        employee.id AS employee_id,
        employee.name AS employee_name,
        employee.type AS employee_type,
        manager.id AS manager_id,
         manager.manager_name AS manager_manager_name
    FROM employee
    LEFT OUTER JOIN manager ON employee.id = manager.id
) AS anon_2
ON anon_1.employee_id > anon_2.employee_id
AND anon_1.employee_name = anon_2.employee_name
```

The creation of subqueries above is very verbose. While it creates the best encapsulation of the two distinct queries, it may be inefficient. `orm.with_polymorphic()` includes an additional flag to help with this situation, `orm.with_polymorphic.flat`, which will "flatten" the subquery / join combination into straight joins, applying aliasing to the individual tables instead. Setting `orm.with_polymorphic.flat` implies `orm.with_polymorphic.aliased`, so only one flag is necessary:

```
engineer_employee = with_polymorphic(
    Employee, [Engineer], flat=True)
manager_employee = with_polymorphic(
    Employee, [Manager], flat=True)

q = s.query(engineer_employee, manager_employee).\
    join(
        manager_employee,
        and_(
            engineer_employee.id > manager_employee.id,
            engineer_employee.name == manager_employee.name
        )
)
q.all()
{opensql}
SELECT
    employee_1.id AS employee_1_id,
    employee_1.name AS employee_1_name,
    employee_1.type AS employee_1_type,
    engineer_1.id AS engineer_1_id,
    engineer_1.engineer_name AS engineer_1_engineer_name,
    employee_2.id AS employee_2_id,
    employee_2.name AS employee_2_name,
    employee_2.type AS employee_2_type,
    manager_1.id AS manager_1_id,
```

```
    manager_1.manager_name AS manager_1_manager_name
FROM employee AS employee_1
LEFT OUTER JOIN engineer AS engineer_1
ON employee_1.id = engineer_1.id
JOIN (
    employee AS employee_2
    LEFT OUTER JOIN manager AS manager_1
    ON employee_2.id = manager_1.id
)
ON employee_1.id > employee_2.id
AND employee_1.name = employee_2.name
```

Note above, when using `orm.with_polymorphic.flat`, it is often the case when used in conjunction with joined table inheritance that we get a right-nested JOIN in our statement. Some older databases, in particular older versions of SQLite, may have a problem with this syntax, although virtually all modern database versions now support this syntax.

Referring to Specific Subclass Attributes

The entity returned by `orm.with_polymorphic()` is an `AliasedClass` object, which can be used in a `Query` like any other alias, including named attributes for those attributes on the `Employee` class. In our previous example, `eng_plus_manager` becomes the entity that we use to refer to the three-way outer join above. It also includes namespaces for each class named in the list of classes, so that attributes specific to those subclasses can be called upon as well. The following example illustrates calling upon attributes specific to `Engineer` as well as `Manager` in terms of `eng_plus_manager`:

```
eng_plus_manager = with_polymorphic(Employee, [Engineer, Manager])
query = session.query(eng_plus_manager).filter(
                or_(
                    eng_plus_manager.Engineer.engineer_info=='x',
                    eng_plus_manager.Manager.manager_data=='y'
                )
            )
```

Setting with_polymorphic at mapper configuration time

The `orm.with_polymorphic()` function serves the purpose of allowing "eager" loading of attributes from subclass tables, as well as the ability to refer to the attributes from subclass tables at query time. Historically, the "eager loading" of columns has been the more important part of the equation. So just as eager loading for relationships can be specified as a configurational option, the `mapper.with_polymorphic` configuration parameter allows an entity to use a polymorphic load by default. We can add the parameter to our `Employee` mapping first introduced at joined_inheritance:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'employee',
        'polymorphic_on':type,
        'with_polymorphic': '*'
    }
```

Above is a common setting for `mapper.with_polymorphic`, which is to indicate an asterisk to load all subclass columns. In the case of joined table inheritance, this option should be used sparingly, as it implies that the mapping will always emit a (often large) series of LEFT OUTER JOIN to many tables,

which is not efficient from a SQL perspective. For single table inheritance, specifying the asterisk is often a good idea as the load is still against a single table only, but an additional lazy load of subclass-mapped columns will be prevented.

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

The `mapper.with_polymorphic` option also accepts a list of classes just like `orm.with_polymorphic()` to polymorphically load among a subset of classes. However, when using Declarative, providing classes to this list is not directly possible as the subclasses we'd like to add are not available yet. Instead, we can specify on each subclass that they should individually participate in polymorphic loading by default using the `mapper.polymorphic_load` parameter:

```
class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    engineer_info = Column(String(50))
    __mapper_args__ = {
        'polymorphic_identity':'engineer',
        'polymorphic_load': 'inline'
    }

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_data = Column(String(50))
    __mapper_args__ = {
        'polymorphic_identity':'manager',
        'polymorphic_load': 'inline'
    }
```

Setting the `mapper.polymorphic_load` parameter to the value `"inline"` means that the `Engineer` and `Manager` classes above are part of the "polymorphic load" of the base `Employee` class by default, exactly as though they had been appended to the `mapper.with_polymorphic` list of classes.

Setting with_polymorphic against a query

The `orm.with_polymorphic()` function evolved from a query-level method `Query.with_polymorphic()`. This method has the same purpose as `orm.with_polymorphic()`, except is not as flexible in its usage patterns in that it only applies to the first entity of the `Query`. It then takes effect for all occurences of that entity, so that the entity (and its subclasses) can be referred to directly, rather than using an alias object. For simple cases it might be considered to be more succinct:

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

The `Query.with_polymorphic()` method has a more complicated job than the `orm.with_polymorphic()` function, as it needs to correctly transform entities like `Engineer` and `Manager` appropriately, but not interfere with other entities. If its flexibility is lacking, switch to using `orm.with_polymorphic()`.

Polymorphic Selectin Loading

An alternative to using the `orm.with_polymorphic()` family of functions to "eagerly" load the additional subclasses on an inheritance mapping, primarily when using joined table inheritance, is to use polymorphic "selectin" loading. This is an eager loading feature which works similarly to the selectin_eager_loading feature of relationship loading. Given our example mapping, we can instruct a load of `Employee` to emit an extra SELECT per subclass by using the `orm.selectin_polymorphic()` loader option:

```
from sqlalchemy.orm import selectin_polymorphic

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

When the above query is run, two additional SELECT statements will be emitted:

```
{opensql}query.all()
SELECT
    employee.id AS employee_id,
    employee.name AS employee_name,
    employee.type AS employee_type
FROM employee
()

SELECT
    engineer.id AS engineer_id,
    employee.id AS employee_id,
    employee.type AS employee_type,
    engineer.engineer_name AS engineer_engineer_name
FROM employee JOIN engineer ON employee.id = engineer.id
WHERE employee.id IN (?, ?) ORDER BY employee.id
(1, 2)

SELECT
    manager.id AS manager_id,
    employee.id AS employee_id,
    employee.type AS employee_type,
    manager.manager_name AS manager_manager_name
FROM employee JOIN manager ON employee.id = manager.id
WHERE employee.id IN (?) ORDER BY employee.id
(3,)
```

We can similarly establish the above style of loading to take place by default by specifying the `mapper.polymorphic_load` parameter, using the value `"selectin"` on a per-subclass basis:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity': 'employee',
        'polymorphic_on': type
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    engineer_name = Column(String(30))
```

```
    __mapper_args__ = {
        'polymorphic_load': 'selectin',
        'polymorphic_identity': 'engineer',
    }

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_name = Column(String(30))

    __mapper_args__ = {
        'polymorphic_load': 'selectin',
        'polymorphic_identity': 'manager',
    }
```

Unlike when using `orm.with_polymorphic()`, when using the `orm.selectin_polymorphic()` style of loading, we do **not** have the ability to refer to the `Engineer` or `Manager` entities within our main query as filter, order by, or other criteria, as these entities are not present in the initial query that is used to locate results. However, we can apply loader options that apply towards `Engineer` or `Manager`, which will take effect when the secondary SELECT is emitted. Below we assume `Manager` has an additional relationship `Manager.paperwork`, that we'd like to eagerly load as well. We can use any type of eager loading, such as joined eager loading via the `joinedload()` function:

```
from sqlalchemy.orm import joinedload
from sqlalchemy.orm import selectin_polymorphic

query = session.query(Employee).options(
    selectin_polymorphic(Employee, [Manager, Engineer]),
    joinedload(Manager.paperwork)
)
```

Using the query above, we get three SELECT statements emitted, however the one against `Manager` will be:

```
SELECT
    manager.id AS manager_id,
    employee.id AS employee_id,
    employee.type AS employee_type,
    manager.manager_name AS manager_manager_name,
    paperwork_1.id AS paperwork_1_id,
    paperwork_1.manager_id AS paperwork_1_manager_id,
    paperwork_1.data AS paperwork_1_data
FROM employee JOIN manager ON employee.id = manager.id
LEFT OUTER JOIN paperwork AS paperwork_1
ON manager.id = paperwork_1.manager_id
WHERE employee.id IN (?) ORDER BY employee.id
(3,)
```

Note that selectin polymorphic loading has similar caveats as that of selectin relationship loading; for entities that make use of a composite primary key, the database in use must support tuples with "IN", currently known to work with MySQL and Postgresql.

New in version 1.2.

Warning: The selectin polymorphic loading feature should be considered as **experimental** within early releases of the 1.2 series.

Combining selectin and with_polymorphic

Note: works as of 1.2.0b3

With careful planning, selectin loading can be applied against a hierarchy that itself uses "with_polymorphic". A particular use case is that of using selectin loading to load a joined-inheritance subtable, which then uses "with_polymorphic" to refer to further sub-classes, which may be joined- or single-table inheritanace. If we added a class `VicePresident` that extends `Manager` using single-table inheritance, we could ensure that a load of `Manager` also fully loads `VicePresident` subtypes at the same time:

```
# use "Employee" example from the enclosing section

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_name = Column(String(30))

    __mapper_args__ = {
        'polymorphic_load': 'selectin',
        'polymorphic_identity': 'manager',
    }

class VicePresident(Manager):
    vp_info = Column(String(30))

    __mapper_args__ = {
        "polymorphic_load": "inline",
        "polymorphic_identity": "vp"
    }
```

Above, we add a `vp_info` column to the `manager` table, local to the `VicePresident` subclass. This subclass is linked to the polymorphic identity `"vp"` which refers to rows which have this data. By setting the load style to "inline", it means that a load of `Manager` objects will also ensure that the `vp_info` column is queried for in the same SELECT statement. A query against `Employee` that encounters a `Manager` row would emit similarly to the following:

```
SELECT employee.id AS employee_id, employee.name AS employee_name,
       employee.type AS employee_type
FROM employee
)

SELECT manager.id AS manager_id, employee.id AS employee_id,
       employee.type AS employee_type,
       manager.manager_name AS manager_manager_name,
       manager.vp_info AS manager_vp_info
FROM employee JOIN manager ON employee.id = manager.id
WHERE employee.id IN (?) ORDER BY employee.id
(1,)
```

Combining "selectin" polymorhic loading with query-time `orm.with_polymorphic()` usage is also possible (though this is very outer-space stuff!); assuming the above mappings had no `polymorphic_load` set up, we could get the same result as follows:

```
from sqlalchemy.orm import with_polymorphic, selectin_polymorphic

manager_poly = with_polymorphic(Manager, [VicePresident])

s.query(Employee).options(
    selectin_polymorphic(Employee, [manager_poly])).all()
```

Referring to specific subtypes on relationships

Mapped attributes which correspond to a `relationship()` are used in querying in order to refer to the linkage between two mappings. Common uses for this are to refer to a `relationship()` in `Query.join()` as well as in loader options like `joinedload()`. When using `relationship()` where the target class is an inheritance hierarchy, the API allows that the join, eager load, or other linkage should target a specific subclass, alias, or `orm.with_polymorphic()` alias, of that class hierarchy, rather than the class directly targeted by the `relationship()`.

The `of_type()` method allows the construction of joins along `relationship()` paths while narrowing the criterion to specific derived aliases or 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`:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    employees = relationship("Employee",
                    backref='company')

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    type = Column(String(20))
    company_id = Column(Integer, ForeignKey('company.id'))
    __mapper_args__ = {
        'polymorphic_on':type,
        'polymorphic_identity':'employee',
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    engineer_info = Column(String(50))
    __mapper_args__ = {'polymorphic_identity':'engineer'}

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_data = Column(String(50))
    __mapper_args__ = {'polymorphic_identity':'manager'}
```

When querying from `Company` onto the `Employee` relationship, the `Query.join()` method as well as operators like `PropComparator.any()` and `PropComparator.has()` will create a join from `company` to `employee`, without including `engineer` or `manager` 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 set of columns representing the subclass using the `of_type()` operator:

```
session.query(Company).\
    join(Company.employees.of_type(Engineer)).\
    filter(Engineer.engineer_info=='someinfo')
```

Similarly, to join from `Company` to the polymorphic entity that includes both `Engineer` and `Manager` columns:

```
manager_and_engineer = with_polymorphic(
                            Employee, [Manager, Engineer])

session.query(Company).\
    join(Company.employees.of_type(manager_and_engineer)).\
    filter(
```

```
        or_(
            manager_and_engineer.Engineer.engineer_info == 'someinfo',
            manager_and_engineer.Manager.manager_data == 'somedata'
        )
    )
```

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

```
session.query(Company).\
        filter(
            Company.employees.of_type(Engineer).
                any(Engineer.engineer_info=='someinfo')
            ).all()
```

Eager Loading of Specific or Polymorphic Subtypes

The `joinedload()`, `subqueryload()`, `contains_eager()` and other eagerloader options support paths which make use of `of_type()`. Below, we load `Company` rows while eagerly loading related `Engineer` objects, querying the `employee` and `engineer` tables simultaneously:

```
session.query(Company).\
    options(
        subqueryload(Company.employees.of_type(Engineer)).
        subqueryload(Engineer.machines)
        )
    )
```

As is the case with `Query.join()`, `of_type()` can be used to combine eager loading and `orm.with_polymorphic()`, so that all sub-attributes of all referenced subtypes can be loaded:

```
manager_and_engineer = with_polymorphic(
                            Employee, [Manager, Engineer],
                            flat=True)

session.query(Company).\
    options(
        joinedload(
            Company.employees.of_type(manager_and_engineer)
        )
    )
```

When using `with_polymorphic()` in conjunction with `joinedload()`, the `with_polymorphic()` object must include the `aliased=True` or `flat=True` flag, so that the polymorphic selectable is aliased (an informative error message is raised otherwise). "flat" is an alternate form of aliasing that produces fewer subqueries.

Once `of_type()` is the target of the eager load, that's the entity we would use for subsequent chaining, not the original class or derived class. If we wanted to further eager load a collection on the eager-loaded `Engineer` class, we access this class from the namespace of the `orm.with_polymorphic()` object:

```
session.query(Company).\
    options(
        joinedload(Company.employees.of_type(manager_and_engineer)).\
        subqueryload(manager_and_engineer.Engineer.computers)
        )
    )
```

Loading objects with joined table inheritance

When using joined table inheritance, if we query for a specific subclass that represents a JOIN of two tables such as our `Engineer` example from the inheritance section, the SQL emitted is a join:

```
session.query(Engineer).all()
```

The above query will emit SQL like:

```
{opensql}
SELECT employee.id AS employee_id,
    employee.name AS employee_name, employee.type AS employee_type,
    engineer.name AS engineer_name
FROM employee JOIN engineer
ON employee.id = engineer.id
```

We will then get a collection of `Engineer` objects back, which will contain all columns from `employee` and `engineer` loaded.

However, when emitting a `Query` against a base class, the behavior is to load only from the base table:

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

Above, the default behavior would be to SELECT only from the `employee` table and not from any "sub" tables (`engineer` and `manager`, in our previous examples):

```
{opensql}
SELECT employee.id AS employee_id,
    employee.name AS employee_name, employee.type AS employee_type
FROM employee
[]
```

After a collection of `Employee` objects has been returned from the query, and as attributes are requested from those `Employee` objects which are represented in either the `engineer` or `manager` 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:

```
{opensql}
SELECT manager.id AS manager_id,
    manager.manager_data AS manager_manager_data
FROM manager
WHERE ? = manager.id
[5]
SELECT engineer.id AS engineer_id,
    engineer.engineer_info AS engineer_engineer_info
FROM engineer
WHERE ? = engineer.id
[2]
```

The `orm.with_polymorphic()` function and related configuration options allow us to instead emit a JOIN up front which will conditionally load against `employee`, `engineer`, or `manager`, very much like joined eager loading works for relationships, removing the necessity for a second per-entity load:

```
from sqlalchemy.orm import with_polymorphic

eng_plus_manager = with_polymorphic(Employee, [Engineer, Manager])

query = session.query(eng_plus_manager)
```

The above produces a query which joins the `employee` table to both the `engineer` and `manager` tables like the following:

```
query.all()
{opensql}
SELECT employee.id AS employee_id,
    engineer.id AS engineer_id,
    manager.id AS manager_id,
    employee.name AS employee_name,
    employee.type AS employee_type,
    engineer.engineer_info AS engineer_engineer_info,
    manager.manager_data AS manager_manager_data
FROM employee
    LEFT OUTER JOIN engineer
    ON employee.id = engineer.id
    LEFT OUTER JOIN manager
    ON employee.id = manager.id
[]
```

The section with_polymorphic discusses the `orm.with_polymorphic()` function and its configurational variants.

See also:

with_polymorphic

Loading objects with single table inheritance

In modern Declarative, single inheritance mappings produce `Column` objects that are mapped only to a subclass, and not available from the superclass, even though they are present on the same table. In our example from single_inheritance, the `Manager` mapping for example had a `Column` specified:

```
class Manager(Employee):
    manager_data = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'manager'
    }
```

Above, there would be no `Employee.manager_data` attribute, even though the `employee` table has a `manager_data` column. A query against `Manager` will include this column in the query, as well as an IN clause to limit rows only to `Manager` objects:

```
session.query(Manager).all()
{opensql}
SELECT
    employee.id AS employee_id,
    employee.name AS employee_name,
    employee.type AS employee_type,
    employee.manager_data AS employee_manager_data
FROM employee
WHERE employee.type IN (?)

('manager',)
```

However, in a similar way to that of joined table inheritance, a query against `Employee` will only query for columns mapped to `Employee`:

```
session.query(Employee).all()
{opensql}
SELECT employee.id AS employee_id,
    employee.name AS employee_name,
    employee.type AS employee_type
FROM employee
```

If we get back an instance of `Manager` from our result, accessing additional columns only mapped to `Manager` emits a lazy load for those columns, in a similar way to joined inheritance:

```
SELECT employee.manager_data AS employee_manager_data
FROM employee
WHERE employee.id = ? AND employee.type IN (?)
```

The `orm.with_polymorphic()` function serves a similar role as joined inheritance in the case of single inheritance; it allows both for eager loading of subclass attributes as well as specification of subclasses in a query, just without the overhead of using OUTER JOIN:

```
employee_poly = with_polymorphic(Employee, '*')

q = session.query(employee_poly).filter(
    or_(
        employee_poly.name == 'a',
        employee_poly.Manager.manager_data == 'b'
    )
)
```

Above, our query remains against a single table however we can refer to the columns present in `Manager` or `Engineer` using the "polymorphic" namespace. Since we specified `"*"` for the entities, both `Engineer` and `Manager` will be loaded at once. SQL emitted would be:

```
q.all()
{opensql}
SELECT
    employee.id AS employee_id, employee.name AS employee_name,
    employee.type AS employee_type,
    employee.manager_data AS employee_manager_data,
    employee.engineer_info AS employee_engineer_info
FROM employee
WHERE employee.name = :name_1
OR employee.manager_data = :manager_data_1
```

Inheritance Loading API

sqlalchemy.orm.**with_polymorphic**(*base*, *classes*, *selectable=False*, *flat=False*, *polymorphic_on=None*, *aliased=False*, *innerjoin=False*, *_use_mapper_path=False*, *_existing_alias=None*)

Produce an `AliasedClass` construct which specifies columns for descendant mappers of the given base.

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.

See also:

with_polymorphic - full discussion of `orm.with_polymorphic()`.

Parameters

- **base** – Base class to be aliased.
- **classes** – a single class or mapper, or list of class/mappers, which inherit from the base class. Alternatively, it may also be the string `'*'`, in which case all descending mapped classes will be added to the FROM clause.
- **aliased** – when True, the selectable will be wrapped in an alias, that is `(SELECT * FROM <fromclauses>) AS anon_1`. This can be important when using the with_polymorphic() to create the target of a JOIN on a backend that does not support parenthesized joins, such as SQLite and older versions of MySQL.

- **flat** – Boolean, will be passed through to the `FromClause.alias()` call so that aliases of `Join` objects don't include an enclosing SELECT. This can lead to more efficient queries in many circumstances. A JOIN against a nested JOIN will be rewritten as a JOIN against an aliased SELECT subquery on backends that don't support this syntax.

  Setting `flat` to `True` implies the `aliased` flag is also `True`.

  New in version 0.9.0.

  See also:

  `Join.alias()`

- **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 classes 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 mapped class. Otherwise, the unaccounted mapped columns will result in their table being appended directly to the FROM clause which will usually lead to incorrect results.

- **polymorphic_on** – a column to be used as the "discriminator" column for the given selectable. If not given, the polymorphic_on attribute of the base classes' mapper will be used, if any. This is useful for mappings that don't have polymorphic loading behavior by default.

- **innerjoin** – if True, an INNER JOIN will be used. This should only be specified if querying for one specific subtype only

`sqlalchemy.orm.selectin_polymorphic`(*base_cls*, *classes*)

Indicate an eager load should take place for all attributes specific to a subclass.

This uses an additional SELECT with IN against all matched primary key values, and is the per-query analogue to the `"selectin"` setting on the `mapper.polymorphic_load` parameter.

New in version 1.2.

See also:

inheritance_polymorphic_load

2.4.4 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, there is an event hook known as `InstanceEvents.load()` which can achieve this; it is also available via a class-specific decorator called `orm.reconstructor()`. When using `orm.reconstructor()`, the mapper will invoke the decorated method with no arguments every time it loads or reconstructs an instance of the class. This is useful for recreating transient properties that are normally assigned in `__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 = []
```

Above, when `obj = MyMappedClass()` is executed, the `__init__` constructor is invoked normally and the `data` argument is required. When instances are loaded during a `Query` operation as in `query(MyMappedClass).one()`, `init_on_load` is called.

Any method may be tagged as the `orm.reconstructor()`, even the `__init__` method itself. It is invoked after all immediate column-level attributes are loaded as well as after eagerly-loaded scalar relationships. Eagerly loaded collections may be only partially populated or not populated at all, depending on the kind of eager loading used.

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.

`orm.reconstructor()` is a shortcut into a larger system of "instance level" events, which can be subscribed to using the event API - see `InstanceEvents` for the full API description of these events.

`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.

See also:

mapping_constructors

`InstanceEvents.load()`

2.4.5 Query API

The Query Object

`Query` is produced in terms of a given `Session`, using the `query()` method:

```
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`, and in less common cases by instantiating the `Query` directly and associating with a `Session` using the `Query.with_session()` method.

For a full walkthrough of `Query` usage, see the ormtutorial_toplevel.

`add_column`(*column*)
Add a column expression to the list of result columns to be returned.

Pending deprecation: `add_column()` will be superseded 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.

Analogous to `sqlalchemy.sql.expression.SelectBase.as_scalar()`.

New in version 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.column_descriptions

# would return:
[
    {
        'name':'User',
        'type':User,
        'aliased':False,
        'expr':User,
        'entity': User
    },
    {
        'name':'id',
        'type':Integer(),
        'aliased':False,
        'expr':User.id,
        'entity': User
    },
    {
        'name':'user2',
        'type':User,
        'aliased':True,
        'expr':user_alias,
        'entity': 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.

This generates the SQL for this Query as follows:

```
SELECT count(1) AS count_1 FROM (
    SELECT <rest of query follows...>
) AS anon_1
```

Changed in version 0.7: The above scheme is newly refined as of 0.7b3.

For fine grained control over specific columns to count, to skip the usage of a subquery or otherwise control of the FROM clause, or to use other aggregate functions, use `func` expressions in conjunction with `query()`, i.e.:

```
from sqlalchemy import func

# count User records, without
# using a subquery.
session.query(func.count(User.id))

# return count of user "id" grouped
# by "name"
session.query(func.count(User.id)).\
        group_by(User.name)

from sqlalchemy import distinct

# count distinct "name" values
session.query(func.count(distinct(User.name)))
```

`cte`(*name=None*, *recursive=False*)

Return the full SELECT statement represented by this `Query` represented as a common table expression (CTE).

Parameters and usage are the same as those of the `SelectBase.cte()` method; see that method for further details.

Here is the PostgreSQL WITH RECURSIVE example. Note that, in this example, the `included_parts` cte and the `incl_alias` alias of it are Core selectables, which means the columns are accessed via the `.c.` attribute. The `parts_alias` object is an `orm.aliased()` instance of the `Part` entity, so column-mapped attributes are available directly:

```
from sqlalchemy.orm import aliased

class Part(Base):
    __tablename__ = 'part'
    part = Column(String, primary_key=True)
    sub_part = Column(String, primary_key=True)
    quantity = Column(Integer)
```

```
included_parts = session.query(
                Part.sub_part,
                Part.part,
                Part.quantity).\
                    filter(Part.part=="our part").\
                    cte(name="included_parts", recursive=True)

incl_alias = aliased(included_parts, name="pr")
parts_alias = aliased(Part, name="p")
included_parts = included_parts.union_all(
    session.query(
        parts_alias.sub_part,
        parts_alias.part,
        parts_alias.quantity).\
            filter(parts_alias.part==incl_alias.c.sub_part)
    )

q = session.query(
        included_parts.c.sub_part,
        func.sum(included_parts.c.quantity).
            label('total_quantity')
    ).\
    group_by(included_parts.c.sub_part)
```

See also:

`HasCTE.cte()`

delete(*synchronize_session='evaluate'*)

Perform a bulk delete query.

Deletes rows matched by this query from the database.

E.g.:

```
sess.query(User).filter(User.age == 25).\
    delete(synchronize_session=False)

sess.query(User).filter(User.age == 25).\
    delete(synchronize_session='evaluate')
```

Warning: The `Query.delete()` method is a "bulk" operation, which bypasses ORM unit-of-work automation in favor of greater performance. **Please read all caveats and warnings below.**

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.

The expression evaluator currently doesn't account for differing string collations between the database and Python.

Returns the count of rows matched as returned by the database's "row count" feature.

Warning: Additional Caveats for bulk query deletes

- This method does **not work for joined inheritance mappings**, since the **multiple table deletes are not supported by SQL** as well as that the **join condition of an inheritance mapper is not automatically rendered**. Care must be taken in any multiple-table delete to first accommodate via some other means how the related table will be deleted, as well as to explicitly include the joining condition between those tables, even in mappings where this is normally automatic. E.g. if a class `Engineer` subclasses `Employee`, a DELETE against the `Employee` table would look like:

```
session.query(Engineer).\
    filter(Engineer.id == Employee.id).\
    filter(Employee.name == 'dilbert').\
    delete()
```

  However the above SQL will not delete from the Engineer table, unless an ON DELETE CASCADE rule is established in the database to handle it.

  Short story, **do not use this method for joined inheritance mappings unless you have taken the additional steps to make this feasible**.

- The polymorphic identity WHERE criteria is **not** included for single- or joined- table updates - this must be added **manually** even for single table inheritance.

- The method does **not** offer in-Python cascading of relationships - it is assumed that ON DELETE CASCADE/SET NULL/etc. is configured for any foreign key references which require it, otherwise the database may emit an integrity violation if foreign key references are being enforced.

  After the DELETE, dependent objects in the `Session` which were impacted by an ON DELETE may not contain the current state, or may have been deleted. This issue is resolved once the `Session` is expired, which normally occurs upon `Session.commit()` or can be forced by using `Session.expire_all()`. Accessing an expired object whose row has been deleted will invoke a SELECT to locate the row; when the row is not found, an `ObjectDeletedError` is raised.

- The `'fetch'` strategy results in an additional SELECT statement emitted and will significantly reduce performance.

- The `'evaluate'` strategy performs a scan of all matching objects within the `Session`; if the contents of the `Session` are expired, such as via a proceeding `Session.commit()` call, **this will result in SELECT queries emitted for every matching object**.

- The `MapperEvents.before_delete()` and `MapperEvents.after_delete()` events **are not invoked** from this method. Instead, the `SessionEvents.after_bulk_delete()` method is provided to act upon a mass DELETE of entity rows.

See also:

`Query.update()`

inserts_and_updates - Core SQL tutorial

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

Note: The `distinct()` call includes logic that will automatically add columns from the ORDER BY of the query to the columns clause of the SELECT statement, to satisfy the common need of the database backend that ORDER BY columns be part of the SELECT list when DISTINCT is used. These columns *are not* added to the list of columns actually fetched by the `Query`, however, so would not affect results. The columns are passed through when using the `Query.statement` accessor, however.

Parameters `*expr` – optional column expressions. When present, the PostgreSQL dialect will render a `DISTINCT ON (<expressions>>)` construct.

`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, or when using `Query.yield_per()`.

`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 `Connection.execution_options()`.

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

`exists`()
A convenience method that turns a query into an EXISTS subquery of the form EXISTS (SELECT 1 FROM … WHERE …).

e.g.:

```
q = session.query(User).filter(User.name == 'fred')
session.query(q.exists())
```

Producing SQL similar to:

```
SELECT EXISTS (
    SELECT 1 FROM users WHERE users.name = :name_1
) AS anon_1
```

The EXISTS construct is usually used in the WHERE clause:

```
session.query(User.id).filter(q.exists()).scalar()
```

Note that some databases such as SQL Server don't allow an EXISTS expression to be present in the columns clause of a SELECT. To select a simple boolean value based on the exists as a WHERE, use `literal()`:

```
from sqlalchemy import literal

session.query(literal(True)).filter(q.exists()).scalar()
```

New in version 0.8.1.

`filter`(**criterion*)

apply the given filtering criterion to a copy of this `Query`, using SQL expressions.

e.g.:

```
session.query(MyClass).filter(MyClass.name == 'some name')
```

Multiple criteria may be specified as comma separated; the effect is that they will be joined together using the `and_()` function:

```
session.query(MyClass).\
    filter(MyClass.name == 'some name', MyClass.id > 5)
```

The criterion is any SQL expression object applicable to the WHERE clause of a select. String expressions are coerced into SQL expression constructs via the `text()` construct.

See also:

`Query.filter_by()` - filter on keyword expressions.

`filter_by`(***kwargs*)

apply the given filtering criterion to a copy of this `Query`, using keyword expressions.

e.g.:

```
session.query(MyClass).filter_by(name = 'some name')
```

Multiple criteria may be specified as comma separated; the effect is that they will be joined together using the `and_()` function:

```
session.query(MyClass).\
    filter_by(name = 'some name', id = 5)
```

The keyword expressions are extracted from the primary entity of the query, or the last entity that was the target of a call to `Query.join()`.

See also:

`Query.filter()` - filter on SQL expressions.

`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 `Query.first()` results in an execution of the underlying query.

See also:

`Query.one()`

Query.one_or_none()

from_self(**entities*)

return a Query that selects from this Query's SELECT statement.

Query.from_self() essentially turns the SELECT statement into a SELECT of itself. Given a query such as:

```
q = session.query(User).filter(User.name.like('e%'))
```

Given the Query.from_self() version:

```
q = session.query(User).filter(User.name.like('e%')).from_self()
```

This query renders as:

```
SELECT anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1) AS anon_1
```

There are lots of cases where Query.from_self() may be useful. A simple one is where above, we may want to apply a row LIMIT to the set of user objects we query against, and then apply additional joins against that row-limited set:

```
q = session.query(User).filter(User.name.like('e%')).\
    limit(5).from_self().\
    join(User.addresses).filter(Address.email.like('q%'))
```

The above query joins to the Address entity but only against the first five results of the User query:

```
SELECT anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1
 LIMIT :param_1) AS anon_1
JOIN address ON anon_1.user_id = address.user_id
WHERE address.email LIKE :email_1
```

Automatic Aliasing

Another key behavior of Query.from_self() is that it applies **automatic aliasing** to the entities inside the subquery, when they are referenced on the outside. Above, if we continue to refer to the User entity without any additional aliasing applied to it, those references wil be in terms of the subquery:

```
q = session.query(User).filter(User.name.like('e%')).\
    limit(5).from_self().\
    join(User.addresses).filter(Address.email.like('q%')).\
    order_by(User.name)
```

The ORDER BY against User.name is aliased to be in terms of the inner subquery:

```
SELECT anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1
 LIMIT :param_1) AS anon_1
JOIN address ON anon_1.user_id = address.user_id
WHERE address.email LIKE :email_1 ORDER BY anon_1.user_name
```

The automatic aliasing feature only works in a **limited** way, for simple filters and orderings. More ambitious constructions such as referring to the entity in joins should prefer to use explicit subquery objects, typically making use of the `Query.subquery()` method to produce an explicit subquery object. Always test the structure of queries by viewing the SQL to ensure a particular structure does what's expected!

Changing the Entities

`Query.from_self()` also includes the ability to modify what columns are being queried. In our example, we want `User.id` to be queried by the inner query, so that we can join to the `Address` entity on the outside, but we only wanted the outer query to return the `Address.email` column:

```
q = session.query(User).filter(User.name.like('e%')).\
    limit(5).from_self(Address.email).\
    join(User.addresses).filter(Address.email.like('q%'))
```

yielding:

```
SELECT address.email AS address_email
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1
 LIMIT :param_1) AS anon_1
JOIN address ON anon_1.user_id = address.user_id
WHERE address.email LIKE :email_1
```

Looking out for Inner / Outer Columns

Keep in mind that when referring to columns that originate from inside the subquery, we need to ensure they are present in the columns clause of the subquery itself; this is an ordinary aspect of SQL. For example, if we wanted to load from a joined entity inside the subquery using `contains_eager()`, we need to add those columns. Below illustrates a join of `Address` to `User`, then a subquery, and then we'd like `contains_eager()` to access the `User` columns:

```
q = session.query(Address).join(Address.user).\
    filter(User.name.like('e%'))

q = q.add_entity(User).from_self().\
    options(contains_eager(Address.user))
```

We use `Query.add_entity()` above **before** we call `Query.from_self()` so that the `User` columns are present in the inner subquery, so that they are available to the `contains_eager()` modifier we are using on the outside, producing:

```
SELECT anon_1.address_id AS anon_1_address_id,
       anon_1.address_email AS anon_1_address_email,
       anon_1.address_user_id AS anon_1_address_user_id,
       anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (
    SELECT address.id AS address_id,
    address.email AS address_email,
    address.user_id AS address_user_id,
    "user".id AS user_id,
    "user".name AS user_name
FROM address JOIN "user" ON "user".id = address.user_id
WHERE "user".name LIKE :name_1) AS anon_1
```

If we didn't call `add_entity(User)`, but still asked `contains_eager()` to load the `User` entity, it would be forced to add the table on the outside without the correct join criteria - note the

anon1, "user" phrase at the end:

```
-- incorrect query
SELECT anon_1.address_id AS anon_1_address_id,
       anon_1.address_email AS anon_1_address_email,
       anon_1.address_user_id AS anon_1_address_user_id,
       "user".id AS user_id,
       "user".name AS user_name
FROM (
    SELECT address.id AS address_id,
    address.email AS address_email,
    address.user_id AS address_user_id
FROM address JOIN "user" ON "user".id = address.user_id
WHERE "user".name LIKE :name_1) AS anon_1, "user"
```

Parameters ***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 is typically either a text() or select() construct, and should return the set of columns appropriate to the entity class represented by this Query.

See also:

orm_tutorial_literal_sql - usage examples in the ORM tutorial

get(*ident*)

Return an instance based on the given primary key identifier, or None if not found.

E.g.:

```
my_user = session.query(User).get(5)

some_object = session.query(VersionedFoo).get((5, 10))
```

get() is special in that it provides direct access to the identity map of the owning Session. If the given primary key identifier is present in the local identity map, the object is returned directly from this collection and no SQL is emitted, unless the object has been marked fully expired. If not present, a SELECT is performed in order to locate the object.

get() also will perform a check if the object is present in the identity map and marked as expired - a SELECT is emitted to refresh the object as well as to ensure that the row is still present. If not, ObjectDeletedError is raised.

get() is only used to return a single mapped instance, not multiple instances or individual column constructs, and strictly on a single primary key value. The originating Query must be constructed in this way, i.e. against a single mapped entity, with no additional filtering criterion. Loading options via options() may be applied however, and will be used if the object is not yet locally present.

A lazy-loading, many-to-one attribute configured by relationship(), using a simple foreign-key-to-primary-key criterion, will also use an operation equivalent to get() in order to retrieve the target value from the local identity map before querying the database. See *Relationship Loading Techniques* for further details on relationship loading.

Parameters **ident** – A scalar or tuple value representing the primary key. For a composite primary key, the order of identifiers corresponds in most cases to that of the mapped Table object's primary key columns. For a mapper() that was

given the `primary key` argument during construction, the order of identifiers corresponds to the elements present in this collection.

Returns The object instance, or `None`.

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

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

New in version 1.1: GROUP BY can be cancelled by passing None, in the same way as ORDER BY.

`having`(*criterion*)
apply a HAVING criterion to the query and return the newly resulting `Query`.

`having()` is used in conjunction with `group_by()`.

HAVING criterion makes it possible to use filters on aggregate functions like COUNT, SUM, AVG, MAX, and MIN, eg.:

```
q = session.query(User.id).\
            join(User.addresses).\
            group_by(User.id).\
            having(func.count(Address.id) > 2)
```

`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 SQL JOIN against this `Query` object's criterion and apply generatively, returning the newly resulting `Query`.

Simple Relationship Joins

Consider a mapping between two classes `User` and `Address`, with a relationship `User.addresses` representing a collection of `Address` objects associated with each `User`. The most common usage of `join()` is to create a JOIN along this relationship, using the `User.addresses` attribute as an indicator for how this should occur:

```
q = session.query(User).join(User.addresses)
```

Where above, the call to `join()` along `User.addresses` will result in SQL equivalent to:

```
SELECT user.* FROM user JOIN address ON user.id = address.user_id
```

In the above example we refer to `User.addresses` as passed to `join()` as the *on clause*, that is, it indicates how the "ON" portion of the JOIN should be constructed. For a single-entity

query such as the one above (i.e. we start by selecting only from `User` and nothing else), the relationship can also be specified by its string name:

```
q = session.query(User).join("addresses")
```

`join()` can also accommodate multiple "on clause" arguments to produce a chain of joins, such as below where a join across four related entities is constructed:

```
q = session.query(User).join("orders", "items", "keywords")
```

The above would be shorthand for three separate calls to `join()`, each using an explicit attribute to indicate the source entity:

```
q = session.query(User).\
        join(User.orders).\
        join(Order.items).\
        join(Item.keywords)
```

Joins to a Target Entity or Selectable

A second form of `join()` allows any mapped entity or core selectable construct as a target. In this usage, `join()` will attempt to create a JOIN along the natural foreign key relationship between two entities:

```
q = session.query(User).join(Address)
```

The above calling form of `join()` will raise an error if either there are no foreign keys between the two entities, or if there are multiple foreign key linkages between them. In the above calling form, `join()` is called upon to create the "on clause" automatically for us. The target can be any mapped entity or selectable, such as a `Table`:

```
q = session.query(User).join(addresses_table)
```

Joins to a Target with an ON Clause

The third calling form allows both the target entity as well as the ON clause to be passed explicitly. Suppose for example we wanted to join to `Address` twice, using an alias the second time. We use `aliased()` to create a distinct alias of `Address`, and join to it using the `target, onclause` form, so that the alias can be specified explicitly as the target along with the relationship to instruct how the ON clause should proceed:

```
a_alias = aliased(Address)

q = session.query(User).\
        join(User.addresses).\
        join(a_alias, User.addresses).\
        filter(Address.email_address=='ed@foo.com').\
        filter(a_alias.email_address=='ed@bar.com')
```

Where above, the generated SQL would be similar to:

```
SELECT user.* FROM user
    JOIN address ON user.id = address.user_id
    JOIN address AS address_1 ON user.id=address_1.user_id
    WHERE address.email_address = :email_address_1
    AND address_1.email_address = :email_address_2
```

The two-argument calling form of `join()` also allows us to construct arbitrary joins with SQL-oriented "on clause" expressions, not relying upon configured relationships at all. Any SQL expression can be passed as the ON clause when using the two-argument form, which should refer to the target entity in some way as well as an applicable source entity:

```
q = session.query(User).join(Address, User.id==Address.user_id)
```

Changed in version 0.7: In SQLAlchemy 0.6 and earlier, the two argument form of `join()` requires the usage of a tuple: `query(User).join((Address, User.id==Address.user_id))`. This calling form is accepted in 0.7 and further, though is not necessary unless multiple join conditions are passed to a single `join()` call, which itself is also not generally necessary as it is now equivalent to multiple calls (this wasn't always the case).

Advanced Join Targeting and Adaption

There is a lot of flexibility in what the "target" can be when using `join()`. As noted previously, it also accepts `Table` constructs and other selectables such as `alias()` and `select()` constructs, with either the one or two-argument forms:

```
addresses_q = select([Address.user_id]).\
            where(Address.email_address.endswith("@bar.com")).\
            alias()

q = session.query(User).\
            join(addresses_q, addresses_q.c.user_id==User.id)
```

`join()` also features the ability to *adapt* a `relationship()` -driven ON clause to the target selectable. Below we construct a JOIN from `User` to a subquery against `Address`, allowing the relationship denoted by `User.addresses` to *adapt* itself to the altered target:

```
address_subq = session.query(Address).\
                filter(Address.email_address == 'ed@foo.com').\
                subquery()

q = session.query(User).join(address_subq, User.addresses)
```

Producing SQL similar to:

```
SELECT user.* FROM user
    JOIN (
        SELECT address.id AS id,
                address.user_id AS user_id,
                address.email_address AS email_address
        FROM address
        WHERE address.email_address = :email_address_1
    ) AS anon_1 ON user.id = anon_1.user_id
```

The above form allows one to fall back onto an explicit ON clause at any time:

```
q = session.query(User).\
        join(address_subq, User.id==address_subq.c.user_id)
```

Controlling what to Join From

While `join()` exclusively deals with the "right" side of the JOIN, we can also control the "left" side, in those cases where it's needed, using `select_from()`. Below we construct a query against `Address` but can still make usage of `User.addresses` as our ON clause by instructing the `Query` to select first from the `User` entity:

```
q = session.query(Address).select_from(User).\
                join(User.addresses).\
                filter(User.name == 'ed')
```

Which will produce SQL similar to:

```
SELECT address.* FROM user
    JOIN address ON user.id=address.user_id
    WHERE user.name = :name_1
```

Constructing Aliases Anonymously

`join()` can construct anonymous aliases using the `aliased=True` flag. This feature is useful when a query is being joined algorithmically, such as when querying self-referentially to an arbitrary depth:

```
q = session.query(Node).\
        join("children", "children", aliased=True)
```

When `aliased=True` is used, the actual "alias" construct is not explicitly available. To work with it, methods such as `Query.filter()` will adapt the incoming entity to the last join point:

```
q = session.query(Node).\
        join("children", "children", aliased=True).\
        filter(Node.name == 'grandchild 1')
```

When using automatic aliasing, the `from_joinpoint=True` argument can allow a multi-node join to be broken into multiple calls to `join()`, so that each path along the way can be further filtered:

```
q = session.query(Node).\
        join("children", aliased=True).\
        filter(Node.name='child 1').\
        join("children", aliased=True, from_joinpoint=True).\
        filter(Node.name == 'grandchild 1')
```

The filtering aliases above can then be reset back to the original `Node` entity using `reset_joinpoint()`:

```
q = session.query(Node).\
        join("children", "children", aliased=True).\
        filter(Node.name == 'grandchild 1').\
        reset_joinpoint().\
        filter(Node.name == 'parent 1)
```

For an example of `aliased=True`, see the distribution example examples_xmlpersistence which illustrates an XPath-like query system using algorithmic joins.

Parameters

- **`*props`** – A collection of one or more join conditions, each consisting of a relationship-bound attribute or string relationship name representing an "on clause", or a single target entity, or a tuple in the form of `(target, onclause)`. A special two-argument calling form of the form `target, onclause` is also accepted.
- **`aliased=False`** – If True, indicate that the JOIN target should be anonymously aliased. Subsequent calls to `filter()` and similar will adapt the incoming criterion to the target alias, until `reset_joinpoint()` is called.
- **`isouter=False`** – If True, the join used will be a left outer join, just as if the `Query.outerjoin()` method were called. This flag is here to maintain consistency with the same flag as accepted by `FromClause.join()` and other Core constructs.

  New in version 1.0.0.
- **`full=False`** – render FULL OUTER JOIN; implies `isouter`.

  New in version 1.1.

- `from_joinpoint=False` – When using `aliased=True`, a setting of True here will cause the join to be from the most recent joined target, rather than starting back from the original FROM clauses of the query.

See also:

ormtutorial_joins in the ORM tutorial.

inheritance_toplevel for details on how `join()` is used for inheritance relationships.

`orm.join()` - a standalone ORM-level join function, used internally by `Query.join()`, which in previous SQLAlchemy versions was the primary ORM-level joining interface.

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

Analogous to `sqlalchemy.sql.expression.SelectBase.label()`.

New in version 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` object'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()`.

For an example of how `merge_result()` is used, see the source code for the example examples_caching, where `merge_result()` is used to efficiently restore state from a cache back into a target `Session`.

`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 returns only scalar values as opposed to full identity-mapped entities.

Calling `one()` results in an execution of the underlying query.

See also:

`Query.first()`

`Query.one_or_none()`

`one_or_none`()
Return at most one result or raise an exception.

Returns `None` 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 returns only scalar values as opposed to full identity-mapped entities.

Calling `Query.one_or_none()` results in an execution of the underlying query.

New in version 1.0.9: Added `Query.one_or_none()`

See also:

`Query.first()`

`Query.one()`

`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 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, passing False will reset ORDER BY and additionally re-allow default mapper.order_by to take place. Note mapper.order_by is deprecated.

`outerjoin`(**props*, ***kwargs*)
Create a left outer join against this `Query` object's criterion and apply generatively, returning 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.

`prefix_with`(**prefixes*)
Apply the prefixes to the query and return the newly resulting `Query`.

Parameters **`*prefixes`** – optional prefixes, typically strings, not using any commas. In particular is useful for MySQL keywords.

e.g.:

```
query = sess.query(User.name).\
    prefix_with('HIGH_PRIORITY').\
    prefix_with('SQL_SMALL_RESULT', 'ALL')
```

Would render:

```
SELECT HIGH_PRIORITY SQL_SMALL_RESULT ALL users.name AS users_name
FROM users
```

New in version 0.7.7.

See also:

`HasPrefixes.prefix_with()`

`reset_joinpoint()`
Return a new `Query`, where the "join point" has been reset back to the base FROM entities of the query.

This method is usually used in conjunction with the `aliased=True` feature of the `join()` method. See the example in `join()` for how this is used.

`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_entity_from`(*from_obj*)
Set the FROM clause of this `Query` to a core selectable, applying it as a replacement FROM clause for corresponding mapped entities.

The `Query.select_entity_from()` method supplies an alternative approach to the use case of applying an `aliased()` construct explicitly throughout a query. Instead of referring to the `aliased()` construct explicitly, `Query.select_entity_from()` automatically *adapts* all occurences of the entity to the target selectable.

Given a case for `aliased()` such as selecting `User` objects from a SELECT statement:

```
select_stmt = select([User]).where(User.id == 7)
user_alias = aliased(User, select_stmt)

q = session.query(user_alias).\
    filter(user_alias.name == 'ed')
```

Above, we apply the `user_alias` object explicitly throughout the query. When it's not feasible for `user_alias` to be referenced explicitly in many places, `Query.select_entity_from()` may be used at the start of the query to adapt the existing `User` entity:

```
q = session.query(User).\
    select_entity_from(select_stmt).\
    filter(User.name == 'ed')
```

Above, the generated SQL will show that the `User` entity is adapted to our statement, even in the case of the WHERE clause:

```
SELECT anon_1.id AS anon_1_id, anon_1.name AS anon_1_name
FROM (SELECT "user".id AS id, "user".name AS name
FROM "user"
WHERE "user".id = :id_1) AS anon_1
WHERE anon_1.name = :name_1
```

The `Query.select_entity_from()` method is similar to the `Query.select_from()` method, in that it sets the FROM clause of the query. The difference is that it additionally applies adaptation to the other parts of the query that refer to the primary entity. If above we had used `Query.select_from()` instead, the SQL generated would have been:

```
-- uses plain select_from(), not select_entity_from()
SELECT "user".id AS user_id, "user".name AS user_name
FROM "user", (SELECT "user".id AS id, "user".name AS name
FROM "user"
WHERE "user".id = :id_1) AS anon_1
WHERE "user".name = :name_1
```

To supply textual SQL to the `Query.select_entity_from()` method, we can make use of the `text()` construct. However, the `text()` construct needs to be aligned with the columns of our entity, which is achieved by making use of the `TextClause.columns()` method:

```
text_stmt = text("select id, name from user").columns(
    User.id, User.name)
q = session.query(User).select_entity_from(text_stmt)
```

`Query.select_entity_from()` itself accepts an `aliased()` object, so that the special options of `aliased()` such as `aliased.adapt_on_names` may be used within the scope of the `Query.select_entity_from()` method's adaptation services. Suppose a view `user_view` also returns rows from `user`. If we reflect this view into a `Table`, this view has no relationship to the `Table` to which we are mapped, however we can use name matching to select from it:

```
user_view = Table('user_view', metadata,
                  autoload_with=engine)
user_view_alias = aliased(
    User, user_view, adapt_on_names=True)
q = session.query(User).\
    select_entity_from(user_view_alias).\
    order_by(User.name)
```

Changed in version 1.1.7: The `Query.select_entity_from()` method now accepts an `aliased()` object as an alternative to a `FromClause` object.

Parameters `from_obj` – a `FromClause` object that will replace the FROM clause of this `Query`. It also may be an instance of `aliased()`.

See also:

`Query.select_from()`

`select_from(*from_obj)`

Set the FROM clause of this `Query` explicitly.

`Query.select_from()` is often used in conjunction with `Query.join()` in order to control which entity is selected from on the "left" side of the join.

The entity or selectable object here effectively replaces the "left edge" of any calls to `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.

A typical example:

```
q = session.query(Address).select_from(User).\
    join(User.addresses).\
    filter(User.name == 'ed')
```

Which produces SQL equivalent to:

```
SELECT address.* FROM user
JOIN address ON user.id=address.user_id
WHERE user.name = :name_1
```

Parameters `*from_obj` – collection of one or more entities to apply to the FROM clause. Entities can be mapped classes, `AliasedClass` objects, `Mapper` objects

as well as core FromClause elements like subqueries.

Changed in version 0.9: This method no longer applies the given FROM object to be the selectable from which matching entities select from; the select_entity_from() method now accomplishes this. See that method for a description of this behavior.

See also:

join()

Query.select_entity_from()

selectable

Return the Select object emitted by this Query.

Used for inspect() compatibility, this is equivalent to:

```
query.enable_eagerloads(False).with_labels().statement
```

slice(*start*, *stop*)

Computes the "slice" of the Query represented by the given indices and returns the resulting Query.

The start and stop indices behave like the argument to Python's built-in range() function. This method provides an alternative to using LIMIT/OFFSET to get a slice of the query.

For example,

```
session.query(User).order_by(User.id).slice(1, 3)
```

renders as

```
SELECT users.id AS users_id,
       users.name AS users_name
FROM users ORDER BY users.id
LIMIT ? OFFSET ?
(2, 1)
```

See also:

Query.limit()

Query.offset()

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(*name=None*, *with_labels=False*, *reduce_columns=False*)

return the full SELECT statement represented by this Query, embedded within an Alias.

Eager JOIN generation within the query is disabled.

Parameters

- **name** – string name to be assigned as the alias; this is passed through to FromClause.alias(). If None, a name will be deterministically generated at compile time.
- **with_labels** – if True, with_labels() will be called on the Query first to apply table-qualified labels to all columns.
- **reduce_columns** – if True, Select.reduce_columns() will be called on the resulting select() construct, to remove same-named columns where one also refers to the other via foreign key or WHERE clause equivalence.

Changed in version 0.8: the `with_labels` and `reduce_columns` keyword arguments were added.

suffix_with(**suffixes*)

Apply the suffix to the query and return the newly resulting `Query`.

Parameters ***suffixes** – optional suffixes, typically strings, not using any commas.

New in version 1.0.0.

See also:

`Query.prefix_with()`

`HasSuffixes.suffix_with()`

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)
```

Note that many database backends do not allow ORDER BY to be rendered on a query called within UNION, EXCEPT, etc. To disable all ORDER BY clauses including those configured on mappers, issue `query.order_by(None)` - the resulting `Query` object will not render ORDER BY within its SELECT statement.

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'*, *update_args=None*)

Perform a bulk update query.

Updates rows matched by this query in the database.

E.g.:

```
sess.query(User).filter(User.age == 25).\
    update({User.age: User.age - 10}, synchronize_session=False)

sess.query(User).filter(User.age == 25).\
    update({"age": User.age - 10}, synchronize_session='evaluate')
```

Warning: The `Query.update()` method is a "bulk" operation, which bypasses ORM unit-of-work automation in favor of greater performance. **Please read all caveats and warnings below.**

Parameters

- **`values`** – a dictionary with attributes names, or alternatively mapped attributes or SQL expressions, as keys, and literal values or sql expressions as values. If parameter-ordered mode is desired, the values can be passed as a list of 2-tuples; this requires that the `preserve_parameter_order` flag is passed to the `Query.update.update_args` dictionary as well.

  Changed in version 1.0.0: - string names in the values dictionary are now resolved against the mapped entity; previously, these strings were passed as literal column names with no mapper-level translation.

- **`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.

- **`update_args`** – Optional dictionary, if present will be passed to the underlying `update()` construct as the `**kw` for the object. May be used to pass dialect-specific arguments such as `mysql_limit`, as well as other special arguments such as `preserve_parameter_order`.

  New in version 1.0.0.

Returns the count of rows matched as returned by the database's "row count" feature.

Warning: Additional Caveats for bulk query updates

- 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, otherwise the database may emit an integrity violation if foreign key references are being enforced.

  After the UPDATE, dependent objects in the `Session` which were impacted by an ON UPDATE CASCADE may not contain the current state; this issue is resolved once the `Session` is expired, which normally occurs upon `Session.commit()` or can be forced by using `Session.expire_all()`.

- The `'fetch'` strategy results in an additional SELECT statement emitted and will significantly reduce performance.

- The `'evaluate'` strategy performs a scan of all matching objects within the `Session`; if the contents of the `Session` are expired, such as via a proceeding `Session.commit()` call, **this will result in SELECT queries emitted for every matching object**.
- The method supports multiple table updates, as detailed in multi_table_updates, and this behavior does extend to support updates of joined-inheritance and other multiple table mappings. However, the **join condition of an inheritance mapper is not automatically rendered**. Care must be taken in any multiple-table update to explicitly include the joining condition between those tables, even in mappings where this is normally automatic. E.g. if a class `Engineer` subclasses `Employee`, an UPDATE of the `Engineer` local table using criteria against the `Employee` local table might look like:

```
session.query(Engineer).\
    filter(Engineer.id == Employee.id).\
    filter(Employee.name == 'dilbert').\
    update({"engineer_type": "programmer"})
```

- The polymorphic identity WHERE criteria is **not** included for single- or joined- table updates - this must be added **manually**, even for single table inheritance.
- The `MapperEvents.before_update()` and `MapperEvents.after_update()` events **are not invoked from this method**. Instead, the `SessionEvents.after_bulk_update()` method is provided to act upon a mass UPDATE of entity rows.

See also:

`Query.delete()`

inserts_and_updates - Core SQL tutorial

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 version 0.6.5.

with_for_update(*read=False*, *nowait=False*, *of=None*, *skip_locked=False*, *key_share=False*)
return a new `Query` with the specified options for the `FOR UPDATE` clause.

The behavior of this method is identical to that of `SelectBase.with_for_update()`. When called with no arguments, the resulting `SELECT` statement will have a `FOR UPDATE` clause appended. When additional arguments are specified, backend-specific options such as `FOR UPDATE NOWAIT` or `LOCK IN SHARE MODE` can take effect.

E.g.:

```
q = sess.query(User).with_for_update(nowait=True, of=User)
```

The above query on a PostgreSQL backend will render like:

```
SELECT users.id AS users_id FROM users FOR UPDATE OF users NOWAIT
```

New in version 0.9.0: `Query.with_for_update()` supersedes the `Query.with_lockmode()` method.

See also:

`GenerativeSelect.with_for_update()` - Core level method with full argument and behavioral description.

with_hint(*selectable*, *text*, *dialect_name='*'*)
Add an indexing or other executional context 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.

See also:

`Query.with_statement_hint()`

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.

Note: The `Query.with_labels()` method *only* applies the output of `Query.statement`, and *not* to any of the result-row invoking systems of `Query` itself, e.g. `Query.first()`, `Query.all()`, etc. To execute a query using `Query.with_labels()`, invoke the `Query.statement` using `Session.execute()`:

```
result = session.execute(query.with_labels().statement)
```

---

with_lockmode(*mode*)
Return a new `Query` object with the specified "locking mode", which essentially refers to the `FOR UPDATE` clause.

Deprecated since version 0.9.0: superseded by `Query.with_for_update()`.

Parameters **mode** – a string representing the desired locking mode. Valid values are:

- `None` - translates to no lockmode

- `'update'` - translates to `FOR UPDATE` (standard SQL, supported by most dialects)
- `'update_nowait'` - translates to `FOR UPDATE NOWAIT` (supported by Oracle, PostgreSQL 8.1 upwards)
- `'read'` - translates to `LOCK IN SHARE MODE` (for MySQL), and `FOR SHARE` (for PostgreSQL)

See also:

`Query.with_for_update()` - improved API for specifying the `FOR UPDATE` clause.

`with_parent`(*instance*, *property=None*, *from_entity=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.

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.
- **`from_entity`** – Entity in which to consider as the left side. This defaults to the "zero" entity of the `Query` itself.

`with_polymorphic`(*cls_or_mappers*, *selectable=None*, *polymorphic_on=None*)

Load columns for inheriting classes.

`Query.with_polymorphic()` applies transformations to the "main" mapped class represented by this `Query`. The "main" mapped class here means the `Query` object's first argument is a full class, i.e. `session.query(SomeClass)`. These transformations allow additional tables to be present in the FROM clause so that columns for a joined-inheritance subclass are available in the query, both for the purposes of load-time efficiency as well as the ability to use these columns at query time.

See the documentation section with_polymorphic for details on how this method is used.

Changed in version 0.8: A new and more flexible function `orm.with_polymorphic()` supersedes `Query.with_polymorphic()`, as it can apply the equivalent functionality to any set of columns or classes in the `Query`, not just the "zero mapper". See that function for a description of arguments.

`with_session`(*session*)

Return a `Query` that will use the given `Session`.

While the `Query` object is normally instantiated using the `Session.query()` method, it is legal to build the `Query` directly without necessarily using a `Session`. Such a `Query` object, or any `Query` already associated with a different `Session`, can produce a new `Query` object associated with a target session using this method:

```
from sqlalchemy.orm import Query

query = Query([MyClass]).filter(MyClass.id == 5)

result = query.with_session(my_session).one()
```

`with_statement_hint`(*text*, *dialect_name='*'*)

add a statement hint to this `Select`.

This method is similar to `Select.with_hint()` except that it does not require an individual table, and instead applies to the statement as a whole.

This feature calls down into `Select.with_statement_hint()`.

New in version 1.0.0.

See also:

`Query.with_hint()`

with_transformation(*fn*)
Return a new `Query` object transformed by the given function.

E.g.:

```
def filter_something(criterion):
    def transform(q):
        return q.filter(criterion)
    return transform

q = q.with_transformation(filter_something(x==5))
```

This allows ad-hoc recipes to be created for `Query` objects. See the example at hybrid_transformers.

New in version 0.7.4.

yield_per(*count*)
Yield only `count` rows at a time.

The purpose of this method is when fetching very large result sets (> 10K rows), to batch results in sub-collections and yield them out partially, so that the Python interpreter doesn't need to declare very large areas of memory which is both time consuming and leads to excessive memory use. The performance from fetching hundreds of thousands of rows can often double when a suitable yield-per setting (e.g. approximately 1000) is used, even with DBAPIs that buffer rows (which are most).

The `Query.yield_per()` method **is not compatible subqueryload eager loading or joinedload eager loading when using collections**. It is potentially compatible with "select in" eager loading, **provided the databse driver supports multiple, independent cursors** (pysqlite and psycopg2 are known to work, MySQL and SQL Server ODBC drivers do not).

Therefore in some cases, it may be helpful to disable eager loads, either unconditionally with `Query.enable_eagerloads()`:

```
q = sess.query(Object).yield_per(100).enable_eagerloads(False)
```

Or more selectively using `lazyload()`; such as with an asterisk to specify the default loader scheme:

```
q = sess.query(Object).yield_per(100).\
    options(lazyload('*'), joinedload(Object.some_related))
```

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 while `yield_per()` will set the `stream_results` execution option to True, currently this is only understood by *psycopg2*, *mysqldb* and *pymysql* dialects which will stream results using server side cursors instead of pre-buffer all rows for this query. Other DBAPIs **pre-buffer all rows** before making them available. The memory use of raw database rows is much less than that of an ORM-mapped object, but should still be taken into consideration when benchmarking.

See also:

`Query.enable_eagerloads()`

ORM-Specific Query Constructs

`sqlalchemy.orm.aliased`(*element*, *alias=None*, *name=None*, *flat=False*, *adapt_on_names=False*)

Produce an alias of the given element, usually an `AliasedClass` instance.

E.g.:

```
my_alias = aliased(MyClass)

session.query(MyClass, my_alias).filter(MyClass.id > my_alias.id)
```

The `aliased()` function is used to create an ad-hoc mapping of a mapped class to a new selectable. By default, a selectable is generated from the normally mapped selectable (typically a `Table`) using the `FromClause.alias()` method. However, `aliased()` can also be used to link the class to a new `select()` statement. Also, the `with_polymorphic()` function is a variant of `aliased()` that is intended to specify a so-called "polymorphic selectable", that corresponds to the union of several joined-inheritance subclasses at once.

For convenience, the `aliased()` function also accepts plain `FromClause` constructs, such as a `Table` or `select()` construct. In those cases, the `FromClause.alias()` method is called on the object and the new `Alias` object returned. The returned `Alias` is not ORM-mapped in this case.

Parameters

- **`element`** – element to be aliased. Is normally a mapped class, but for convenience can also be a `FromClause` element.
- **`alias`** – Optional selectable unit to map the element to. This should normally be a `Alias` object corresponding to the `Table` to which the class is mapped, or to a `select()` construct that is compatible with the mapping. By default, a simple anonymous alias of the mapped table is generated.
- **`name`** – optional string name to use for the alias, if not specified by the `alias` parameter. The name, among other things, forms the attribute name that will be accessible via tuples returned by a `Query` object.
- **`flat`** – Boolean, will be passed through to the `FromClause.alias()` call so that aliases of `Join` objects don't include an enclosing SELECT. This can lead to more efficient queries in many circumstances. A JOIN against a nested JOIN will be rewritten as a JOIN against an aliased SELECT subquery on backends that don't support this syntax.

  New in version 0.9.0.

  See also:

  `Join.alias()`

- **`adapt_on_names`** – if True, more liberal "matching" will be used when mapping the mapped columns of the ORM entity to those of the given selectable - a name-based match will be performed if the given selectable doesn't otherwise

have a column that corresponds to one on the entity. The use case for this is when associating an entity with some derived selectable such as one that uses aggregate functions:

```
class UnitPrice(Base):
    __tablename__ = 'unit_price'
    ...
    unit_id = Column(Integer)
    price = Column(Numeric)

aggregated_unit_price = Session.query(
                            func.sum(UnitPrice.price).label('price')
                        ).group_by(UnitPrice.unit_id).subquery()

aggregated_unit_price = aliased(UnitPrice,
            alias=aggregated_unit_price, adapt_on_names=True)
```

Above, functions on `aggregated_unit_price` which refer to `.price` will return the `func.sum(UnitPrice.price).label('price')` column, as it is matched on the name "price". Ordinarily, the "price" function wouldn't have any "column correspondence" to the actual `UnitPrice.price` column as it is not a proxy of the original.

New in version 0.7.3.

`class sqlalchemy.orm.util.AliasedClass(`*cls*, *alias=None*, *name=None*, *flat=False*, *adapt_on_names=False*, *with_polymorphic_mappers=()*, *with_polymorphic_discriminator=None*, *base_alias=None*, *use_mapper_path=False*, *represents_outer_join=False*`)`

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 `orm.aliased()` function, or alternatively via the `orm.with_polymorphic()` function.

Usage example:

```
# 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)
```

The resulting object is an instance of `AliasedClass`. This object implements an attribute scheme which produces the same attribute and method interface as the original mapped class, allowing `AliasedClass` to be compatible with any attribute technique which works on the original class, including hybrid attributes (see hybrids_toplevel).

The `AliasedClass` can be inspected for its underlying `Mapper`, aliased selectable, and other information using `inspect()`:

```
from sqlalchemy import inspect
my_alias = aliased(MyClass)
insp = inspect(my_alias)
```

The resulting inspection object is an instance of `AliasedInsp`.

See `aliased()` and `with_polymorphic()` for construction argument descriptions.

class sqlalchemy.orm.util.AliasedInsp(*entity*, *mapper*, *selectable*, *name*, *with_polymorphic_mappers*, *polymorphic_on*, *_base_alias*, *_use_mapper_path*, *adapt_on_names*, *represents_outer_join*)

Provide an inspection interface for an `AliasedClass` object.

The `AliasedInsp` object is returned given an `AliasedClass` using the `inspect()` function:

```
from sqlalchemy import inspect
from sqlalchemy.orm import aliased

my_alias = aliased(MyMappedClass)
insp = inspect(my_alias)
```

Attributes on `AliasedInsp` include:

- `entity` - the `AliasedClass` represented.
- `mapper` - the `Mapper` mapping the underlying class.
- `selectable` - the `Alias` construct which ultimately represents an aliased `Table` or `Select` construct.
- `name` - the name of the alias. Also is used as the attribute name when returned in a result tuple from `Query`.
- `with_polymorphic_mappers` - collection of `Mapper` objects indicating all those mappers expressed in the select construct for the `AliasedClass`.
- `polymorphic_on` - an alternate column or SQL expression which will be used as the "discriminator" for a polymorphic load.

See also:

inspection_toplevel

class sqlalchemy.orm.query.Bundle(*name*, **exprs*, ***kw*)

A grouping of SQL expressions that are returned by a `Query` under one namespace.

The `Bundle` essentially allows nesting of the tuple-based results returned by a column-oriented `Query` object. It also is extensible via simple subclassing, where the primary capability to override is that of how the set of expressions should be returned, allowing post-processing as well as custom return types, without involving ORM identity-mapped classes.

New in version 0.9.0.

See also:

bundles

c = None

An alias for `Bundle.columns`.

columns = None

A namespace of SQL expressions referred to by this `Bundle`.

e.g.:

```
bn = Bundle("mybundle", MyClass.x, MyClass.y)

q = sess.query(bn).filter(bn.c.x == 5)
```

Nesting of bundles is also supported:

```
b1 = Bundle("b1",
        Bundle('b2', MyClass.a, MyClass.b),
        Bundle('b3', MyClass.x, MyClass.y)
    )
```

```
q = sess.query(b1).filter(
    b1.c.b2.c.a == 5).filter(b1.c.b3.c.y == 9)
```

See also:

`Bundle.c`

`create_row_processor`(*query*, *procs*, *labels*)
Produce the "row processing" function for this `Bundle`.

May be overridden by subclasses.

See also:

bundles - includes an example of subclassing.

`label`(*name*)
Provide a copy of this `Bundle` passing a new label.

`single_entity = False`
If True, queries for a single Bundle will be returned as a single entity, rather than an element within a keyed tuple.

`class sqlalchemy.util.KeyedTuple`
`tuple` subclass that adds labeled names.

E.g.:

```
>>> k = KeyedTuple([1, 2, 3], labels=["one", "two", "three"])
>>> k.one
1
>>> k.two
2
```

Result rows returned by `Query` that contain multiple ORM entities and/or column expressions make use of this class to return rows.

The `KeyedTuple` exhibits similar behavior to the `collections.namedtuple()` construct provided in the Python standard library, however is architected very differently. Unlike `collections.namedtuple()`, `KeyedTuple` is does not rely on creation of custom subtypes in order to represent a new series of keys, instead each `KeyedTuple` instance receives its list of keys in place. The subtype approach of `collections.namedtuple()` introduces significant complexity and performance overhead, which is not necessary for the `Query` object's use case.

Changed in version 0.8: Compatibility methods with `collections.namedtuple()` have been added including `KeyedTuple._fields` and `KeyedTuple._asdict()`.

See also:

ormtutorial_querying

`_asdict()`
Return the contents of this `KeyedTuple` as a dictionary.

This method provides compatibility with `collections.namedtuple()`, with the exception that the dictionary returned is **not** ordered.

New in version 0.8.

`_fields`
Return a tuple of string key names for this `KeyedTuple`.

This method provides compatibility with `collections.namedtuple()`.

New in version 0.8.

See also:

KeyedTuple.keys()

keys()
Return a list of string key names for this KeyedTuple.

See also:

KeyedTuple._fields

class sqlalchemy.orm.strategy_options.**Load**(*entity*)
Represents loader options which modify the state of a Query in order to affect how various mapped attributes are loaded.

The Load object is in most cases used implicitly behind the scenes when one makes use of a query option like joinedload(), defer(), or similar. However, the Load object can also be used directly, and in some cases can be useful.

To use Load directly, instantiate it with the target mapped class as the argument. This style of usage is useful when dealing with a Query that has multiple entities:

```
myopt = Load(MyClass).joinedload("widgets")
```

The above myopt can now be used with Query.options(), where it will only take effect for the MyClass entity:

```
session.query(MyClass, MyOtherClass).options(myopt)
```

One case where Load is useful as public API is when specifying "wildcard" options that only take effect for a certain class:

```
session.query(Order).options(Load(Order).lazyload('*'))
```

Above, all relationships on Order will be lazy-loaded, but other attributes on those descendant objects will load using their normal loader strategy.

See also:

loading_toplevel

baked_lazyload(*loadopt*, *attr*)
Produce a new Load object with the orm.baked_lazyload() option applied.

See orm.baked_lazyload() for usage examples.

contains_eager(*loadopt*, *attr*, *alias=None*)
Produce a new Load object with the orm.contains_eager() option applied.

See orm.contains_eager() for usage examples.

defaultload(*loadopt*, *attr*)
Produce a new Load object with the orm.defaultload() option applied.

See orm.defaultload() for usage examples.

defer(*loadopt*, *key*)
Produce a new Load object with the orm.defer() option applied.

See orm.defer() for usage examples.

immediateload(*loadopt*, *attr*)
Produce a new Load object with the orm.immediateload() option applied.

See orm.immediateload() for usage examples.

joinedload(*loadopt*, *attr*, *innerjoin=None*)
Produce a new Load object with the orm.joinedload() option applied.

See orm.joinedload() for usage examples.

`lazyload`(*loadopt*, *attr*)
Produce a new `Load` object with the `orm.lazyload()` option applied.

See `orm.lazyload()` for usage examples.

`load_only`(*loadopt*, **attrs*)
Produce a new `Load` object with the `orm.load_only()` option applied.

See `orm.load_only()` for usage examples.

`noload`(*loadopt*, *attr*)
Produce a new `Load` object with the `orm.noload()` option applied.

See `orm.noload()` for usage examples.

`raiseload`(*loadopt*, *attr*, *sql_only=False*)
Produce a new `Load` object with the `orm.raiseload()` option applied.

See `orm.raiseload()` for usage examples.

`selectin_polymorphic`(*loadopt*, *classes*)
Produce a new `Load` object with the `orm.selectin_polymorphic()` option applied.

See `orm.selectin_polymorphic()` for usage examples.

`selectinload`(*loadopt*, *attr*)
Produce a new `Load` object with the `orm.selectinload()` option applied.

See `orm.selectinload()` for usage examples.

`subqueryload`(*loadopt*, *attr*)
Produce a new `Load` object with the `orm.subqueryload()` option applied.

See `orm.subqueryload()` for usage examples.

`undefer`(*loadopt*, *key*)
Produce a new `Load` object with the `orm.undefer()` option applied.

See `orm.undefer()` for usage examples.

`undefer_group`(*loadopt*, *name*)
Produce a new `Load` object with the `orm.undefer_group()` option applied.

See `orm.undefer_group()` for usage examples.

`with_expression`(*loadopt*, *key*, *expression*)
Produce a new `Load` object with the `orm.with_expression()` option applied.

See `orm.with_expression()` for usage examples.

`sqlalchemy.orm.join`(*left*, *right*, *onclause=None*, *isouter=False*, *full=False*, *join_to_left=None*)
Produce an inner join between left and right clauses.

`orm.join()` is an extension to the core join interface provided by `sql.expression.join()`, where the left and right selectables may be not only core selectable objects such as `Table`, but also mapped classes or `AliasedClass` instances. The "on" clause can be a SQL expression, or an attribute or string name referencing a configured `relationship()`.

`orm.join()` is not commonly needed in modern usage, as its functionality is encapsulated within that of the `Query.join()` method, which features a significant amount of automation beyond `orm.join()` by itself. Explicit usage of `orm.join()` with `Query` involves usage of the `Query.select_from()` method, as in:

```
from sqlalchemy.orm import join
session.query(User).\
    select_from(join(User, Address, User.addresses)).\
    filter(Address.email_address=='foo@bar.com')
```

In modern SQLAlchemy the above join can be written more succinctly as:

```
session.query(User).\
        join(User.addresses).\
        filter(Address.email_address=='foo@bar.com')
```

See `Query.join()` for information on modern usage of ORM level joins.

Changed in version 0.8.1: - the `join_to_left` parameter is no longer used, and is deprecated.

`sqlalchemy.orm.outerjoin`(*left*, *right*, *onclause=None*, *full=False*, *join_to_left=None*)
Produce a left outer join between left and right clauses.

This is the "outer join" version of the `orm.join()` function, featuring the same behavior except that an OUTER JOIN is generated. See that function's documentation for other usage details.

`sqlalchemy.orm.with_parent`(*instance*, *prop*, *from_entity=None*)
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.

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.
- **`from_entity`** – Entity in which to consider as the left side. This defaults to the "zero" entity of the `Query` itself.

  New in version 1.2.

2.5 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`.

2.5.1 Session Basics

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 with another Session when you want to work with them again, so that they can resume their normal task of representing database state.

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 class 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 import create_engine
from sqlalchemy.orm import sessionmaker

# an Engine, which the Session will use for connection
# resources
some_engine = create_engine('postgresql://scott:tiger@localhost/')

# 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 factory for us, which we assign to the name Session. This factory, when called, will create a new Session object using the configurational arguments we've given the factory. In this case, as is typical, we've configured the factory to specify a particular Engine for connection resources.

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.

When you write your application, place the sessionmaker factory at the global level. This factory can then be used by the rest of the application as the source of new Session instances, keeping the configuration for how Session objects are constructed in one place.

The sessionmaker factory can also be used in conjunction with other helpers, which are passed a user-defined sessionmaker that is then maintained by the helper. Some of these helpers are discussed in the section session_faq_whentocreate.

Adding Additional Configuration to an Existing sessionmaker()

A common scenario is where the sessionmaker is invoked at module import time, however the generation of one or more Engine instances to be associated with the sessionmaker has not yet proceeded. For this use case, the sessionmaker construct offers the sessionmaker.configure() method, which will place additional configuration directives into an existing sessionmaker that will take place when the construct is invoked:

```
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()
```

Creating Ad-Hoc Session Objects with Alternate Arguments

For the use case where an application needs to create a new `Session` with special arguments that deviate from what is normally used throughout the application, such as a `Session` that binds to an alternate source of connectivity, or a `Session` that should have other arguments such as `expire_on_commit` established differently from what most of the application wants, specific arguments can be passed to the `sessionmaker` factory's `sessionmaker.__call__()` method. These arguments will override whatever configurations have already been placed, such as below, where a new `Session` is constructed against a specific `Connection`:

```
# at the module level, the global sessionmaker,
# bound to a specific Engine
Session = sessionmaker(bind=engine)

# later, some unit of code wants to create a
# Session that is bound to a specific Connection
conn = engine.connect()
session = Session(bind=conn)
```

The typical rationale for the association of a `Session` with a specific `Connection` is that of a test fixture that maintains an external transaction - see session_external_transaction for an example of this.

Session Frequently Asked Questions

By this point, many users already have questions about sessions. This section presents a mini-FAQ (note that we have also a *real FAQ*) of the most basic issues one is presented with when using a `Session`.

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 `sessionmaker.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. The calls to instantiate `Session` would then be placed at the point in the application where database conversations begin.

When do I construct a `Session`, when do I commit it, and when do I close it?

tl;dr;

1. As a general rule, keep the lifecycle of the session **separate and external** from functions and objects that access and/or manipulate database data. This will greatly help with achieving a predictable and consistent transactional scope.
2. Make sure you have a clear notion of where transactions begin and end, and keep transactions **short**, meaning, they end at the series of a sequence of operations, instead of being held open indefinitely.

A `Session` is typically constructed at the beginning of a logical operation where database access is potentially anticipated.

The `Session`, whenever it is used to talk to the database, begins a database transaction as soon as it starts communicating. Assuming the `autocommit` flag is left at its recommended default of `False`, this transaction remains in progress until the `Session` is rolled back, committed, or closed. The `Session` will begin a new transaction if it is used again, subsequent to the previous transaction ending; from this it follows that the `Session` is capable of having a lifespan across many transactions, though only one at a time. We refer to these two concepts as **transaction scope** and **session scope**.

The implication here is that the SQLAlchemy ORM is encouraging the developer to establish these two scopes in their application, including not only when the scopes begin and end, but also the expanse of those scopes, for example should a single `Session` instance be local to the execution flow within a function or method, should it be a global object used by the entire application, or somewhere in between these two.

The burden placed on the developer to determine this scope is one area where the SQLAlchemy ORM necessarily has a strong opinion about how the database should be used. The unit of work pattern is specifically one of accumulating changes over time and flushing them periodically, keeping in-memory state in sync with what's known to be present in a local transaction. This pattern is only effective when meaningful transaction scopes are in place.

It's usually not very hard to determine the best points at which to begin and end the scope of a `Session`, though the wide variety of application architectures possible can introduce challenging situations.

A common choice is to tear down the `Session` at the same time the transaction ends, meaning the transaction and session scopes are the same. This is a great choice to start out with as it removes the need to consider session scope as separate from transaction scope.

While there's no one-size-fits-all recommendation for how transaction scope should be determined, there are common patterns. Especially if one is writing a web application, the choice is pretty much established.

A web application is the easiest case because such an application is already constructed around a single, consistent scope - this is the **request**, which represents an incoming request from a browser, the processing of that request to formulate a response, and finally the delivery of that response back to the client. Integrating web applications with the `Session` is then the straightforward task of linking the scope of the `Session` to that of the request. The `Session` can be established as the request begins, or using a lazy initialization pattern which establishes one as soon as it is needed. The request then proceeds, with some system in place where application logic can access the current `Session` in a manner associated with how the actual request object is accessed. As the request ends, the `Session` is torn down as well, usually through the usage of event hooks provided by the web framework. The transaction used by the `Session` may also be committed at this point, or alternatively the application may opt for an explicit commit pattern, only committing for those requests where one is warranted, but still always tearing down the `Session` unconditionally at the end.

Some web frameworks include infrastructure to assist in the task of aligning the lifespan of a `Session` with that of a web request. This includes products such as Flask-SQLAlchemy, for usage in conjunction with the Flask web framework, and Zope-SQLAlchemy, typically used with the Pyramid framework. SQLAlchemy recommends that these products be used as available.

In those situations where the integration libraries are not provided or are insufficient, SQLAlchemy includes its own "helper" class known as `scoped_session`. A tutorial on the usage of this object is at unitofwork_contextual. It provides both a quick way to associate a `Session` with the current thread, as well as patterns to associate `Session` objects with other kinds of scopes.

As mentioned before, for non-web applications there is no one clear pattern, as applications themselves don't have just one pattern of architecture. The best strategy is to attempt to demarcate "operations", points at which a particular thread begins to perform a series of operations for some period of time, which can be committed at the end. Some examples:

- A background daemon which spawns off child forks would want to create a `Session` local to each child process, work with that `Session` through the life of the "job" that the fork is handling, then tear it down when the job is completed.
- For a command-line script, the application would create a single, global `Session` that is established when the program begins to do its work, and commits it right as the program is completing its task.
- For a GUI interface-driven application, the scope of the `Session` may best be within the scope of a user-generated event, such as a button push. Or, the scope may correspond to explicit user interaction, such as the user "opening" a series of records, then "saving" them.

As a general rule, the application should manage the lifecycle of the session *externally* to functions that deal with specific data. This is a fundamental separation of concerns which keeps data-specific operations agnostic of the context in which they access and manipulate that data.

E.g. **don't do this**:

```
### this is the **wrong way to do it** ###

class ThingOne(object):
    def go(self):
        session = Session()
        try:
            session.query(FooBar).update({"x": 5})
            session.commit()
        except:
            session.rollback()
            raise

class ThingTwo(object):
    def go(self):
        session = Session()
        try:
            session.query(Widget).update({"q": 18})
            session.commit()
        except:
            session.rollback()
            raise

def run_my_program():
    ThingOne().go()
    ThingTwo().go()
```

Keep the lifecycle of the session (and usually the transaction) **separate and external**:

```
### this is a **better** (but not the only) way to do it ###

class ThingOne(object):
    def go(self, session):
        session.query(FooBar).update({"x": 5})

class ThingTwo(object):
    def go(self, session):
```

```
        session.query(Widget).update({"q": 18})

def run_my_program():
    session = Session()
    try:
        ThingOne().go(session)
        ThingTwo().go(session)

        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()
```

The most comprehensive approach, recommended for more substantial applications, will try to keep the details of session, transaction and exception management as far as possible from the details of the program doing its work. For example, we can further separate concerns using a context manager:

```
### another way (but again *not the only way*) to do it ###

from contextlib import contextmanager

@contextmanager
def session_scope():
    """Provide a transactional scope around a series of operations."""
    session = Session()
    try:
        yield session
        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()


def run_my_program():
    with session_scope() as session:
        ThingOne().go(session)
        ThingTwo().go(session)
```

Is the Session a cache?

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

Additionally, the Session stores object instances using a weak reference by default. This also defeats the purpose of using the Session as a cache.

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 dogpile.cache, via the examples_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)
```

The newer core_inspection_toplevel system can also be used:

```
from sqlalchemy import inspect
session = inspect(someobject).session
```

Is the session thread-safe?

The Session is very much intended to be used in a **non-concurrent** fashion, which usually means in only one thread at a time.

The Session should be used in such a way that one instance exists for a single series of operations within a single transaction. One expedient way to get this effect is by associating a Session with the current thread (see unitofwork_contextual for background). Another is to use a pattern where the Session is passed between functions and is otherwise not shared with other threads.

The bigger point is that 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.

Making sure the Session is only used in a single concurrent thread at a time is called a "share nothing" approach to concurrency. But actually, not sharing the Session implies a more significant pattern; it means not just the Session object itself, but also **all objects that are associated with that Session**, must be kept within the scope of a single concurrent thread. The set of mapped objects associated with a Session are essentially proxies for data within database rows accessed over a database connection, and so just like the Session itself, the whole set of objects is really just a large-scale proxy for a database connection (or connections). Ultimately, it's mostly the DBAPI connection itself that we're keeping away from concurrent access; but since the Session and all the objects associated with it are all proxies for that DBAPI connection, the entire graph is essentially not safe for concurrent access.

If there are in fact multiple threads participating in the same task, then you may consider sharing the session and its objects between those threads; however, in this extremely unusual scenario the application would need to ensure that a proper locking scheme is implemented so that there isn't *concurrent* access to the Session or its state. A more common approach to this situation is to maintain a single Session per concurrent thread, but to instead *copy* objects from one Session to another, often using the Session.merge() method to copy the state of an object into a new object local to a different Session.

Basics of Using a Session

The most basic Session use patterns are presented here.

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 ormtutorial_toplevel, and further documented in query_api_toplevel.

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 unitofwork_cascades.

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()
```

Deleting Objects Referenced from Collections and Scalar Relationships

The ORM in general never modifies the contents of a collection or scalar relationship during the flush process. This means, if your class has a `relationship()` that refers to a collection of objects, or a reference to a single object such as many-to-one, the contents of this attribute will not be modified when

the flush process occurs. Instead, if the `Session` is expired afterwards, either through the expire-on-commit behavior of `Session.commit()` or through explicit use of `Session.expire()`, the referenced object or collection upon a given object associated with that `Session` will be cleared and will re-load itself upon next access.

This behavior is not to be confused with the flush process' impact on column- bound attributes that refer to foreign key and primary key columns; these attributes are modified liberally within the flush, since these are the attributes that the flush process intends to manage. Nor should it be confused with the behavior of backreferences, as described at relationships_backref; a backreference event will modify a collection or scalar attribute reference, however this behavior takes place during direct manipulation of related collections and object references, which is explicit within the calling application and is outside of the flush process.

A common confusion that arises regarding this behavior involves the use of the `delete()` method. When `Session.delete()` is invoked upon an object and the `Session` is flushed, the row is deleted from the database. Rows that refer to the target row via foreign key, assuming they are tracked using a `relationship()` between the two mapped object types, will also see their foreign key attributes UPDATED to null, or if delete cascade is set up, the related rows will be deleted as well. However, even though rows related to the deleted object might be themselves modified as well, **no changes occur to relationship-bound collections or object references on the objects** involved in the operation within the scope of the flush itself. This means if the object was a member of a related collection, it will still be present on the Python side until that collection is expired. Similarly, if the object were referenced via many-to-one or one-to-one from another object, that reference will remain present on that object until the object is expired as well.

Below, we illustrate that after an `Address` object is marked for deletion, it's still present in the collection associated with the parent `User`, even after a flush:

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

When the above session is committed, all attributes are expired. The next access of `user.addresses` will re-load the collection, revealing the desired state:

```
>>> session.commit()
>>> address in user.addresses
False
```

The usual practice of deleting items within collections is to forego the usage of `delete()` directly, and instead use cascade behavior to automatically invoke the deletion as a result of removing the object from the parent collection. The `delete-orphan` cascade accomplishes this, as illustrated in the example below:

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

    # ...

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

# ...

del user.addresses[1]
session.flush()
```

Where above, upon removing the `Address` object from the `User.addresses` collection, the `delete-orphan` cascade has the effect of marking the `Address` object for deletion in the same way as passing it to `delete()`.

The `delete-orphan` cascade can also be applied to a many-to-one or one-to-one relationship, so that when an object is de-associated from its parent, it is also automatically marked for deletion. Using `delete-orphan` cascade on a many-to-one or one-to-one requires an additional flag `relationship.single_parent` which invokes an assertion that this related object is not to shared with any other parent simultaneously:

```
class User(Base):
    # ...

    preference = relationship(
        "Preference", cascade="all, delete, delete-orphan",
        single_parent=True)
```

Above, if a hypothetical `Preference` object is removed from a `User`, it will be deleted on flush:

```
some_user.preference = None
session.flush()  # will delete the Preference object
```

See also unitofwork_cascades for detail on cascades.

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.

Note: The term "transaction" here refers to a transactional construct within the `Session` itself which may be maintaining zero or more actual database (DBAPI) transactions. An individual DBAPI connection begins participation in the "transaction" as it is first used to execute a SQL statement, then remains present until the session-level "transaction" is completed. See unitofwork_transaction for further detail.

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`.

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.

2.5.2 State Management

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).
- **Deleted** - An instance which has been deleted within a flush, but the transaction has not yet completed. Objects in this state are essentially in the opposite of "pending" state; when the session's transaction is committed, the object will move to the detached state. Alternatively, when the session's transaction is rolled back, a deleted object moves *back* to the persistent state.

  Changed in version 1.1: The 'deleted' state is a newly added session object state distinct from the 'persistent' state.
- **Detached** - an instance which corresponds, or previously corresponded, to a record in the database, but is not currently in any session. The detached object will contain a database identity marker, however because it is not associated with a session, it is unknown whether or not this database identity actually exists in a target database. Detached objects are safe to use normally, except that they have no ability to load unloaded attributes or attributes that were previously marked as "expired".

For a deeper dive into all possible state transitions, see the section session_lifecycle_events which describes each transition as well as how to programmatically track each one.

Getting the Current State of an Object

The actual state of any mapped object can be viewed at any time using the `inspect()` system:

```
>>> from sqlalchemy import inspect
>>> insp = inspect(my_object)
>>> insp.persistent
True
```

See also:

`InstanceState.transient`

`InstanceState.pending`

`InstanceState.persistent`

`InstanceState.deleted`

`InstanceState.detached`

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

# dictionary of all persistent objects, keyed on their
# identity key
session.identity_map
```

(Documentation: `Session.new`, `Session.dirty`, `Session.deleted`, `Session.identity_map`).

Session Referencing Behavior

Objects within the session are *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 cause objects in the `Session` to remain strongly referenced, usually a simple approach is all that's needed. Examples of externally managed strong-referencing behavior include loading objects into a local dictionary keyed to their primary key, or into lists or sets for the span of time that they need to remain referenced. These collections can be associated with a `Session`, if desired, by placing them into the `Session.info` dictionary.

An event based approach is also feasible. A simple recipe that provides "strong referencing" behavior for all objects as they remain within the persistent state is as follows:

```
from sqlalchemy import event

def strong_reference_session(session):
    @event.listens_for(session, "pending_to_persistent")
    @event.listens_for(session, "deleted_to_persistent")
    @event.listens_for(session, "detached_to_persistent")
    @event.listens_for(session, "loaded_as_persistent")
    def strong_ref_object(sess, instance):
        if 'refs' not in sess.info:
            sess.info['refs'] = refs = set()
        else:
            refs = sess.info['refs']
```

```
        refs.add(instance)


    @event.listens_for(session, "persistent_to_detached")
    @event.listens_for(session, "persistent_to_deleted")
    @event.listens_for(session, "persistent_to_transient")
    def deref_object(sess, instance):
        sess.info['refs'].discard(instance)
```

Above, we intercept the `SessionEvents.pending_to_persistent()`, `SessionEvents.detached_to_persistent()`, `SessionEvents.deleted_to_persistent()` and `SessionEvents.loaded_as_persistent()` event hooks in order to intercept objects as they enter the persistent transition, and the `SessionEvents.persistent_to_detached()` and `SessionEvents.persistent_to_deleted()` hooks to intercept objects as they leave the persistent state.

The above function may be called for any `Session` in order to provide strong-referencing behavior on a per-`Session` basis:

```
from sqlalchemy.orm import Session

my_session = Session()
strong_reference_session(my_session)
```

It may also be called for any `sessionmaker`:

```
from sqlalchemy.orm import sessionmaker

maker = sessionmaker()
strong_reference_session(maker)
```

Merging

`merge()` transfers state from an outside object into a new or already existing instance within a session. It also reconciles the incoming data against the state of the database, producing a history stream which will be applied towards the next flush, or alternatively can be made to produce a simple "transfer" of state without producing change history or accessing the database. 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 locate that instance in the local identity map. If the `load=True` flag is left at its default, it also checks the database for this primary key if not located locally.
- If the given instance has no primary key, or if no instance can be found with the primary key given, a new instance is created.
- The state of the given instance is then copied onto the located/newly created instance. For attributes which are present on the source instance, the value is transferred to the target instance. For mapped attributes which aren't present on the source, the attribute is expired on the target instance, discarding its existing value.

  If the `load=True` flag is left at its default, this copy process emits events and will load the target object's unloaded collections for each attribute present on the source object, so that the incoming state can be reconciled against what's present in the database. If `load` is passed as `False`, the incoming data is "stamped" directly without producing any history.
- The operation is cascaded to related objects and collections, as indicated by the `merge` cascade (see unitofwork_cascades).

- The new instance is returned.

With `merge()`, the given "source" instance is not modified nor is it associated with the target `Session`, and remains available to be merged with any number of other `Session` objects. `merge()` is useful for taking the state of any kind of object structure without regard for its origins or current session associations and copying its state into a new session. Here's some 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, loading each object from the database by primary key and then updating its state with the new state given.
- An application is storing objects in an in-memory cache, shared by many `Session` objects simultaneously. `merge()` is used each time an object is retrieved from the cache to create a local copy of it in each `Session` which requests it. The cached object remains detached; only its state is moved into copies of itself that are local to individual `Session` objects.

  In the caching use case, it's common to use the `load=False` flag to remove the overhead of reconciling the object's state with the database. There's also a "bulk" version of `merge()` called `merge_result()` that was designed to work with cache-extended `Query` objects - see the section examples_caching.
- An application wants to transfer the state of a series of objects into a `Session` maintained by a worker thread or other concurrent system. `merge()` makes a copy of each object to be placed into this new `Session`. At the end of the operation, the parent thread/process maintains the objects it started with, and the thread/worker can proceed with local copies of those objects.

  In the "transfer between threads/processes" use case, the application may want to use the `load=False` flag as well to avoid overhead and redundant SQL queries as the data is transferred.

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 transference 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.

The cascade_backrefs=False option of relationship() will also prevent the Address from being added to the session via the a1.user = u1 assignment.

Further detail on cascade operation is at unitofwork_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()
```

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()`).

Refreshing / Expiring

Expiring means that the database-persisted data held inside a series of object attributes is erased, in such a way that when those attributes are next accessed, a SQL query is emitted which will refresh that data from the database.

When we talk about expiration of data we are usually talking about an object that is in the persistent state. For example, if we load an object as follows:

```
user = session.query(User).filter_by(name='user1').first()
```

The above `User` object is persistent, and has a series of attributes present; if we were to look inside its `__dict__`, we'd see that state loaded:

```
>>> user.__dict__
{
  'id': 1, 'name': u'user1',
  '_sa_instance_state': <...>,
}
```

where `id` and `name` refer to those columns in the database. `_sa_instance_state` is a non-database-persisted value used by SQLAlchemy internally (it refers to the `InstanceState` for the instance. While not directly relevant to this section, if we want to get at it, we should use the `inspect()` function to access it).

At this point, the state in our `User` object matches that of the loaded database row. But upon expiring the object using a method such as `Session.expire()`, we see that the state is removed:

```
>>> session.expire(user)
>>> user.__dict__
{'_sa_instance_state': <...>}
```

We see that while the internal "state" still hangs around, the values which correspond to the `id` and `name` columns are gone. If we were to access one of these columns and are watching SQL, we'd see this:

```
>>> print(user.name)
{opensql}SELECT user.id AS user_id, user.name AS user_name
FROM user
WHERE user.id = ?
(1,)
{stop}user1
```

Above, upon accessing the expired attribute `user.name`, the ORM initiated a lazy load to retrieve the most recent state from the database, by emitting a SELECT for the user row to which this user refers. Afterwards, the `__dict__` is again populated:

```
>>> user.__dict__
{
  'id': 1, 'name': u'user1',
  '_sa_instance_state': <...>,
}
```

Note: While we are peeking inside of `__dict__` in order to see a bit of what SQLAlchemy does with object attributes, we **should not modify** the contents of `__dict__` directly, at least as far as those attributes which the SQLAlchemy ORM is maintaining (other attributes outside of SQLA's realm are fine). This is because SQLAlchemy uses descriptors in order to track the changes we make to an object, and when we modify `__dict__` directly, the ORM won't be able to track that we changed something.

Another key behavior of both `expire()` and `refresh()` is that all un-flushed changes on an object are discarded. That is, if we were to modify an attribute on our `User`:

```
>>> user.name = 'user2'
```

but then we call `expire()` without first calling `flush()`, our pending value of `'user2'` is discarded:

```
>>> session.expire(user)
>>> user.name
'user1'
```

The `expire()` method can be used to mark as "expired" all ORM-mapped attributes for an instance:

```
# expire all ORM-mapped attributes on obj1
session.expire(obj1)
```

it can also be passed a list of string attribute names, referring to specific attributes to be marked as expired:

```
# expire only attributes obj1.attr1, obj1.attr2
session.expire(obj1, ['attr1', 'attr2'])
```

The `refresh()` method has a similar interface, but instead of expiring, it emits an immediate SELECT for the object's row immediately:

```
# reload all attributes on obj1
session.refresh(obj1)
```

`refresh()` also accepts a list of string attribute names, but unlike `expire()`, expects at least one name to be that of a column-mapped attribute:

```
# reload obj1.attr1, obj1.attr2
session.refresh(obj1, ['attr1', 'attr2'])
```

The `Session.expire_all()` method allows us to essentially call `Session.expire()` on all objects contained within the `Session` at once:

```
session.expire_all()
```

What Actually Loads

The SELECT statement that's emitted when an object marked with `expire()` or loaded with `refresh()` varies based on several factors, including:

- The load of expired attributes is triggered from **column-mapped attributes only**. While any kind of attribute can be marked as expired, including a `relationship()` - mapped attribute, accessing an expired `relationship()` attribute will emit a load only for that attribute, using standard relationship-oriented lazy loading. Column-oriented attributes, even if expired, will not load as part of this operation, and instead will load when any column-oriented attribute is accessed.
- `relationship()`- mapped attributes will not load in response to expired column-based attributes being accessed.
- Regarding relationships, `refresh()` is more restrictive than `expire()` with regards to attributes that aren't column-mapped. Calling `refresh()` and passing a list of names that only includes relationship-mapped attributes will actually raise an error. In any case, non-eager-loading `relationship()` attributes will not be included in any refresh operation.
- `relationship()` attributes configured as "eager loading" via the `lazy` parameter will load in the case of `refresh()`, if either no attribute names are specified, or if their names are inclued in the list of attributes to be refreshed.
- Attributes that are configured as `deferred()` will not normally load, during either the expired-attribute load or during a refresh. An unloaded attribute that's `deferred()` instead loads on its own when directly accessed, or if part of a "group" of deferred attributes where an unloaded attribute in that group is accessed.
- For expired attributes that are loaded on access, a joined-inheritance table mapping will emit a SELECT that typically only includes those tables for which unloaded attributes are present. The action here is sophisticated enough to load only the parent or child table, for example, if the subset of columns that were originally expired encompass only one or the other of those tables.
- When `refresh()` is used on a joined-inheritance table mapping, the SELECT emitted will resemble that of when `Session.query()` is used on the target object's class. This is typically all those tables that are set up as part of the mapping.

When to Expire or Refresh

The `Session` uses the expiration feature automatically whenever the transaction referred to by the session ends. Meaning, whenever `Session.commit()` or `Session.rollback()` is called, all objects within the `Session` are expired, using a feature equivalent to that of the `Session.expire_all()` method. The rationale is that the end of a transaction is a demarcating point at which there is no more context available in order to know what the current state of the database is, as any number of other transactions may be affecting it. Only when a new transaction starts can we again have access to the current state of the database, at which point any number of changes may have occurred.

Transaction Isolation

Of course, most databases are capable of handling multiple transactions at once, even involving the same rows of data. When a relational database handles multiple transactions involving the same tables or rows, this is when the isolation aspect of the database comes into play. The isolation behavior of different databases varies considerably and even on a single database can be configured to behave in

different ways (via the so-called isolation level setting). In that sense, the `Session` can't fully predict when the same SELECT statement, emitted a second time, will definitely return the data we already have, or will return new data. So as a best guess, it assumes that within the scope of a transaction, unless it is known that a SQL expression has been emitted to modify a particular row, there's no need to refresh a row unless explicitly told to do so.

The `Session.expire()` and `Session.refresh()` methods are used in those cases when one wants to force an object to re-load its data from the database, in those cases when it is known that the current state of data is possibly stale. Reasons for this might include:

- some SQL has been emitted within the transaction outside of the scope of the ORM's object handling, such as if a `Table.update()` construct were emitted using the `Session.execute()` method;
- if the application is attempting to acquire data that is known to have been modified in a concurrent transaction, and it is also known that the isolation rules in effect allow this data to be visible.

The second bullet has the important caveat that "it is also known that the isolation rules in effect allow this data to be visible." This means that it cannot be assumed that an UPDATE that happened on another database connection will yet be visible here locally; in many cases, it will not. This is why if one wishes to use `expire()` or `refresh()` in order to view data between ongoing transactions, an understanding of the isolation behavior in effect is essential.

See also:

`Session.expire()`

`Session.expire_all()`

`Session.refresh()`

isolation - glossary explanation of isolation which includes links to Wikipedia.

The SQLAlchemy Session In-Depth - a video + slides with an in-depth discussion of the object lifecycle including the role of data expiration.

2.5.3 Cascades

Mappers support the concept of configurable cascade behavior on `relationship()` constructs. This refers to how operations performed on a "parent" object relative to a particular `Session` should be propagated to items referred to by that relationship (e.g. "child" objects), and is affected by the `relationship.cascade` option.

The default behavior of cascade is limited to cascades of the so-called cascade_save_update and cascade_merge settings. The typical "alternative" setting for cascade is to add the cascade_delete and cascade_delete_orphan options; these settings are appropriate for related objects which only exist as long as they are attached to their parent, and are otherwise deleted.

Cascade behavior is configured using the `cascade` option on `relationship()`:

```
class Order(Base):
    __tablename__ = 'order'

    items = relationship("Item", cascade="all, delete-orphan")
    customer = relationship("User", cascade="save-update")
```

To set cascades on a backref, the same flag can be used with the `backref()` function, which ultimately feeds its arguments back into `relationship()`:

```
class Item(Base):
    __tablename__ = 'item'

    order = relationship("Order",
```

```
                backref=backref("items", cascade="all, delete-orphan")
            )
```

The Origins of Cascade

SQLAlchemy's notion of cascading behavior on relationships, as well as the options to configure them, are primarily derived from the similar feature in the Hibernate ORM; Hibernate refers to "cascade" in a few places such as in Example: Parent/Child. If cascades are confusing, we'll refer to their conclusion, stating "The sections we have just covered can be a bit confusing. However, in practice, it all works out nicely."

The default value of `cascade` is `save-update, merge`. The typical alternative setting for this parameter is either `all` or more commonly `all, delete-orphan`. The `all` symbol is a synonym for `save-update, merge, refresh-expire, expunge, delete`, and using it in conjunction with `delete-orphan` indicates that the child object should follow along with its parent in all cases, and be deleted once it is no longer associated with that parent.

The list of available values which can be specified for the `cascade` parameter are described in the following subsections.

save-update

`save-update` cascade indicates that when an object is placed into a `Session` via `Session.add()`, all the objects associated with it via this `relationship()` should also be added to that same `Session`. Suppose we have an object `user1` with two related objects `address1`, `address2`:

```
>>> user1 = User()
>>> address1, address2 = Address(), Address()
>>> user1.addresses = [address1, address2]
```

If we add `user1` to a `Session`, it will also add `address1`, `address2` implicitly:

```
>>> sess = Session()
>>> sess.add(user1)
>>> address1 in sess
True
```

`save-update` cascade also affects attribute operations for objects that are already present in a `Session`. If we add a third object, `address3` to the `user1.addresses` collection, it becomes part of the state of that `Session`:

```
>>> address3 = Address()
>>> user1.append(address3)
>>> address3 in sess
>>> True
```

`save-update` has the possibly surprising behavior which is that persistent objects which were *removed* from a collection or in some cases a scalar attribute may also be pulled into the `Session` of a parent object; this is so that the flush process may handle that related object appropriately. This case can usually only arise if an object is removed from one `Session` and added to another:

```
>>> user1 = sess1.query(User).filter_by(id=1).first()
>>> address1 = user1.addresses[0]
>>> sess1.close()    # user1, address1 no longer associated with sess1
>>> user1.addresses.remove(address1)  # address1 no longer associated with user1
>>> sess2 = Session()
>>> sess2.add(user1)    # ... but it still gets added to the new session,
```

```
>>> address1 in sess2  # because it's still "pending" for flush
True
```

The `save-update` cascade is on by default, and is typically taken for granted; it simplifies code by allowing a single call to `Session.add()` to register an entire structure of objects within that `Session` at once. While it can be disabled, there is usually not a need to do so.

One case where `save-update` cascade does sometimes get in the way is in that it takes place in both directions for bi-directional relationships, e.g. backrefs, meaning that the association of a child object with a particular parent can have the effect of the parent object being implicitly associated with that child object's `Session`; this pattern, as well as how to modify its behavior using the `cascade_backrefs` flag, is discussed in the section backref_cascade.

delete

The `delete` cascade indicates that when a "parent" object is marked for deletion, its related "child" objects should also be marked for deletion. If for example we we have a relationship `User.addresses` with `delete` cascade configured:

```
class User(Base):
    # ...

    addresses = relationship("Address", cascade="save-update, merge, delete")
```

If using the above mapping, we have a `User` object and two related `Address` objects:

```
>>> user1 = sess.query(User).filter_by(id=1).first()
>>> address1, address2 = user1.addresses
```

If we mark `user1` for deletion, after the flush operation proceeds, `address1` and `address2` will also be deleted:

```
>>> sess.delete(user1)
>>> sess.commit()
{opensql}DELETE FROM address WHERE address.id = ?
((1,), (2,))
DELETE FROM user WHERE user.id = ?
(1,)
COMMIT
```

Alternatively, if our `User.addresses` relationship does *not* have `delete` cascade, SQLAlchemy's default behavior is to instead de-associate `address1` and `address2` from `user1` by setting their foreign key reference to `NULL`. Using a mapping as follows:

```
class User(Base):
    # ...

    addresses = relationship("Address")
```

Upon deletion of a parent `User` object, the rows in `address` are not deleted, but are instead de-associated:

```
>>> sess.delete(user1)
>>> sess.commit()
{opensql}UPDATE address SET user_id=? WHERE address.id = ?
(None, 1)
UPDATE address SET user_id=? WHERE address.id = ?
(None, 2)
DELETE FROM user WHERE user.id = ?
(1,)
COMMIT
```

`delete` cascade is more often than not used in conjunction with cascade_delete_orphan cascade, which will emit a DELETE for the related row if the "child" object is deassociated from the parent. The combination of `delete` and `delete-orphan` cascade covers both situations where SQLAlchemy has to decide between setting a foreign key column to NULL versus deleting the row entirely.

ORM-level "delete" cascade vs. FOREIGN KEY level "ON DELETE" cascade

The behavior of SQLAlchemy's "delete" cascade has a lot of overlap with the `ON DELETE CASCADE` feature of a database foreign key, as well as with that of the `ON DELETE SET NULL` foreign key setting when "delete" cascade is not specified. Database level "ON DELETE" cascades are specific to the "FOREIGN KEY" construct of the relational database; SQLAlchemy allows configuration of these schema-level constructs at the DDL level using options on `ForeignKeyConstraint` which are described at on_update_on_delete.

It is important to note the differences between the ORM and the relational database's notion of "cascade" as well as how they integrate:

- A database level `ON DELETE` cascade is configured effectively on the **many-to-one** side of the relationship; that is, we configure it relative to the `FOREIGN KEY` constraint that is the "many" side of a relationship. At the ORM level, **this direction is reversed**. SQLAlchemy handles the deletion of "child" objects relative to a "parent" from the "parent" side, which means that `delete` and `delete-orphan` cascade are configured on the **one-to-many** side.
- Database level foreign keys with no `ON DELETE` setting are often used to **prevent** a parent row from being removed, as it would necessarily leave an unhandled related row present. If this behavior is desired in a one-to-many relationship, SQLAlchemy's default behavior of setting a foreign key to `NULL` can be caught in one of two ways:
  - The easiest and most common is just to set the foreign-key-holding column to `NOT NULL` at the database schema level. An attempt by SQLAlchemy to set the column to NULL will fail with a simple NOT NULL constraint exception.
  - The other, more special case way is to set the `passive_deletes` flag to the string `"all"`. This has the effect of entirely disabling SQLAlchemy's behavior of setting the foreign key column to NULL, and a DELETE will be emitted for the parent row without any affect on the child row, even if the child row is present in memory. This may be desirable in the case when database-level foreign key triggers, either special `ON DELETE` settings or otherwise, need to be activated in all cases when a parent row is deleted.
- Database level `ON DELETE` cascade is **vastly more efficient** than that of SQLAlchemy. The database can chain a series of cascade operations across many relationships at once; e.g. if row A is deleted, all the related rows in table B can be deleted, and all the C rows related to each of those B rows, and on and on, all within the scope of a single DELETE statement. SQLAlchemy on the other hand, in order to support the cascading delete operation fully, has to individually load each related collection in order to target all rows that then may have further related collections. That is, SQLAlchemy isn't sophisticated enough to emit a DELETE for all those related rows at once within this context.
- SQLAlchemy doesn't **need** to be this sophisticated, as we instead provide smooth integration with the database's own `ON DELETE` functionality, by using the `passive_deletes` option in conjunction with properly configured foreign key constraints. Under this behavior, SQLAlchemy only emits DELETE for those rows that are already locally present in the `Session`; for any collections that are unloaded, it leaves them to the database to handle, rather than emitting a SELECT for them. The section passive_deletes provides an example of this use.
- While database-level `ON DELETE` functionality works only on the "many" side of a relationship, SQLAlchemy's "delete" cascade has **limited** ability to operate in the *reverse* direction as well, meaning it can be configured on the "many" side to delete an object on the "one" side when the reference on the "many" side is deleted. However this can easily result in constraint violations if there are other objects referring to this "one" side from the "many", so it typically is only

useful when a relationship is in fact a "one to one". The `single_parent` flag should be used to establish an in-Python assertion for this case.

When using a `relationship()` that also includes a many-to-many table using the `secondary` option, SQLAlchemy's delete cascade handles the rows in this many-to-many table automatically. Just like, as described in relationships_many_to_many_deletion, the addition or removal of an object from a many-to-many collection results in the INSERT or DELETE of a row in the many-to-many table, the `delete` cascade, when activated as the result of a parent object delete operation, will DELETE not just the row in the "child" table but also in the many-to-many table.

delete-orphan

`delete-orphan` cascade adds behavior to the `delete` cascade, such that a child object will be marked for deletion when it is de-associated from the parent, not just when the parent is marked for deletion. This is a common feature when dealing with a related object that is "owned" by its parent, with a NOT NULL foreign key, so that removal of the item from the parent collection results in its deletion.

`delete-orphan` cascade implies that each child object can only have one parent at a time, so is configured in the vast majority of cases on a one-to-many relationship. Setting it on a many-to-one or many-to-many relationship is more awkward; for this use case, SQLAlchemy requires that the `relationship()` be configured with the `single_parent` argument, establishes Python-side validation that ensures the object is associated with only one parent at a time.

merge

`merge` cascade indicates that the `Session.merge()` operation should be propagated from a parent that's the subject of the `Session.merge()` call down to referred objects. This cascade is also on by default.

refresh-expire

`refresh-expire` is an uncommon option, indicating that the `Session.expire()` operation should be propagated from a parent down to referred objects. When using `Session.refresh()`, the referred objects are expired only, but not actually refreshed.

expunge

`expunge` cascade indicates that when the parent object is removed from the `Session` using `Session.expunge()`, the operation should be propagated down to referred objects.

Controlling Cascade on Backrefs

The cascade_save_update cascade by default takes place on attribute change events emitted from backrefs. This is probably a confusing statement more easily described through demonstration; it means that, given a mapping such as this:

```
mapper(Order, order_table, properties={
    'items' : relationship(Item, 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 `items` 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.items
True
>>> i1 in session
True
```

This behavior can be disabled using the `cascade_backrefs` flag:

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

So above, the assignment of `i1.order = o1` will append `i1` to the `items` 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.

2.5.4 Transactions and Connection Management

Managing Transactions

A newly constructed `Session` may be said to be in the "begin" state. In this state, the `Session` has not established any connection or transactional state with any of the `Engine` objects that may be associated with it.

The `Session` then receives requests to operate upon a database connection. Typically, this means it is called upon to execute SQL statements using a particular `Engine`, which may be via `Session.query()`, `Session.execute()`, or within a flush operation of pending data, which occurs when such state exists and `Session.commit()` or `Session.flush()` is called.

As these requests are received, each new `Engine` encountered is associated with an ongoing transactional state maintained by the `Session`. When the first `Engine` is operated upon, the `Session` can be said to have left the "begin" state and entered "transactional" state. For each `Engine` encountered, a `Connection` is associated with it, which is acquired via the `Engine.contextual_connect()` method. If a `Connection` was directly associated with the `Session` (see session_external_transaction for an example of this), it is added to the transactional state directly.

For each `Connection`, the `Session` also maintains a `Transaction` object, which is acquired by calling `Connection.begin()` on each `Connection`, or if the `Session` object has been established using the flag `twophase=True`, a `TwoPhaseTransaction` object acquired via `Connection.begin_twophase()`. These transactions are all committed or rolled back corresponding to the invocation of the `Session.commit()` and `Session.rollback()` methods. A commit operation will also call the `TwoPhaseTransaction.prepare()` method on all transactions if applicable.

When the transactional state is completed after a rollback or commit, the `Session` releases all `Transaction` and `Connection` resources, and goes back to the "begin" state, which will again invoke new `Connection` and `Transaction` objects as new requests to emit SQL statements are received.

The example below illustrates this lifecycle:

```
engine = create_engine("...")
Session = sessionmaker(bind=engine)

# new session.   no connections are in use.
```

```
session = Session()
try:
    # first query.  a Connection is acquired
    # from the Engine, and a Transaction
    # started.
    item1 = session.query(Item).get(1)

    # second query.  the same Connection/Transaction
    # are used.
    item2 = session.query(Item).get(2)

    # pending changes are created.
    item1.foo = 'bar'
    item2.bar = 'foo'

    # commit.  The pending changes above
    # are flushed via flush(), the Transaction
    # is committed, the Connection object closed
    # and discarded, the underlying DBAPI connection
    # returned to the connection pool.
    session.commit()
except:
    # on rollback, the same closure of state
    # as that of commit proceeds.
    session.rollback()
    raise
finally:
    # close the Session.  This will expunge any remaining
    # objects as well as reset any existing SessionTransaction
    # state.  Neither of these steps are usually essential.
    # However, if the commit() or rollback() itself experienced
    # an unanticipated internal failure (such as due to a mis-behaved
    # user-defined event handler), .close() will ensure that
    # invalid state is removed.
    session.close()
```

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. (But note that if the return value is used as a context manager, i.e. in a with-statement, then this rollback/commit is issued by the context manager upon exiting the context, and so should not be added explicitly.)

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 caus-

ing all subsequent attribute/instance access to reference the full state of the `Session` right before `begin_nested()` was called.

`begin_nested()`, in the same manner as the less often used `begin()` method, returns a `SessionTransaction` object which works as a context manager. It can be succinctly used around individual record inserts in order to catch things like unique constraint exceptions:

```
for record in records:
    try:
        with session.begin_nested():
            session.merge(record)
    except:
        print("Skipped record %s" % record)
session.commit()
```

Autocommit Mode

The examples of session lifecycle at unitofwork_transaction refer to a `Session` that runs in its default mode of `autocommit=False`. In this mode, the `Session` begins new transactions automatically as soon as it needs to do work upon a database connection; the transaction then stays in progress until the `Session.commit()` or `Session.rollback()` methods are called.

The `Session` also features an older legacy mode of use called **autocommit mode**, where a transaction is not started implicitly, and unless the the `Session.begin()` method is invoked, the `Session` will perform each database operation on a new connection checked out from the connection pool, which is then released back to the pool immediately after the operation completes. This refers to methods like `Session.execute()` as well as when executing a query returned by `Session.query()`. For a flush operation, the `Session` starts a new transaction for the duration of the flush, and commits it when complete.

> **Warning:** "autocommit" mode is a **legacy mode of use** and should not be considered for new projects. If autocommit mode is used, it is strongly advised that the application at least ensure that tranasction scope is made present via the `Session.begin()` method, rather than using the session in pure autocommit mode.
>
> If the `Session.begin()` method is not used, and operations are allowed to proceed using ad-hoc connections with immediate autocommit, then the application probably should set `autoflush=False, expire_on_commit=False`, since these features are intended to be used only within the context of a database transaction.

Modern usage of "autocommit mode" tends to be for framework integrations that wish to control specifically when the "begin" state occurs. A session which is configured with `autocommit=True` may be placed into the "begin" state using the `Session.begin()` method. After the cycle completes upon `Session.commit()` or `Session.rollback()`, connection and transaction resources are released and the `Session` goes back into "autocommit" mode, until `Session.begin()` is called again:

```
Session = sessionmaker(bind=engine, 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 `Session.begin()` method also returns a transactional token which is compatible with the `with` statement:

```
Session = sessionmaker(bind=engine, 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 Subtransactions with Autocommit

A subtransaction indicates usage of the `Session.begin()` method in conjunction with the `subtransactions=True` flag. This produces a non-transactional, delimiting construct that allows nesting of calls to `begin()` and `commit()`. Its 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.

`subtransactions=True` is generally only useful in conjunction with autocommit, and is equivalent to the pattern described at connections_nested_transactions, 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:

```
# method_a starts a transaction and calls method_b
def method_a(session):
    session.begin(subtransactions=True)
    try:
        method_b(session)
        session.commit()  # transaction is committed here
    except:
        session.rollback() # rolls back the transaction
        raise

# method_b also starts a transaction, but when
# called from method_a participates in the ongoing
# transaction.
def method_b(session):
    session.begin(subtransactions=True)
    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()
```

Subtransactions are used by the `Session.flush()` process to ensure that the flush operation takes place within a transaction, regardless of autocommit. When autocommit is disabled, it is still useful in that it forces the `Session` into a "pending rollback" state, as a failed flush cannot be resumed in mid-operation, where the end user still maintains the "scope" of the transaction overall.

Enabling Two-Phase Commit

For backends which support two-phase operaration (currently MySQL and PostgreSQL), 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()
```

Setting Transaction Isolation Levels

Isolation refers to the behavior of the transaction at the database level in relation to other transactions occurring concurrently. There are four well-known modes of isolation, and typically the Python DBAPI allows these to be set on a per-connection basis, either through explicit APIs or via database-specific calls.

SQLAlchemy's dialects support settable isolation modes on a per-`Engine` or per-`Connection` basis, using flags at both the `create_engine()` level as well as at the `Connection.execution_options()` level.

When using the ORM `Session`, it acts as a *facade* for engines and connections, but does not expose transaction isolation directly. So in order to affect transaction isolation level, we need to act upon the `Engine` or `Connection` as appropriate.

See also:

`create_engine.isolation_level`

SQLite Transaction Isolation

PostgreSQL Isolation Level

MySQL Isolation Level

Setting Isolation Engine-Wide

To set up a `Session` or `sessionmaker` with a specific isolation level globally, use the `create_engine.isolation_level` parameter:

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

eng = create_engine(
    "postgresql://scott:tiger@localhost/test",
    isolation_level='REPEATABLE_READ')

maker = sessionmaker(bind=eng)
```

```
session = maker()
```

Setting Isolation for Individual Sessions

When we make a new `Session`, either using the constructor directly or when we call upon the callable produced by a `sessionmaker`, we can pass the `bind` argument directly, overriding the pre-existing bind. We can combine this with the `Engine.execution_options()` method in order to produce a copy of the original `Engine` that will add this option:

```
session = maker(
    bind=engine.execution_options(isolation_level='SERIALIZABLE'))
```

For the case where the `Session` or `sessionmaker` is configured with multiple "binds", we can either re-specify the `binds` argument fully, or if we want to only replace specific binds, we can use the `Session.bind_mapper()` or `Session.bind_table()` methods:

```
session = maker()
session.bind_mapper(
    User, user_engine.execution_options(isolation_level='SERIALIZABLE'))
```

We can also use the individual transaction method that follows.

Setting Isolation for Individual Transactions

A key caveat regarding isolation level is that the setting cannot be safely modified on a `Connection` where a transaction has already started. Databases cannot change the isolation level of a transaction in progress, and some DBAPIs and SQLAlchemy dialects have inconsistent behaviors in this area. Some may implicitly emit a ROLLBACK and some may implicitly emit a COMMIT, others may ignore the setting until the next transaction. Therefore SQLAlchemy emits a warning if this option is set when a transaction is already in play. The `Session` object does not provide for us a `Connection` for use in a transaction where the transaction is not already begun. So here, we need to pass execution options to the `Session` at the start of a transaction by passing `Session.connection.execution_options` provided by the `Session.connection()` method:

```
from sqlalchemy.orm import Session

sess = Session(bind=engine)
sess.connection(execution_options={'isolation_level': 'SERIALIZABLE'})

# work with session

# commit transaction.  the connection is released
# and reverted to its previous isolation level.
sess.commit()
```

Above, we first produce a `Session` using either the constructor or a `sessionmaker`. Then we explicitly set up the start of a transaction by calling upon `Session.connection()`, which provides for execution options that will be passed to the connection before the transaction is begun. If we are working with a `Session` that has multiple binds or some other custom scheme for `Session.get_bind()`, we can pass additional arguments to `Session.connection()` in order to affect how the bind is procured:

```
sess = my_sesssionmaker()

# set up a transaction for the bind associated with
# the User mapper
sess.connection(
    mapper=User,
```

```
    execution_options={'isolation_level': 'SERIALIZABLE'})

# work with session

# commit transaction.  the connection is released
# and reverted to its previous isolation level.
sess.commit()
```

The `Session.connection.execution_options` argument is only accepted on the **first** call to `Session.connection()` for a particular bind within a transaction. If a transaction is already begun on the target connection, a warning is emitted:

```
>>> session = Session(eng)
>>> session.execute("select 1")
<sqlalchemy.engine.result.ResultProxy object at 0x1017a6c50>
>>> session.connection(execution_options={'isolation_level': 'SERIALIZABLE'})
sqlalchemy/orm/session.py:310: SAWarning: Connection is already established
for the given bind; execution_options ignored
```

New in version 0.9.9: Added the `Session.connection.execution_options` parameter to `Session.connection()`.

Tracking Transaction State with Events

See the section session_transaction_events for an overview of the available event hooks for session transaction state changes.

Joining a Session into an External Transaction (such as for test suites)

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 = self.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):
        self.session.close()

        # rollback - everything that happened with the
        # Session above (including calls to commit())
        # is rolled back.
        self.trans.rollback()

        # return connection to the Engine
        self.connection.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.

Supporting Tests with Rollbacks

The above recipe works well for any kind of database enabled test, except for a test that needs to actually invoke `Session.rollback()` within the scope of the test itself. The above recipe can be expanded, such that the `Session` always runs all operations within the scope of a SAVEPOINT, which is established at the start of each transaction, so that tests can also rollback the "transaction" as well while still remaining in the scope of a larger "transaction" that's never committed, using two extra events:

```
from sqlalchemy import event


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)

        # start the session in a SAVEPOINT...
        self.session.begin_nested()

        # then each time that SAVEPOINT ends, reopen it
        @event.listens_for(self.session, "after_transaction_end")
        def restart_savepoint(session, transaction):
            if transaction.nested and not transaction._parent.nested:

                # ensure that state is expired the way
                # session.commit() at the top level normally does
                # (optional step)
                session.expire_all()

                session.begin_nested()

    # ... the tearDown() method stays the same
```

2.5.5 Additional Persistence Techniques

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.

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)
```

Forcing NULL on a column with a default

The ORM considers any attribute that was never set on an object as a "default" case; the attribute will be omitted from the INSERT statement:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(String(50), nullable=True)

obj = MyObject(id=1)
session.add(obj)
session.commit()  # INSERT with the 'data' column omitted; the database
                  # itself will persist this as the NULL value
```

Omitting a column from the INSERT means that the column will have the NULL value set, *unless* the column has a default set up, in which case the default value will be persisted. This holds true both from a pure SQL perspective with server-side defaults, as well as the behavior of SQLAlchemy's insert behavior with both client-side and server-side defaults:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(String(50), nullable=True, server_default="default")

obj = MyObject(id=1)
session.add(obj)
session.commit()  # INSERT with the 'data' column omitted; the database
                  # itself will persist this as the value 'default'
```

However, in the ORM, even if one assigns the Python value `None` explicitly to the object, this is treated the **same** as though the value were never assigned:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(String(50), nullable=True, server_default="default")

obj = MyObject(id=1, data=None)
session.add(obj)
session.commit()  # INSERT with the 'data' column explicitly set to None;
                  # the ORM still omits it from the statement and the
                  # database will still persist this as the value 'default'
```

The above operation will persist into the `data` column the server default value of `"default"` and not SQL NULL, even though `None` was passed; this is a long-standing behavior of the ORM that many applications hold as an assumption.

So what if we want to actually put NULL into this column, even though the column has a default value? There are two approaches. One is that on a per-instance level, we assign the attribute using the `null` SQL construct:

```
from sqlalchemy import null

obj = MyObject(id=1, data=null())
session.add(obj)
session.commit()  # INSERT with the 'data' column explicitly set as null();
                  # the ORM uses this directly, bypassing all client-
                  # and server-side defaults, and the database will
                  # persist this as the NULL value
```

The `null` SQL construct always translates into the SQL NULL value being directly present in the target INSERT statement.

If we'd like to be able to use the Python value `None` and have this also be persisted as NULL despite the presence of column defaults, we can configure this for the ORM using a Core-level modifier `TypeEngine.evaluates_none()`, which indicates a type where the ORM should treat the value `None` the same as any other value and pass it through, rather than omitting it as a "missing" value:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(
      String(50).evaluates_none(),  # indicate that None should always be passed
      nullable=True, server_default="default")

obj = MyObject(id=1, data=None)
session.add(obj)
session.commit()  # INSERT with the 'data' column explicitly set to None;
                  # the ORM uses this directly, bypassing all client-
                  # and server-side defaults, and the database will
                  # persist this as the NULL value
```

Evaluating None

The `TypeEngine.evaluates_none()` modifier is primarily intended to signal a type where the Python value "None" is significant, the primary example being a JSON type which may want to persist the JSON `null` value rather than SQL NULL. We are slightly repurposing it here in order to signal to the ORM that we'd like `None` to be passed into the type whenever present, even though no special type-level behaviors are assigned to it.

New in version 1.1: added the `TypeEngine.evaluates_none()` method in order to indicate that a "None" value should be treated as significant.

Partitioning Strategies

Simple 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()
```

Above, operations against either class will make usage of the `Engine` linked to that class. Upon a flush operation, similar rules take place to ensure each class is written to the right database.

The transactions among the multiple databases can optionally be coordinated via two phase commit, if the underlying backend supports it. See session_twophase for an example.

Custom Vertical Partitioning

More comprehensive rule-based class-level partitioning can be built by overriding the `Session.get_bind()` method. Below we illustrate a custom `Session` which delivers the following rules:

1. Flush operations are delivered to the engine named `master`.

2. Operations on objects that subclass `MyOtherClass` all occur on the `other` engine.
3. Read operations for all other classes occur on a random choice of the `slave1` or `slave2` database.

```
engines = {
    'master':create_engine("sqlite:///master.db"),
    'other':create_engine("sqlite:///other.db"),
    'slave1':create_engine("sqlite:///slave1.db"),
    'slave2':create_engine("sqlite:///slave2.db"),
}

from sqlalchemy.orm import Session, sessionmaker
import random

class RoutingSession(Session):
    def get_bind(self, mapper=None, clause=None):
        if mapper and issubclass(mapper.class_, MyOtherClass):
            return engines['other']
        elif self._flushing:
            return engines['master']
        else:
            return engines[
                random.choice(['slave1','slave2'])
            ]
```

The above `Session` class is plugged in using the `class_` argument to `sessionmaker`:

```
Session = sessionmaker(class_=RoutingSession)
```

This approach can be combined with multiple `MetaData` objects, using an approach such as that of using the declarative `__abstract__` keyword, described at declarative_abstract.

Horizontal Partitioning

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

See the "sharding" example: examples_sharding.

Bulk Operations

Note: Bulk Operations mode is a new series of operations made available on the `Session` object for the purpose of invoking INSERT and UPDATE statements with greatly reduced Python overhead, at the expense of much less functionality, automation, and error checking. As of SQLAlchemy 1.0, these features should be considered as "beta", and additionally are intended for advanced users.

New in version 1.0.0.

Bulk operations on the `Session` include `Session.bulk_save_objects()`, `Session.bulk_insert_mappings()`, and `Session.bulk_update_mappings()`. The purpose of these methods is to directly expose internal elements of the unit of work system, such that facilities for emitting INSERT and UPDATE statements given dictionaries or object states can be utilized alone, bypassing the normal unit of work mechanics of state, relationship and attribute management. The advantages to this approach is strictly one of reduced Python overhead:

- The flush() process, including the survey of all objects, their state, their cascade status, the status of all objects associated with them via `relationship()`, and the topological sort of all operations to be performed is completely bypassed. This reduces a great amount of Python overhead.

- The objects as given have no defined relationship to the target `Session`, even when the operation is complete, meaning there's no overhead in attaching them or managing their state in terms of the identity map or session.
- The `Session.bulk_insert_mappings()` and `Session.bulk_update_mappings()` methods accept lists of plain Python dictionaries, not objects; this further reduces a large amount of overhead associated with instantiating mapped objects and assigning state to them, which normally is also subject to expensive tracking of history on a per-attribute basis.
- The set of objects passed to all bulk methods are processed in the order they are received. In the case of `Session.bulk_save_objects()`, when objects of different types are passed, the INSERT and UPDATE statements are necessarily broken up into per-type groups. In order to reduce the number of batch INSERT or UPDATE statements passed to the DBAPI, ensure that the incoming list of objects are grouped by type.
- The process of fetching primary keys after an INSERT also is disabled by default. When performed correctly, INSERT statements can now more readily be batched by the unit of work process into `executemany()` blocks, which perform vastly better than individual statement invocations.
- UPDATE statements can similarly be tailored such that all attributes are subject to the SET clase unconditionally, again making it much more likely that `executemany()` blocks can be used.

The performance behavior of the bulk routines should be studied using the examples_performance example suite. This is a series of example scripts which illustrate Python call-counts across a variety of scenarios, including bulk insert and update scenarios.

See also:

examples_performance - includes detailed examples of bulk operations contrasted against traditional Core and ORM methods, including performance metrics.

Usage

The methods each work in the context of the `Session` object's transaction, like any other:

```
s = Session()
objects = [
    User(name="u1"),
    User(name="u2"),
    User(name="u3")
]
s.bulk_save_objects(objects)
```

For `Session.bulk_insert_mappings()`, and `Session.bulk_update_mappings()`, dictionaries are passed:

```
s.bulk_insert_mappings(User,
  [dict(name="u1"), dict(name="u2"), dict(name="u3")]
)
```

See also:

`Session.bulk_save_objects()`

`Session.bulk_insert_mappings()`

`Session.bulk_update_mappings()`

Comparison to Core Insert / Update Constructs

The bulk methods offer performance that under particular circumstances can be close to that of using the core `Insert` and `Update` constructs in an "executemany" context (for a description of "executemany", see

execute_multiple in the Core tutorial). In order to achieve this, the `Session.bulk_insert_mappings.return_defaults` flag should be disabled so that rows can be batched together. The example suite in examples_performance should be carefully studied in order to gain familiarity with how fast bulk performance can be achieved.

ORM Compatibility

The bulk insert / update methods lose a significant amount of functionality versus traditional ORM use. The following is a listing of features that are **not available** when using these methods:

- persistence along `relationship()` linkages
- sorting of rows within order of dependency; rows are inserted or updated directly in the order in which they are passed to the methods
- Session-management on the given objects, including attachment to the session, identity map management.
- Functionality related to primary key mutation, ON UPDATE cascade
- SQL expression inserts / updates (e.g. flush_embedded_sql_expressions)
- ORM events such as `MapperEvents.before_insert()`, etc. The bulk session methods have no event support.

Features that **are available** include:

- INSERTs and UPDATEs of mapped objects
- Version identifier support
- Multi-table mappings, such as joined-inheritance - however, an object to be inserted across multiple tables either needs to have primary key identifiers fully populated ahead of time, else the `Session.bulk_save_objects.return_defaults` flag must be used, which will greatly reduce the performance benefits

2.5.6 Contextual/Thread-local Sessions

Recall from the section session_faq_whentocreate, the concept of "session scopes" was introduced, with an emphasis on web applications and the practice of linking the scope of a `Session` with that of a web request. Most modern web frameworks include integration tools so that the scope of the `Session` can be managed automatically, and these tools should be used as they are available.

SQLAlchemy includes its own helper object, which helps with the establishment of user-defined `Session` scopes. It is also used by third-party integration systems to help construct their integration schemes.

The object is the `scoped_session` object, and it represents a **registry** of `Session` objects. If you're not familiar with the registry pattern, a good introduction can be found in Patterns of Enterprise Architecture.

Note: The `scoped_session` object is a very popular and useful object used by many SQLAlchemy applications. However, it is important to note that it presents **only one approach** to the issue of `Session` management. If you're new to SQLAlchemy, and especially if the term "thread-local variable" seems strange to you, we recommend that if possible you familiarize first with an off-the-shelf integration system such as Flask-SQLAlchemy or zope.sqlalchemy.

A `scoped_session` is constructed by calling it, passing it a **factory** which can create new `Session` objects. A factory is just something that produces a new object when called, and in the case of `Session`, the most common factory is the `sessionmaker`, introduced earlier in this section. Below we illustrate this usage:

```
>>> from sqlalchemy.orm import scoped_session
>>> from sqlalchemy.orm import sessionmaker

>>> session_factory = sessionmaker(bind=some_engine)
>>> Session = scoped_session(session_factory)
```

The `scoped_session` object we've created will now call upon the `sessionmaker` when we "call" the registry:

```
>>> some_session = Session()
```

Above, `some_session` is an instance of `Session`, which we can now use to talk to the database. This same `Session` is also present within the `scoped_session` registry we've created. If we call upon the registry a second time, we get back the **same** `Session`:

```
>>> some_other_session = Session()
>>> some_session is some_other_session
True
```

This pattern allows disparate sections of the application to call upon a global `scoped_session`, so that all those areas may share the same session without the need to pass it explicitly. The `Session` we've established in our registry will remain, until we explicitly tell our registry to dispose of it, by calling `scoped_session.remove()`:

```
>>> Session.remove()
```

The `scoped_session.remove()` method first calls `Session.close()` on the current `Session`, which has the effect of releasing any connection/transactional resources owned by the `Session` first, then discarding the `Session` itself. "Releasing" here means that connections are returned to their connection pool and any transactional state is rolled back, ultimately using the `rollback()` method of the underlying DBAPI connection.

At this point, the `scoped_session` object is "empty", and will create a **new** `Session` when called again. As illustrated below, this is not the same `Session` we had before:

```
>>> new_session = Session()
>>> new_session is some_session
False
```

The above series of steps illustrates the idea of the "registry" pattern in a nutshell. With that basic idea in hand, we can discuss some of the details of how this pattern proceeds.

Implicit Method Access

The job of the `scoped_session` is simple; hold onto a `Session` for all who ask for it. As a means of producing more transparent access to this `Session`, the `scoped_session` also includes **proxy behavior**, meaning that the registry itself can be treated just like a `Session` directly; when methods are called on this object, they are **proxied** to the underlying `Session` being maintained by the registry:

```
Session = scoped_session(some_factory)

# equivalent to:
#
# session = Session()
# print(session.query(MyClass).all())
#
print(Session.query(MyClass).all())
```

The above code accomplishes the same task as that of acquiring the current `Session` by calling upon the registry, then using that `Session`.

Thread-Local Scope

Users who are familiar with multithreaded programming will note that representing anything as a global variable is usually a bad idea, as it implies that the global object will be accessed by many threads concurrently. The `Session` object is entirely designed to be used in a **non-concurrent** fashion, which in terms of multithreading means "only in one thread at a time". So our above example of `scoped_session` usage, where the same `Session` object is maintained across multiple calls, suggests that some process needs to be in place such that multiple calls across many threads don't actually get a handle to the same session. We call this notion **thread local storage**, which means, a special object is used that will maintain a distinct object per each application thread. Python provides this via the threading.local() construct. The `scoped_session` object by default uses this object as storage, so that a single `Session` is maintained for all who call upon the `scoped_session` registry, but only within the scope of a single thread. Callers who call upon the registry in a different thread get a `Session` instance that is local to that other thread.

Using this technique, the `scoped_session` provides a quick and relatively simple (if one is familiar with thread-local storage) way of providing a single, global object in an application that is safe to be called upon from multiple threads.

The `scoped_session.remove()` method, as always, removes the current `Session` associated with the thread, if any. However, one advantage of the `threading.local()` object is that if the application thread itself ends, the "storage" for that thread is also garbage collected. So it is in fact "safe" to use thread local scope with an application that spawns and tears down threads, without the need to call `scoped_session.remove()`. However, the scope of transactions themselves, i.e. ending them via `Session.commit()` or `Session.rollback()`, will usually still be something that must be explicitly arranged for at the appropriate time, unless the application actually ties the lifespan of a thread to the lifespan of a transaction.

Using Thread-Local Scope with Web Applications

As discussed in the section session_faq_whentocreate, a web application is architected around the concept of a **web request**, and integrating such an application with the `Session` usually implies that the `Session` will be associated with that request. As it turns out, most Python web frameworks, with notable exceptions such as the asynchronous frameworks Twisted and Tornado, use threads in a simple way, such that a particular web request is received, processed, and completed within the scope of a single *worker thread*. When the request ends, the worker thread is released to a pool of workers where it is available to handle another request.

This simple correspondence of web request and thread means that to associate a `Session` with a thread implies it is also associated with the web request running within that thread, and vice versa, provided that the `Session` is created only after the web request begins and torn down just before the web request ends. So it is a common practice to use `scoped_session` as a quick way to integrate the `Session` with a web application. The sequence diagram below illustrates this flow:

```
Web Server          Web Framework        SQLAlchemy ORM Code
--------------      --------------       ------------------------------
startup        ->   Web framework        # Session registry is established
                    initializes          Session = scoped_session(sessionmaker())

incoming
web request    ->   web request     ->   # The registry is *optionally*
                    starts               # called upon explicitly to create
                                         # a Session local to the thread and/or request
                                         Session()

                                         # the Session registry can otherwise
                                         # be used at any time, creating the
                                         # request-local Session() if not present,
                                         # or returning the existing one
                                         Session.query(MyClass) # ...
```

```
                                    Session.add(some_object) # ...

                                    # if data was modified, commit the
                                    # transaction
                                    Session.commit()

                  web request ends  -> # the registry is instructed to
                                    # remove the Session
                                    Session.remove()

                  sends output      <-
outgoing web    <-
response
```

Using the above flow, the process of integrating the `Session` with the web application has exactly two requirements:

1. Create a single `scoped_session` registry when the web application first starts, ensuring that this object is accessible by the rest of the application.
2. Ensure that `scoped_session.remove()` is called when the web request ends, usually by integrating with the web framework's event system to establish an "on request end" event.

As noted earlier, the above pattern is **just one potential way** to integrate a `Session` with a web framework, one which in particular makes the significant assumption that the **web framework associates web requests with application threads**. It is however **strongly recommended that the integration tools provided with the web framework itself be used, if available**, instead of `scoped_session`.

In particular, while using a thread local can be convenient, it is preferable that the `Session` be associated **directly with the request**, rather than with the current thread. The next section on custom scopes details a more advanced configuration which can combine the usage of `scoped_session` with direct request based scope, or any kind of scope.

Using Custom Created Scopes

The `scoped_session` object's default behavior of "thread local" scope is only one of many options on how to "scope" a `Session`. A custom scope can be defined based on any existing system of getting at "the current thing we are working with".

Suppose a web framework defines a library function `get_current_request()`. An application built using this framework can call this function at any time, and the result will be some kind of `Request` object that represents the current request being processed. If the `Request` object is hashable, then this function can be easily integrated with `scoped_session` to associate the `Session` with the request. Below we illustrate this in conjunction with a hypothetical event marker provided by the web framework `on_request_end`, which allows code to be invoked whenever a request ends:

```
from my_web_framework import get_current_request, on_request_end
from sqlalchemy.orm import scoped_session, sessionmaker

Session = scoped_session(sessionmaker(bind=some_engine), scopefunc=get_current_request)

@on_request_end
def remove_session(req):
    Session.remove()
```

Above, we instantiate `scoped_session` in the usual way, except that we pass our request-returning function as the "scopefunc". This instructs `scoped_session` to use this function to generate a dictionary key whenever the registry is called upon to return the current `Session`. In this case it is particularly

important that we ensure a reliable "remove" system is implemented, as this dictionary is not otherwise self-managed.

Contextual Session API

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

Provides scoped management of `Session` objects.

See unitofwork_contextual for a tutorial.

configure(***kwargs*)

reconfigure the `sessionmaker` used by this `scoped_session`.

See `sessionmaker.configure()`.

query_property(*query_cls=None*)

return a class property which produces a `Query` object against the class and the current `Session` 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 `Session`, if present.

This will first call `Session.close()` method on the current `Session`, which releases any existing transactional/connection resources still being held; transactions specifically are rolled back. The `Session` is then discarded. Upon next usage within the same scope, the `scoped_session` will produce a new `Session` object.

session_factory = None

The *session_factory* provided to *__init__* is stored in this attribute and may be accessed at a later time. This can be useful when a new non-scoped `Session` or `Connection` to the database is needed.

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.

clear()

Clear the current scope, if any.

has()

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

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

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

2.5.7 Tracking Object and Session Changes with Events

SQLAlchemy features an extensive Event Listening system used throughout the Core and ORM. Within the ORM, there are a wide variety of event listener hooks, which are documented at an API level at orm_event_toplevel. This collection of events has grown over the years to include lots of very useful new events as well as some older events that aren't as relevant as they once were. This section will attempt to introduce the major event hooks and when they might be used.

Persistence Events

Probably the most widely used series of events are the "persistence" events, which correspond to the flush process. The flush is where all the decisions are made about pending changes to objects and are then emitted out to the database in the form of INSERT, UPDATE, and DELETE staetments.

before_flush()

The SessionEvents.before_flush() hook is by far the most generally useful event to use when an application wants to ensure that additional persistence changes to the database are made when a flush proceeds. Use SessionEvents.before_flush() in order to operate upon objects to validate their state as well as to compose additional objects and references before they are persisted. Within this event, it is **safe to manipulate the Session's state**, that is, new objects can be attached to it, objects can be deleted, and indivual attributes on objects can be changed freely, and these changes will be pulled into the flush process when the event hook completes.

The typical SessionEvents.before_flush() hook will be tasked with scanning the collections Session.new, Session.dirty and Session.deleted in order to look for objects where something will be happening.

For illustrations of SessionEvents.before_flush(), see examples such as examples_versioned_history and examples_versioned_rows.

after_flush()

The SessionEvents.after_flush() hook is called after the SQL has been emitted for a flush process, but **before** the state of the objects that were flushed has been altered. That is, you can still inspect the Session.new, Session.dirty and Session.deleted collections to see what was just flushed, and you can also use history tracking features like the ones provided by AttributeState to see what changes were just persisted. In the SessionEvents.after_flush() event, additional SQL can be emitted to the database based on what's observed to have changed.

after_flush_postexec()

SessionEvents.after_flush_postexec() is called soon after SessionEvents.after_flush(), but is invoked **after** the state of the objects has been modified to account for the flush that just took place. The Session.new, Session.dirty and Session.deleted collections are normally completely empty here. Use SessionEvents.after_flush_postexec() to inspect the identity map for finalized objects and possibly emit additional SQL. In this hook, there is the ability to make new changes on objects, which means the Session will again go into a "dirty" state; the mechanics of the Session here will cause it to

flush **again** if new changes are detected in this hook if the flush were invoked in the context of `Session.commit()`; otherwise, the pending changes will be bundled as part of the next normal flush. When the hook detects new changes within a `Session.commit()`, a counter ensures that an endless loop in this regard is stopped after 100 iterations, in the case that an `SessionEvents.after_flush_postexec()` hook continually adds new state to be flushed each time it is called.

Mapper-level Events

In addition to the flush-level hooks, there is also a suite of hooks that are more fine-grained, in that they are called on a per-object basis and are broken out based on INSERT, UPDATE or DELETE. These are the mapper persistence hooks, and they too are very popular, however these events need to be approached more cautiously, as they proceed within the context of the flush process that is already ongoing; many operations are not safe to proceed here.

The events are:

- `MapperEvents.before_insert()`
- `MapperEvents.after_insert()`
- `MapperEvents.before_update()`
- `MapperEvents.after_update()`
- `MapperEvents.before_delete()`
- `MapperEvents.after_delete()`

Each event is passed the `Mapper`, the mapped object itself, and the `Connection` which is being used to emit an INSERT, UPDATE or DELETE statement. The appeal of these events is clear, in that if an application wants to tie some activity to when a specific type of object is persisted with an INSERT, the hook is very specific; unlike the `SessionEvents.before_flush()` event, there's no need to search through collections like `Session.new` in order to find targets. However, the flush plan which represents the full list of every single INSERT, UPDATE, DELETE statement to be emitted has *already been decided* when these events are called, and no changes may be made at this stage. Therefore the only changes that are even possible to the given objects are upon attributes **local** to the object's row. Any other change to the object or other objects will impact the state of the `Session`, which will fail to function properly.

Operations that are not supported within these mapper-level persistence events include:

- `Session.add()`
- `Session.delete()`
- Mapped collection append, add, remove, delete, discard, etc.
- Mapped relationship attribute set/del events, i.e. `someobject.related = someotherobject`

The reason the `Connection` is passed is that it is encouraged that **simple SQL operations take place here**, directly on the `Connection`, such as incrementing counters or inserting extra rows within log tables. When dealing with the `Connection`, it is expected that Core-level SQL operations will be used; e.g. those described in sqlexpression_toplevel.

There are also many per-object operations that don't need to be handled within a flush event at all. The most common alternative is to simply establish additional state along with an object inside its `__init__()` method, such as creating additional objects that are to be associated with the new object. Using validators as described in simple_validators is another approach; these functions can intercept changes to attributes and establish additional state changes on the target object in response to the attribute change. With both of these approaches, the object is in the correct state before it ever gets to the flush step.

Object Lifecycle Events

Another use case for events is to track the lifecycle of objects. This refers to the states first introduced at session_object_states.

New in version 1.1: added a system of events that intercept all possible state transitions of an object within the `Session`.

All the states above can be tracked fully with events. Each event represents a distinct state transition, meaning, the starting state and the destination state are both part of what are tracked. With the exception of the initial transient event, all the events are in terms of the `Session` object or class, meaning they can be associated either with a specific `Session` object:

```
from sqlalchemy import event
from sqlalchemy.orm import Session

session = Session()

@event.listens_for(session, 'transient_to_pending')
def object_is_pending(session, obj):
    print("new pending: %s" % obj)
```

Or with the `Session` class itself, as well as with a specific `sessionmaker`, which is likely the most useful form:

```
from sqlalchemy import event
from sqlalchemy.orm import sessionmaker

maker = sessionmaker()

@event.listens_for(maker, 'transient_to_pending')
def object_is_pending(session, obj):
    print("new pending: %s" % obj)
```

The listeners can of course be stacked on top of one function, as is likely to be common. For example, to track all objects that are entering the persistent state:

```
@event.listens_for(maker, "pending_to_persistent")
@event.listens_for(maker, "deleted_to_persistent")
@event.listens_for(maker, "detached_to_persistent")
@event.listens_for(maker, "loaded_as_persistent")
def detect_all_persistent(session, instance):
    print("object is now persistent: %s" % instance)
```

Transient

All mapped objects when first constructed start out as transient. In this state, the object exists alone and doesn't have an association with any `Session`. For this initial state, there's no specific "transition" event since there is no `Session`, however if one wanted to intercept when any transient object is created, the `InstanceEvents.init()` method is probably the best event. This event is applied to a specific class or superclass. For example, to intercept all new objects for a particular declarative base:

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

Base = declarative_base()

@event.listens_for(Base, "init", propagate=True)
def intercept_init(instance, args, kwargs):
    print("new transient: %s" % instance)
```

Transient to Pending

The transient object becomes pending when it is first associated with a `Session` via the `Session.add()` or `Session.add_all()` method. An object may also become part of a `Session` as a result of a "cascade" from a referencing object that was explicitly added. The transient to pending transition is detectable using the `SessionEvents.transient_to_pending()` event:

```
@event.listens_for(sessionmaker, "transient_to_pending")
def intercept_transient_to_pending(session, object_):
    print("transient to pending: %s" % object_)
```

Pending to Persistent

The pending object becomes persistent when a flush proceeds and an INSERT statement takes place for the instance. The object now has an identity key. Track pending to persistent with the `SessionEvents.pending_to_persistent()` event:

```
@event.listens_for(sessionmaker, "pending_to_persistent")
def intercept_pending_to_persistent(session, object_):
    print("pending to persistent: %s" % object_)
```

Pending to Transient

The pending object can revert back to transient if the `Session.rollback()` method is called before the pending object has been flushed, or if the `Session.expunge()` method is called for the object before it is flushed. Track pending to transient with the `SessionEvents.pending_to_transient()` event:

```
@event.listens_for(sessionmaker, "pending_to_transient")
def intercept_pending_to_transient(session, object_):
    print("transient to pending: %s" % object_)
```

Loaded as Persistent

Objects can appear in the `Session` directly in the persistent state when they are loaded from the database. Tracking this state transition is synonymous with tracking objects as they are loaded, and is synonymous with using the `InstanceEvents.load()` instance-level event. However, the `SessionEvents.loaded_as_persistent()` event is provided as a session-centric hook for intercepting objects as they enter the persistent state via this particular avenue:

```
@event.listens_for(sessionmaker, "loaded_as_persistent")
def intercept_loaded_as_persistent(session, object_):
    print("object loaded into persistent state: %s" % object_)
```

Persistent to Transient

The persistent object can revert to the transient state if the `Session.rollback()` method is called for a transaction where the object was first added as pending. In the case of the ROLLBACK, the INSERT statement that made this object persistent is rolled back, and the object is evicted from the `Session` to again become transient. Track objects that were reverted to transient from persistent using the `SessionEvents.persistent_to_transient()` event hook:

```
@event.listens_for(sessionmaker, "persistent_to_transient")
def intercept_persistent_to_transient(session, object_):
    print("persistent to transient: %s" % object_)
```

Persistent to Deleted

The persistent object enters the deleted state when an object marked for deletion is deleted from the database within the flush process. Note that this is **not the same** as when the `Session.delete()` method is called for a target object. The `Session.delete()` method only **marks** the object for deletion; the actual DELETE statement is not emitted until the flush proceeds. It is subsequent to the flush that the "deleted" state is present for the target object.

Within the "deleted" state, the object is only marginally associated with the `Session`. It is not present in the identity map nor is it present in the `Session.deleted` collection that refers to when it was pending for deletion.

From the "deleted" state, the object can go either to the detached state when the transaction is committed, or back to the persistent state if the transaction is instead rolled back.

Track the persistent to deleted transition with `SessionEvents.persistent_to_deleted()`:

```
@event.listens_for(sessionmaker, "persistent_to_deleted")
def intercept_persistent_to_deleted(session, object_):
    print("object was DELETEd, is now in deleted state: %s" % object_)
```

Deleted to Detached

The deleted object becomes detached when the session's transaction is committed. After the `Session.commit()` method is called, the database transaction is final and the `Session` now fully discards the deleted object and removes all associations to it. Track the deleted to detached transition using `SessionEvents.deleted_to_detached()`:

```
@event.listens_for(sessionmaker, "deleted_to_detached")
def intercept_deleted_to_detached(session, object_):
    print("deleted to detached: %s" % object_)
```

Note: While the object is in the deleted state, the `InstanceState.deleted` attribute, accessible using `inspect(object).deleted`, returns True. However when the object is detached, `InstanceState.deleted` will again return False. To detect that an object was deleted, regardless of whether or not it is detached, use the `InstanceState.was_deleted` accessor.

Persistent to Detached

The persistent object becomes detached when the object is de-associated with the `Session`, via the `Session.expunge()`, `Session.expunge_all()`, or `Session.close()` methods.

Note: An object may also become **implicitly detached** if its owning `Session` is dereferenced by the application and discarded due to garbage collection. In this case, **no event is emitted**.

Track objects as they move from persistent to detached using the `SessionEvents.persistent_to_detached()` event:

```
@event.listens_for(sessionmaker, "persistent_to_detached")
def intecept_persistent_to_detached(session, object_):
    print("object became detached: %s" % object_)
```

Detached to Persistent

The detached object becomes persistent when it is re-associated with a session using the `Session.add()` or equivalent method. Track objects moving back to persistent from detached using the `SessionEvents.detached_to_persistent()` event:

```
@event.listens_for(sessionmaker, "detached_to_persistent")
def intecept_detached_to_persistent(session, object_):
    print("object became persistent again: %s" % object_)
```

Deleted to Persistent

The deleted object can be reverted to the persistent state when the transaction in which it was DELETEd was rolled back using the `Session.rollback()` method. Track deleted objects moving back to the persistent state using the `SessionEvents.deleted_to_persistent()` event:

```
@event.listens_for(sessionmaker, "transient_to_pending")
def intercept_transient_to_pending(session, object_):
    print("transient to pending: %s" % object_)
```

Transaction Events

Transaction events allow an application to be notifed when transaction boundaries occur at the `Session` level as well as when the `Session` changes the transactional state on `Connection` objects.

- `SessionEvents.after_transaction_create()`, `SessionEvents.after_transaction_end()` - these events track the logical transaction scopes of the `Session` in a way that is not specific to individual database connections. These events are intended to help with integration of transaction-tracking systems such as `zope.sqlalchemy`. Use these events when the application needs to align some external scope with the transactional scope of the `Session`. These hooks mirror the "nested" transactional behavior of the `Session`, in that they track logical "subtransactions" as well as "nested" (e.g. SAVEPOINT) transactions.
- `SessionEvents.before_commit()`, `SessionEvents.after_commit()`, `SessionEvents.after_begin()`, `SessionEvents.after_rollback()`, `SessionEvents.after_soft_rollback()` - These events allow tracking of transaction events from the perspective of database connections. `SessionEvents.after_begin()` in particular is a per-connection event; a `Session` that maintains more than one connection will emit this event for each connection individually as those connections become used within the current transaction. The rollback and commit events then refer to when the DBAPI connections themselves have received rollback or commit instructions directly.

Attribute Change Events

The attribute change events allow interception of when specific attributes on an object are modified. These events include `AttributeEvents.set()`, `AttributeEvents.append()`, and `AttributeEvents.remove()`. These events are extremely useful, particularly for per-object validation operations; however, it is often much more convenient to use a "validator" hook, which uses these hooks behind the scenes; see simple_validators for background on this. The attribute events are also behind the mechanics of backreferences. An example illustrating use of attribute events is in examples_instrumentation.

2.5.8 Session API

Session and sessionmaker()

class sqlalchemy.orm.session.**sessionmaker**(*bind=None*, *class_=<class 'sqlalchemy.orm.session.Session'>*, *autoflush=True*, *autocommit=False*, *expire_on_commit=True*, *info=None*, ***kw*)

A configurable `Session` factory.

The `sessionmaker` factory generates new `Session` objects when called, creating them given the configurational arguments established here.

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 method `configure()`, which can be used to specify additional keyword arguments to the factory, which will take effect for subsequent `Session` objects generated. This is usually used to associate one or more `Engine` objects with an existing `sessionmaker` factory before it is first used:

```
# application starts
Session = sessionmaker()

# ... later
engine = create_engine('sqlite:///foo.db')
Session.configure(bind=engine)

sess = Session()
```

close_all()

Close *all* sessions in memory.

configure(***new_kw*)

(Re)configure the arguments for this sessionmaker.

e.g.:

```
Session = sessionmaker()

Session.configure(bind=create_engine('sqlite://'))
```

identity_key(*orm_util*, **args*, ***kwargs*)

Return an identity key.

This is an alias of `util.identity_key()`.

object_session(*instance*)

Return the `Session` to which an object belongs.

This is an alias of `object_session()`.

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*, *enable_baked_queries=True*, *info=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*.

add(*instance*, *_warn=True*)

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.

> **Warning:** The Session.begin() method is part of a larger pattern of use with the Session known as **autocommit mode**. This is essentially a **legacy mode of use** and is not necessary for new applications. The Session normally handles the work of "begin" transparently, which in turn relies upon the Python DBAPI to transparently "begin" transactions; there is **no need to explcitly begin transactions** when using modern Session programming patterns. In its default mode of autocommit=False, the Session does all of its work within the context of a transaction, so as soon as you call Session.commit(), the next transaction is implicitly started when the next database operation is invoked. See session_autocommit for further background.

The method will raise an error if this Session is already inside of a transaction, unless subtransactions or nested are specified. A "subtransaction" is essentially a code embedding pattern that does not affect the transactional state of the database connection unless a rollback is emitted, in which case the whole transaction is rolled back. For documentation on subtransactions, please see session_subtransactions.

Parameters

- **subtransactions** – if True, indicates that this begin() can create a "subtransaction".
- **nested** – if True, begins a SAVEPOINT transaction and is equivalent to calling begin_nested(). For documentation on SAVEPOINT transactions, please see session_begin_nested.

Returns the SessionTransaction object. Note that SessionTransaction acts as a Python context manager, allowing Session.begin() to be used in a "with" block. See session_autocommit for an example.

See also:

session_autocommit

Session.begin_nested()

begin_nested()

Begin a "nested" transaction on this Session, e.g. SAVEPOINT.

The target database(s) and associated drivers must support SQL SAVEPOINT for this method to function correctly.

For documentation on SAVEPOINT transactions, please see session_begin_nested.

Returns the `SessionTransaction` object. Note that `SessionTransaction` acts as a context manager, allowing `Session.begin_nested()` to be used in a "with" block. See session_begin_nested for a usage example.

See also:

session_begin_nested

Serializable isolation / Savepoints / Transactional DDL - special workarounds required with the SQLite driver in order for SAVEPOINT to work correctly.

`bind_mapper`(*mapper*, *bind*)
Associate a `Mapper` with a "bind", e.g. a `Engine` or `Connection`.

The given mapper is added to a lookup used by the `Session.get_bind()` method.

`bind_table`(*table*, *bind*)
Associate a `Table` with a "bind", e.g. a `Engine` or `Connection`.

The given mapper is added to a lookup used by the `Session.get_bind()` method.

`bulk_insert_mappings`(*mapper*, *mappings*, *return_defaults=False*, *render_nulls=False*)
Perform a bulk insert of the given list of mapping dictionaries.

The bulk insert feature allows plain Python dictionaries to be used as the source of simple INSERT operations which can be more easily grouped together into higher performing "executemany" operations. Using dictionaries, there is no "history" or session state management features in use, reducing latency when inserting large numbers of simple rows.

The values within the dictionaries as given are typically passed without modification into Core `Insert()` constructs, after organizing the values within them across the tables to which the given mapper is mapped.

New in version 1.0.0.

> **Warning:** The bulk insert feature allows for a lower-latency INSERT of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw INSERT of records.
>
> **Please read the list of caveats at** bulk_operations **before using this method, and fully test and confirm the functionality of all code developed using these systems.**

Parameters

- `mapper` – a mapped class, or the actual `Mapper` object, representing the single kind of object represented within the mapping list.
- `mappings` – a list of dictionaries, each one containing the state of the mapped row to be inserted, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary must contain all keys to be populated into all tables.
- `return_defaults` – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted **one at a time**, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however, `Session.bulk_insert_mappings.return_defaults` **greatly reduces the performance gains** of the method overall. If the rows to be inserted only refer to a single table, then there is no reason this flag should be set as the returned default information is not used.

- **render_nulls** – When True, a value of None will result in a NULL value being included in the INSERT statement, rather than the column being omitted from the INSERT. This allows all the rows being INSERTed to have the identical set of columns which allows the full set of rows to be batched to the DBAPI. Normally, each column-set that contains a different combination of NULL values than the previous row must omit a different series of columns from the rendered INSERT statement, which means it must be emitted as a separate statement. By passing this flag, the full set of rows are guaranteed to be batchable into one batch; the cost however is that server-side defaults which are invoked by an omitted column will be skipped, so care must be taken to ensure that these are not necessary.

  Warning: When this flag is set, **server side default SQL values will not be invoked** for those columns that are inserted as NULL; the NULL value will be sent explicitly. Care must be taken to ensure that no server-side default functions need to be invoked for the operation as a whole.

  New in version 1.1.

See also:

bulk_operations

Session.bulk_save_objects()

Session.bulk_update_mappings()

bulk_save_objects(*objects*, *return_defaults=False*, *update_changed_only=True*)

Perform a bulk save of the given list of objects.

The bulk save feature allows mapped objects to be used as the source of simple INSERT and UPDATE operations which can be more easily grouped together into higher performing "executemany" operations; the extraction of data from the objects is also performed using a lower-latency process that ignores whether or not attributes have actually been modified in the case of UPDATEs, and also ignores SQL expressions.

The objects as given are not added to the session and no additional state is established on them, unless the **return_defaults** flag is also set, in which case primary key attributes and server-side default values will be populated.

New in version 1.0.0.

Warning: The bulk save feature allows for a lower-latency INSERT/UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw INSERT/UPDATES of records.

Please read the list of caveats at bulk_operations **before using this method, and fully test and confirm the functionality of all code developed using these systems.**

Parameters

- **objects** – a list of mapped object instances. The mapped objects are persisted as is, and are **not** associated with the Session afterwards.

  For each object, whether the object is sent as an INSERT or an UPDATE is dependent on the same rules used by the Session in traditional operation; if the object has the InstanceState.key attribute set, then the object is assumed to be "detached" and will result in an UPDATE. Otherwise, an INSERT is used.

In the case of an UPDATE, statements are grouped based on which attributes have changed, and are thus to be the subject of each SET clause. If `update_changed_only` is False, then all attributes present within each object are applied to the UPDATE statement, which may help in allowing the statements to be grouped together into a larger executemany(), and will also reduce the overhead of checking history on attributes.

- `return_defaults` – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted **one at a time**, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however, `Session.bulk_save_objects.return_defaults` **greatly reduces the performance gains** of the method overall.
- `update_changed_only` – when True, UPDATE statements are rendered based on those attributes in each state that have logged changes. When False, all attributes present are rendered into the SET clause with the exception of primary key attributes.

See also:

bulk_operations

`Session.bulk_insert_mappings()`

`Session.bulk_update_mappings()`

`bulk_update_mappings`(*mapper*, *mappings*)
Perform a bulk update of the given list of mapping dictionaries.

The bulk update feature allows plain Python dictionaries to be used as the source of simple UPDATE operations which can be more easily grouped together into higher performing "executemany" operations. Using dictionaries, there is no "history" or session state management features in use, reducing latency when updating large numbers of simple rows.

New in version 1.0.0.

Warning: The bulk update feature allows for a lower-latency UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw UPDATES of records.

Please read the list of caveats at bulk_operations **before using this method, and fully test and confirm the functionality of all code developed using these systems.**

Parameters

- `mapper` – a mapped class, or the actual `Mapper` object, representing the single kind of object represented within the mapping list.
- `mappings` – a list of dictionaries, each one containing the state of the mapped row to be updated, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary may contain keys corresponding to all tables. All those keys which are present and are not part of the primary key are applied to the SET clause of the UPDATE statement; the primary key values, which are required, are applied to the WHERE clause.

See also:

bulk_operations

Session.bulk_insert_mappings()

Session.bulk_save_objects()

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.

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.

When using the Session in its default mode of 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.

See also:

session_committing

connection(*mapper=None*, *clause=None*, *bind=None*, *close_with_result=False*, *execution_options=None*, ***kw*)
Return a Connection object corresponding to this Session object's transactional state.

If this Session is configured with autocommit=False, either the Connection corresponding to the current transaction is returned, or if no transaction is in progress, a new one is begun and the Connection returned (note that no transactional state is established with the DBAPI until the first SQL statement is emitted).

Alternatively, if this Session is configured with autocommit=True, an ad-hoc Connection is returned using Engine.contextual_connect() on the underlying Engine.

Ambiguity in multi-bind or unbound Session objects can be resolved through any of the optional keyword arguments. This ultimately makes usage of the get_bind() method for resolution.

Parameters

- **bind** – Optional Engine to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over mapper, clause.
- **mapper** – Optional mapper() mapped class, used to identify the appropriate bind. This argument takes precedence over clause.
- **clause** – A ClauseElement (i.e. select(), text(), etc.) which will be used to locate a bind, if a bind cannot otherwise be identified.
- **close_with_result** – Passed to Engine.connect(), indicating the Connection should be considered "single use", automatically closing when the

first result set is closed. This flag only has an effect if this Session is configured with autocommit=True and does not already have a transaction in progress.

- **execution_options** – a dictionary of execution options that will be passed to Connection.execution_options(), **when the connection is first procured only**. If the connection is already present within the Session, a warning is emitted and the arguments are ignored.

  New in version 0.9.9.

  See also:

  session_transaction_isolation

- ****kw** – Additional keyword arguments are sent to get_bind(), allowing additional arguments to be passed to custom implementations of get_bind().

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.

E.g.:

```
some_mapped_object in session.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 Session.is_modified() method.

enable_relationship_loading(*obj*)
Associate an object with this Session for related object loading.

> **Warning:** enable_relationship_loading() exists to serve special use cases and is not recommended for general use.

Accesses of attributes mapped with relationship() will attempt to load a value from the database using this Session as the source of connectivity. The values will be loaded based on foreign key values present on this object - it follows that this functionality generally only works for many-to-one-relationships.

The object will be attached to this session, but will **not** participate in any persistence operations; its state for almost all purposes will remain either "transient" or "detached", except for the case of relationship loading.

Also note that backrefs will often not work as expected. Altering a relationship-bound attribute on the target object may not fire off a backref event, if the effective value is what was already loaded from a foreign-key-holding value.

The Session.enable_relationship_loading() method is similar to the load_on_pending flag on relationship(). Unlike that flag, Session.enable_relationship_loading() allows an object to remain transient while still being able to load related items.

To make a transient object associated with a Session via Session.enable_relationship_loading() pending, add it to the Session using Session.add() normally.

Session.enable_relationship_loading() 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 method is not intended for general use.

New in version 0.8.

See also:

load_on_pending at relationship() - this flag allows per-relationship loading of many-to-ones on items that are pending.

execute(*clause*, *params=None*, *mapper=None*, *bind=None*, ***kw*)

Execute a SQL expression construct or string statement within the current transaction.

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

E.g.:

```
result = session.execute(
            user_table.select().where(user_table.c.id == 5)
        )
```

execute() accepts any executable clause construct, such as select(), insert(), update(), delete(), and text(). Plain SQL strings can be passed as well, which in the case of Session.execute() only will be interpreted the same as if it were passed via a text() construct. That is, the following usage:

```
result = session.execute(
            "SELECT * FROM user WHERE id=:param",
            {"param":5}
        )
```

is equivalent to:

```
from sqlalchemy import text
result = session.execute(
            text("SELECT * FROM user WHERE id=:param"),
            {"param":5}
        )
```

The second positional argument to Session.execute() is an optional parameter set. Similar to that of Connection.execute(), whether this is passed as a single dictionary, or a list of dictionaries, determines whether the DBAPI cursor's execute() or executemany() is used to execute the statement. An INSERT construct may be invoked for a single row:

```
result = session.execute(
    users.insert(), {"id": 7, "name": "somename"})
```

or for multiple rows:

```
result = session.execute(users.insert(), [
                        {"id": 7, "name": "somename7"},
                        {"id": 8, "name": "somename8"},
                        {"id": 9, "name": "somename9"}
                    ])
```

The statement is executed within the current transactional context of this Session. The Connection which is used to execute the statement can also be acquired directly by calling

the Session.connection() method. Both methods use a rule-based resolution scheme in order to determine the Connection, which in the average case is derived directly from the "bind" of the Session itself, and in other cases can be based on the mapper() and Table objects passed to the method; see the documentation for Session.get_bind() for a full description of this scheme.

The Session.execute() method does *not* invoke autoflush.

The ResultProxy returned by the Session.execute() method is returned with the "close_with_result" flag set to true; the significance of this flag is that if this Session is autocommitting and does not have a transaction-dedicated Connection available, a temporary Connection is established for the statement execution, which is closed (meaning, returned to the connection pool) when the ResultProxy has consumed all available data. This applies *only* when the Session is configured with autocommit=True and no transaction has been started.

Parameters

- **clause** – An executable statement (i.e. an Executable expression such as expression.select()) or string SQL statement to be executed.
- **params** – Optional dictionary, or list of dictionaries, containing bound parameter values. If a single dictionary, single-row execution occurs; if a list of dictionaries, an "executemany" will be invoked. The keys in each dictionary must correspond to parameter names present in the statement.
- **mapper** – Optional mapper() or mapped class, used to identify the appropriate bind. This argument takes precedence over clause when locating a bind. See Session.get_bind() for more details.
- **bind** – Optional Engine to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over mapper and clause when locating a bind.
- ****kw** – Additional keyword arguments are sent to Session.get_bind() to allow extensibility of "bind" schemes.

See also:

sqlexpression_toplevel - Tutorial on using Core SQL constructs.

connections_toplevel - Further information on direct statement execution.

Connection.execute() - core level statement execution method, which is Session.execute() ultimately uses in order to execute the statement.

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.

See also:

session_expire - introductory material

`Session.expire()`

`Session.refresh()`

`expire_all()`
Expires all persistent instances within this Session.

When any attributes on a persistent 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.

See also:

session_expire - introductory material

`Session.expire()`

`Session.refresh()`

`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, unless an error occurs, in which case the entire transaction is rolled back. 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 into the flush.

Parameters `objects` – Optional; restricts the flush operation to operate only on elements that are in the given collection.

This feature is for an extremely narrow set of use cases where particular objects may need to be operated upon before the full flush() occurs. It is not intended for general use.

`get_bind(`*mapper=None*, *clause=None*`)`
Return a "bind" to which this `Session` is bound.

The "bind" is usually an instance of `Engine`, except in the case where the `Session` has been explicitly bound directly to a `Connection`.

For a multiply-bound or unbound `Session`, the `mapper` or `clause` arguments are used to determine the appropriate bind to return.

Note that the "mapper" argument is usually present when `Session.get_bind()` is called via an ORM operation such as a `Session.query()`, each individual INSERT/UPDATE/DELETE operation within a `Session.flush()`, call, etc.

The order of resolution is:

1. if mapper given and session.binds is present, locate a bind based on mapper.
2. if clause given and session.binds is present, locate a bind based on `Table` objects found in the given clause present in session.binds.
3. if session.bind is present, return that.
4. if clause given, attempt to return a bind linked to the `MetaData` ultimately associated with the clause.
5. if mapper given, attempt to return a bind linked to the `MetaData` ultimately associated with the `Table` or other selectable to which the mapper is mapped.
6. No bind can be found, `UnboundExecutionError` is raised.

Parameters

- `mapper` – Optional `mapper()` mapped class or instance of `Mapper`. The bind can be derived from a `Mapper` first by consulting the "binds" map associated with this `Session`, and secondly by consulting the `MetaData` associated with the `Table` to which the `Mapper` is mapped for a bind.
- `clause` – A `ClauseElement` (i.e. `select()`, `text()`, etc.). If the `mapper` argument is not present or could not produce a bind, the given expression construct will be searched for a bound element, typically a `Table` associated with bound `MetaData`.

`identity_key`(*orm_util*, **args*, ***kwargs*)
Return an identity key.

This is an alias of `util.identity_key()`.

`identity_map = None`
A mapping of object identities to objects themselves.

Iterating through `Session.identity_map.values()` provides access to the full set of persistent objects (i.e., those that have row identity) currently in the session.

See also:

`identity_key()` - helper function to produce the keys used in this dictionary.

`info`
A user-modifiable dictionary.

The initial value of this dictionary can be populated using the `info` argument to the `Session` constructor or `sessionmaker` constructor or factory methods. The dictionary here is always local to this `Session` and can be modified independently of all other `Session` objects.

New in version 0.9.0.

`invalidate()`
Close this Session, using connection invalidation.

This is a variant of `Session.close()` that will additionally ensure that the `Connection.invalidate()` method will be called on all `Connection` objects. This can be called when the database is known to be in a state where the connections are no longer safe to be used.

E.g.:

```
try:
    sess = Session()
    sess.add(User())
    sess.commit()
except gevent.Timeout:
    sess.invalidate()
    raise
except:
    sess.rollback()
    raise
```

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.

New in version 0.9.9.

`is_active`
True if this `Session` is in "transaction mode" and is not in "partial rollback" state.

The `Session` in its default mode of `autocommit=False` is essentially always in "transaction mode", in that a `SessionTransaction` is associated with it as soon as it is instantiated. This `SessionTransaction` is immediately replaced with a new one as soon as it is ended, due to a rollback, commit, or close operation.

"Transaction mode" does *not* indicate whether or not actual database connection resources are in use; the `SessionTransaction` object coordinates among zero or more actual database transactions, and starts out with none, accumulating individual DBAPI connections as different data sources are used within its scope. The best way to track when a particular `Session` has actually begun to use DBAPI resources is to implement a listener using the `SessionEvents.after_begin()` method, which will deliver both the `Session` as well as the target `Connection` to a user-defined event listener.

The "partial rollback" state refers to when an "inner" transaction, typically used during a flush, encounters an error and emits a rollback of the DBAPI connection. At this point, the `Session` is in "partial rollback" and awaits for the user to call `Session.rollback()`, in order to close out the transaction stack. It is in this "partial rollback" period that the `is_active` flag returns False. After the call to `Session.rollback()`, the `SessionTransaction` is replaced with a new one and `is_active` returns `True` again.

When a `Session` is used in `autocommit=True` mode, the `SessionTransaction` is only instantiated within the scope of a flush call, or when `Session.begin()` is called. So `is_active` will always be `False` outside of a flush or `Session.begin()` block in this mode, and will be `True` within the `Session.begin()` block as long as it doesn't enter "partial rollback" state.

From all the above, it follows that the only purpose to this flag is for application frameworks that wish to detect is a "rollback" is necessary within a generic error handling routine, for `Session` objects that would otherwise be in "partial rollback" mode. In a typical integration case, this is also not necessary as it is standard practice to emit `Session.rollback()` unconditionally within the outermost exception catch.

To track the transactional state of a `Session` fully, use event listeners, primarily the `SessionEvents.after_begin()`, `SessionEvents.after_commit()`, `SessionEvents.after_rollback()` and related events.

`is_modified`(*instance*, *include_collections=True*, *passive=True*)
Return `True` if the given instance has locally modified attributes.

This method retrieves the history for each instrumented attribute on the instance and performs a comparison of the current value to its previously committed value, if any.

It is in effect a more expensive and accurate version of checking for the given instance in the `Session.dirty` collection; a full test for each attribute's net "dirty" status is performed.

E.g.:

```
return session.is_modified(someobject)
```

Changed in version 0.8: When using SQLAlchemy 0.7 and earlier, the `passive` flag should **always** be explicitly set to `True`, else SQL loads/autoflushes may proceed which can affect the modified state itself: `session.is_modified(someobject, passive=True)`. In 0.8 and above, the behavior is corrected and this flag is ignored.

A few caveats to this method apply:

- Instances present in the `Session.dirty` collection may report `False` when tested with this method. This is because the object may have received change events via attribute mutation, thus placing it in `Session.dirty`, but ultimately the state is the same as that loaded from the database, resulting in no net change here.
- 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 upon set only if the attribute container has the `active_history` flag set to `True`. This flag is set typically for primary key attributes and scalar object references that are not a simple many-to-one. To set this flag for any arbitrary mapped column, use the `active_history` argument with `column_property()`.

  Parameters

  - **`instance`** – mapped instance to be tested for pending changes.
  - **`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.
  - **`passive`** – Changed in version 0.8: Ignored for backwards compatibility. When using SQLAlchemy 0.7 and earlier, this flag should always be set to `True`.

`merge`(*instance*, *load=True*)

Copy the state of a given instance into a corresponding instance within this `Session`.

`Session.merge()` examines the primary key attributes of the source instance, and attempts to reconcile it with an instance of the same primary key in the session. If not found locally, it attempts to load the object from the database based on primary key, and if none can be located, creates a new instance. The state of each attribute on the source instance is then copied to the target instance. The resulting target instance is then returned by the method; the original source instance is left unmodified, and un-associated with the `Session` if not already.

This operation cascades to associated instances if the association is mapped with `cascade="merge"`.

See unitofwork_merging for a detailed discussion of merging.

Changed in version 1.1: - `Session.merge()` will now reconcile pending objects with overlapping primary keys in the same way as persistent. See change_3601 for discussion.

Parameters

- **instance** – Instance to be merged.
- **load** – Boolean, when False, `merge()` switches into a "high performance" mode which causes it to forego emitting history events as well as all database access. This flag is used for cases such as transferring graphs of objects into a `Session` from a second level cache, or to transfer just-loaded objects into the `Session` owned by a worker thread or process without re-querying the database.

  The `load=False` use case adds the caveat that the given object has to be in a "clean" state, that is, has no pending changes to be flushed - even if the incoming object is detached from any `Session`. This is so that when the merge operation populates local attributes and cascades to related objects and collections, the values can be "stamped" onto the target object as is, without generating any history or attribute events, and without the need to reconcile the incoming data with any existing related objects or collections that might not be loaded. The resulting objects from `load=False` are always produced as "clean", so it is only appropriate that the given objects should be "clean" as well, else this suggests a mis-use of the method.

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

no_autoflush
Return a context manager that disables autoflush.

e.g.:

```
with session.no_autoflush:

    some_object = SomeClass()
    session.add(some_object)
    # won't autoflush
    some_object.related_thing = session.query(SomeRelated).first()
```

Operations that proceed within the `with:` block will not be subject to flushes occurring upon query access. This is useful when initializing a series of objects which involve existing database queries, where the uncompleted object should not yet be flushed.

New in version 0.7.6.

object_session(*instance*)
Return the `Session` to which an object belongs.

This is an alias of `object_session()`.

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.

Deprecated since version 0.7: The non-weak-referencing identity map feature is no longer needed.

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*, *with_for_update=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.
- `with_for_update` – optional boolean `True` indicating FOR UPDATE should be used, or may be a dictionary containing flags to indicate a more specific set of FOR UPDATE flags for the SELECT; flags should match the parameters of `Query.with_for_update()`. Supersedes the `Session.refresh.lockmode` parameter.

  New in version 1.2.
- `lockmode` – Passed to the `Query` as used by `with_lockmode()`. Superseded by `Session.refresh.with_for_update`.

See also:

session_expire - introductory material

`Session.expire()`

`Session.expire_all()`

`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.

See also:

session_rollback

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

`transaction = None`
The current active or inactive `SessionTransaction`.

`class sqlalchemy.orm.session.SessionTransaction`(*session*, *parent=None*, *nested=False*)
A `Session`-level transaction.

`SessionTransaction` is a mostly behind-the-scenes object not normally referenced directly by application code. It coordinates among multiple `Connection` objects, maintaining a database transaction for each one individually, committing or rolling them back all at once. It also provides optional two-phase commit behavior which can augment this coordination operation.

The `Session.transaction` attribute of `Session` refers to the current `SessionTransaction` object in use, if any. The `SessionTransaction.parent` attribute refers to the parent `SessionTransaction` in the stack of `SessionTransaction` objects. If this attribute is `None`, then

this is the top of the stack. If non-None, then this SessionTransaction refers either to a so-called "subtransaction" or a "nested" transaction. A "subtransaction" is a scoping concept that demarcates an inner portion of the outermost "real" transaction. A nested transaction, which is indicated when the SessionTransaction.nested attribute is also True, indicates that this SessionTransaction corresponds to a SAVEPOINT.

Life Cycle

A SessionTransaction is associated with a Session in its default mode of autocommit=False immediately, associated with no database connections. As the Session is called upon to emit SQL on behalf of various Engine or Connection objects, a corresponding Connection and associated Transaction is added to a collection within the SessionTransaction object, becoming one of the connection/transaction pairs maintained by the SessionTransaction. The start of a SessionTransaction can be tracked using the SessionEvents.after_transaction_create() event.

The lifespan of the SessionTransaction ends when the Session.commit(), Session.rollback() or Session.close() methods are called. At this point, the SessionTransaction removes its association with its parent Session. A Session that is in autocommit=False mode will create a new SessionTransaction to replace it immediately, whereas a Session that's in autocommit=True mode will remain without a SessionTransaction until the Session.begin() method is called. The end of a SessionTransaction can be tracked using the SessionEvents.after_transaction_end() event.

Nesting and Subtransactions

Another detail of SessionTransaction behavior is that it is capable of "nesting". This means that the Session.begin() method can be called while an existing SessionTransaction is already present, producing a new SessionTransaction that temporarily replaces the parent SessionTransaction. When a SessionTransaction is produced as nested, it assigns itself to the Session.transaction attribute, and it additionally will assign the previous SessionTransaction to its Session.parent attribute. The behavior is effectively a stack, where Session.transaction refers to the current head of the stack, and the SessionTransaction.parent attribute allows traversal up the stack until SessionTransaction.parent is None, indicating the top of the stack.

When the scope of SessionTransaction is ended via Session.commit() or Session.rollback(), it restores its parent SessionTransaction back onto the Session.transaction attribute.

The purpose of this stack is to allow nesting of Session.rollback() or Session.commit() calls in context with various flavors of Session.begin(). This nesting behavior applies to when Session.begin_nested() is used to emit a SAVEPOINT transaction, and is also used to produce a so-called "subtransaction" which allows a block of code to use a begin/rollback/commit sequence regardless of whether or not its enclosing code block has begun a transaction. The flush() method, whether called explicitly or via autoflush, is the primary consumer of the "subtransaction" feature, in that it wishes to guarantee that it works within in a transaction block regardless of whether or not the Session is in transactional mode when the method is called.

Note that the flush process that occurs within the "autoflush" feature as well as when the Session.flush() method is used **always** creates a SessionTransaction object. This object is normally a subtransaction, unless the Session is in autocommit mode and no transaction exists at all, in which case it's the outermost transaction. Any event-handling logic or other inspection logic needs to take into account whether a SessionTransaction is the outermost transaction, a subtransaction, or a "nested" / SAVEPOINT transaction.

See also:

Session.rollback()

Session.commit()

Session.begin()

Session.begin_nested()

Session.is_active

SessionEvents.after_transaction_create()

SessionEvents.after_transaction_end()

SessionEvents.after_commit()

SessionEvents.after_rollback()

SessionEvents.after_soft_rollback()

nested = False
Indicates if this is a nested, or SAVEPOINT, transaction.

When SessionTransaction.nested is True, it is expected that SessionTransaction.parent will be True as well.

parent
The parent SessionTransaction of this SessionTransaction.

If this attribute is None, indicates this SessionTransaction is at the top of the stack, and corresponds to a real "COMMIT"/"ROLLBACK" block. If non-None, then this is either a "subtransaction" or a "nested" / SAVEPOINT transaction. If the SessionTransaction.nested attribute is True, then this is a SAVEPOINT, and if False, indicates this a subtransaction.

New in version 1.0.16: - use ._parent for previous versions

Session Utilities

sqlalchemy.orm.session.**make_transient**(*instance*)
Alter the state of the given instance so that it is transient.

Note: make_transient() is a special-case function for advanced use cases only.

The given mapped instance is assumed to be in the persistent or detached state. The function will remove its association with any Session as well as its InstanceState.identity. The effect is that the object will behave as though it were newly constructed, except retaining any attribute / collection values that were loaded at the time of the call. The InstanceState.deleted flag is also reset if this object had been deleted as a result of using Session.delete().

Warning: make_transient() does **not** "unexpire" or otherwise eagerly load ORM-mapped attributes that are not currently loaded at the time the function is called. This includes attributes which:

- were expired via Session.expire()
- were expired as the natural effect of committing a session transaction, e.g. Session.commit()
- are normally lazy loaded but are not currently loaded
- are "deferred" via deferred and are not yet loaded
- were not present in the query which loaded this object, such as that which is common in joined table inheritance and other scenarios.

After make_transient() is called, unloaded attributes such as those above will normally resolve to the value None when accessed, or an empty collection for a collection-oriented attribute. As the object is transient and un-associated with any database identity, it will no longer retrieve these values.

See also:

make_transient_to_detached()

sqlalchemy.orm.session.**make_transient_to_detached**(*instance*)
Make the given transient instance detached.

> **Note:** `make_transient_to_detached()` is a special-case function for advanced use cases only.

All attribute history on the given instance will be reset as though the instance were freshly loaded from a query. Missing attributes will be marked as expired. The primary key attributes of the object, which are required, will be made into the "key" of the instance.

The object can then be added to a session, or merged possibly with the load=False flag, at which point it will look as if it were loaded that way, without emitting SQL.

This is a special use case function that differs from a normal call to `Session.merge()` in that a given persistent state can be manufactured without any SQL calls.

New in version 0.9.5.

See also:

`make_transient()`

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

This is essentially the same as the `InstanceState.session` accessor. See that attribute for details.

sqlalchemy.orm.util.**was_deleted**(*object*)
Return True if the given object was deleted within a session flush.

This is regardless of whether or not the object is persistent or detached.

New in version 0.8.0.

See also:

`InstanceState.was_deleted`

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 Events*.

sqlalchemy.orm.util.**object_state**(*instance*)
Given an object, return the `InstanceState` associated with the object.

Raises `sqlalchemy.orm.exc.UnmappedInstanceError` if no mapping is configured.

Equivalent functionality is available via the `inspect()` function as:

```
inspect(instance)
```

Using the inspection system will raise `sqlalchemy.exc.NoInspectionAvailable` if the instance is not part of a mapping.

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*, *passive=symbol('PASSIVE_OFF')*)
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.
- **passive** – indicates loading behavior for the attribute if the value is not already present. This is a bitflag attribute, which defaults to the symbol `PASSIVE_OFF` indicating all necessary SQL should be emitted.

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.**flag_modified**(*instance*, *key*)
Mark an attribute on an instance as 'modified'.

This sets the 'modified' flag on the instance and establishes an unconditional change event for the given attribute. The attribute must have a value present, else an `InvalidRequestError` is raised.

To mark an object "dirty" without referring to any specific attribute so that it is considered within a flush, use the `attributes.flag_dirty()` call.

See also:

`attributes.flag_dirty()`

sqlalchemy.orm.attributes.**flag_dirty**(*instance*)
Mark an instance as 'dirty' without any specific attribute mentioned.

This is a special operation that will allow the object to travel through the flush process for interception by events such as `SessionEvents.before_flush()`. Note that no SQL will be emitted in the flush process for an object that has no changes, even if marked dirty via this method. However, a `SessionEvents.before_flush()` handler will be able to see the object in the `Session.dirty` collection and may establish changes on it, which will then be included in the SQL emitted.

New in version 1.2.

See also:

`attributes.flag_modified()`

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

This function is the internal version of `object_state()`. The `object_state()` and/or the `inspect()` function is preferred here as they each emit an informative exception if the given object is not mapped.

sqlalchemy.orm.instrumentation.**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.**set_attribute**(*instance*, *key*, *value*, *initiator=None*)
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.

Parameters

- **instance** – the object that will be modified
- **key** – string name of the attribute
- **value** – value to assign
- **initiator** – an instance of `Event` that would have been propagated from a previous event listener. This argument is used when the `set_attribute()` function is being used within an existing event listening function where an `Event` object is being supplied; the object may be used to track the origin of the chain of events.

  New in version 1.2.3.

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 occurred on an instrumented attribute.

The easiest way to get a `History` object for a particular attribute on an object is to use the `inspect()` function:

```
from sqlalchemy import inspect

hist = inspect(myobject).attrs.myattribute.history
```

Each tuple member is an iterable sequence:

- `added` - the collection of items added to the attribute (the first tuple element).
- `unchanged` - the collection of items that have not changed on the attribute (the second tuple element).
- `deleted` - 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.

2.6 Events and Internals

2.6.1 ORM Events

The ORM includes a wide variety of hooks available for subscription.

For an introduction to the most commonly used ORM events, see the section session_events_toplevel. The event system in general is discussed at event_toplevel. Non-ORM events such as those regarding connections and low-level statement execution are described in core_event_toplevel.

Attribute Events

`class sqlalchemy.orm.events.AttributeEvents`
Define events for object attributes.

These are typically defined on the class-bound descriptor for the target class.

e.g.:

```
from sqlalchemy import event

def my_append_listener(target, value, initiator):
    print "received append event for target: %s" % target

event.listen(MyClass.collection, 'append', my_append_listener)
```

Listeners have the option to return a possibly modified version of the value, when the `retval=True` flag is passed to `listen()`:

```
def validate_phone(target, value, oldvalue, initiator):
    "Strip non-numeric characters from a phone number"

    return re.sub(r'\D', '', value)

# setup listener on UserContact.phone attribute, instructing
# it to use the return value
listen(UserContact.phone, 'set', validate_phone, retval=True)
```

A validation function like the above can also raise an exception such as `ValueError` to halt the operation.

Several modifiers are available to the `listen()` function.

Parameters

- **`active_history=False`** – When True, indicates that the "set" event would like to receive the "old" value being replaced unconditionally, even if this requires firing off database loads. Note that `active_history` can also be set directly via `column_property()` and `relationship()`.
- **`propagate=False`** – When True, the listener function will be established not just for the class attribute given, but for attributes of the same name on all current subclasses of that class, as well as all future subclasses of that class, using an additional listener that listens for instrumentation events.

- **raw=False** – When True, the "target" argument to the event will be the `InstanceState` management object, rather than the mapped instance itself.
- **retval=False** – when True, the user-defined event listening must return the "value" argument from the function. This gives the listening function the opportunity to change the value that is ultimately used for a "set" or "append" event.

append(*target*, *value*, *initiator*)
Receive a collection append event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'append')
def receive_append(target, value, initiator):
    "listen for the 'append' event"

    # ... (event handling logic) ...
```

The append event is invoked for each element as it is appended to the collection. This occurs for single-item appends as well as for a "bulk replace" operation.

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the `InstanceState` object.
- **value** – the value being appended. If this listener is registered with `retval=True`, the listener function must return this value, or a new value which replaces it.
- **initiator** – An instance of `attributes.Event` representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation, as well as be inspected for information about the source of the event.

Returns if the event was registered with `retval=True`, the given value, or a new effective value, should be returned.

See also:

`AttributeEvents.bulk_replace()`

bulk_replace(*target*, *values*, *initiator*)
Receive a collection 'bulk replace' event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'bulk_replace')
def receive_bulk_replace(target, values, initiator):
    "listen for the 'bulk_replace' event"

    # ... (event handling logic) ...
```

This event is invoked for a sequence of values as they are incoming to a bulk collection set operation, which can be modified in place before the values are treated as ORM objects. This is an "early hook" that runs before the bulk replace routine attempts to reconcile which objects are already present in the collection and which are being removed by the net replace operation.

It is typical that this method be combined with use of the `AttributeEvents.append()` event. When using both of these events, note that a bulk replace operation will invoke the `AttributeEvents.append()` event for all new items, even after `AttributeEvents.bulk_replace()` has been invoked for the collection as a whole. In order to determine if an `AttributeEvents.append()` event is part of a bulk replace, use the symbol `OP_BULK_REPLACE` to test the incoming initiator:

```
from sqlalchemy.orm.attributes import OP_BULK_REPLACE

@event.listens_for(SomeObject.collection, "bulk_replace")
def process_collection(target, values, initiator):
    values[:] = [_make_value(value) for value in values]

@event.listens_for(SomeObject.collection, "append", retval=True)
def process_collection(target, value, initiator):
    # make sure bulk_replace didn't already do it
    if initiator is None or initiator.op is not OP_BULK_REPLACE:
        return _make_value(value)
    else:
        return value
```

New in version 1.2.

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the `InstanceState` object.
- **value** – a sequence (e.g. a list) of the values being set. The handler can modify this list in place.
- **initiator** – An instance of `attributes.Event` representing the initiation of the event.

dispose_collection(*target*, *collection*, *collection_adpater*)

Receive a 'collection dispose' event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'dispose_collection')
def receive_dispose_collection(target, collection, collection_adpater):
    "listen for the 'dispose_collection' event"

    # ... (event handling logic) ...
```

This event is triggered for a collection-based attribute when a collection is replaced, that is:

```
u1.addresses.append(a1)

u1.addresses = [a2, a3]  # <- old collection is disposed
```

The old collection received will contain its previous contents.

Changed in version 1.2: The collection passed to `AttributeEvents.dispose_collection()` will now have its contents before the dispose intact; previously, the collection would be empty.

New in version 1.0.0: the `AttributeEvents.init_collection()` and `AttributeEvents.dispose_collection()` events supersede the `collection.linker` hook.

init_collection(*target*, *collection*, *collection_adapter*)

Receive a 'collection init' event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'init_collection')
def receive_init_collection(target, collection, collection_adapter):
    "listen for the 'init_collection' event"

    # ... (event handling logic) ...
```

This event is triggered for a collection-based attribute, when the initial "empty collection" is first generated for a blank attribute, as well as for when the collection is replaced with a new one, such as via a set event.

E.g., given that `User.addresses` is a relationship-based collection, the event is triggered here:

```
u1 = User()
u1.addresses.append(a1)  #  <- new collection
```

and also during replace operations:

```
u1.addresses = [a2, a3]  #  <- new collection
```

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the `InstanceState` object.
- **collection** – the new collection. This will always be generated from what was specified as `RelationshipProperty.collection_class`, and will always be empty.
- **collection_adpater** – the `CollectionAdapter` that will mediate internal access to the collection.

New in version 1.0.0: the `AttributeEvents.init_collection()` and `AttributeEvents.dispose_collection()` events supersede the `collection.linker` hook.

init_scalar(*target*, *value*, *dict_*)
Receive a scalar "init" event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'init_scalar')
def receive_init_scalar(target, value, dict_):
    "listen for the 'init_scalar' event"

    # ... (event handling logic) ...
```

This event is invoked when an uninitialized, unpersisted scalar attribute is accessed. A value of `None` is typically returned in this case; no changes are made to the object's state.

The event handler can alter this behavior in two ways. One is that a value other than `None` may be returned. The other is that the value may be established as part of the object's state, which will also have the effect that it is persisted.

Typical use is to establish a specific default value of an attribute upon access:

```
SOME_CONSTANT = 3.1415926

@event.listens_for(
```

```
    MyClass.some_attribute, "init_scalar",
    retval=True, propagate=True)
def _init_some_attribute(target, dict_, value):
    dict_['some_attribute'] = SOME_CONSTANT
    return SOME_CONSTANT
```

Above, we initialize the attribute `MyClass.some_attribute` to the value of `SOME_CONSTANT`. The above code includes the following features:

- By setting the value `SOME_CONSTANT` in the given `dict_`, we indicate that the value is to be persisted to the database. **The given value is only persisted to the database if we explicitly associate it with the object**. The `dict_` given is the `__dict__` element of the mapped object, assuming the default attribute instrumentation system is in place.
- By establishing the `retval=True` flag, the value we return from the function will be returned by the attribute getter. Without this flag, the event is assumed to be a passive observer and the return value of our function is ignored.
- The `propagate=True` flag is significant if the mapped class includes inheriting subclasses, which would also make use of this event listener. Without this flag, an inheriting subclass will not use our event handler.

When we establish the value in the given dictionary, the value will be used in the INSERT statement established by the unit of work. Normally, the default returned value of `None` is not established as part of the object, to avoid the issue of mutations occurring to the object in response to a normally passive "get" operation, and also sidesteps the issue of whether or not the `AttributeEvents.set()` event should be awkwardly fired off during an attribute access operation. This does not impact the INSERT operation since the `None` value matches the value of `NULL` that goes into the database in any case; note that `None` is skipped during the INSERT to ensure that column and SQL-level default functions can fire off.

The attribute set event `AttributeEvents.set()` as well as the related validation feature provided by `orm.validates` is **not** invoked when we apply our value to the given `dict_`. To have these events to invoke in response to our newly generated value, apply the value to the given object as a normal attribute set operation:

```
SOME_CONSTANT = 3.1415926

@event.listens_for(
    MyClass.some_attribute, "init_scalar",
    retval=True, propagate=True)
def _init_some_attribute(target, dict_, value):
    # will also fire off attribute set events
    target.some_attribute = SOME_CONSTANT
    return SOME_CONSTANT
```

When multiple listeners are set up, the generation of the value is "chained" from one listener to the next by passing the value returned by the previous listener that specifies `retval=True` as the `value` argument of the next listener.

The `AttributeEvents.init_scalar()` event may be used to extract values from the default values and/or callables established on mapped `Column` objects. See the "active column defaults" example in examples_instrumentation for an example of this.

New in version 1.1.

Parameters

- **`target`** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the `InstanceState` object.
- **`value`** – the value that is to be returned before this event listener were invoked. This value begins as the value `None`, however will be the return value of the previous event handler function if multiple listeners are present.

- **dict_** – the attribute dictionary of this mapped object. This is normally the `__dict__` of the object, but in all cases represents the destination that the attribute system uses to get at the actual value of this attribute. Placing the value in this dictionary has the effect that the value will be used in the INSERT statement generated by the unit of work.

See also:

examples_instrumentation - see the `active_column_defaults.py` example.

modified(*target*, *initiator*)

Receive a 'modified' event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'modified')
def receive_modified(target, initiator):
    "listen for the 'modified' event"

    # ... (event handling logic) ...
```

This event is triggered when the `attributes.flag_modified()` function is used to trigger a modify event on an attribute without any specific value being set.

New in version 1.2.

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the `InstanceState` object.
- **initiator** – An instance of `attributes.Event` representing the initiation of the event.

remove(*target*, *value*, *initiator*)

Receive a collection remove event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'remove')
def receive_remove(target, value, initiator):
    "listen for the 'remove' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the `InstanceState` object.
- **value** – the value being removed.
- **initiator** – An instance of `attributes.Event` representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.

  Changed in version 0.9.0: the `initiator` argument is now passed as a `attributes.Event` object, and may be modified by backref handlers within a chain of backref-linked events.

Returns No return value is defined for this event.

set(*target*, *value*, *oldvalue*, *initiator*)
Receive a scalar set event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'set')
def receive_set(target, value, oldvalue, initiator):
    "listen for the 'set' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeClass.some_attribute, 'set', named=True)
def receive_set(**kw):
    "listen for the 'set' event"
    target = kw['target']
    value = kw['value']

    # ... (event handling logic) ...
```

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the `InstanceState` object.
- **value** – the value being set. If this listener is registered with `retval=True`, the listener function must return this value, or a new value which replaces it.
- **oldvalue** – the previous value being replaced. This may also be the symbol `NEVER_SET` or `NO_VALUE`. If the listener is registered with `active_history=True`, the previous value of the attribute will be loaded from the database if the existing value is currently unloaded or expired.
- **initiator** – An instance of `attributes.Event` representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.

  Changed in version 0.9.0: the `initiator` argument is now passed as a `attributes.Event` object, and may be modified by backref handlers within a chain of backref-linked events.

Returns if the event was registered with `retval=True`, the given value, or a new effective value, should be returned.

Mapper Events

class sqlalchemy.orm.events.**MapperEvents**
Define events specific to mappings.

e.g.:

```
from sqlalchemy import event

def my_before_insert_listener(mapper, connection, target):
    # execute a stored procedure upon INSERT,
    # apply the value to the row to be inserted
    target.calculated_value = connection.scalar(
                                "select my_special_function(%d)"
                                % target.special_number)
```

```
# associate the listener function with SomeClass,
# to execute during the "before_insert" hook
event.listen(
    SomeClass, 'before_insert', my_before_insert_listener)
```

Available targets include:

- mapped classes
- unmapped superclasses of mapped or to-be-mapped classes (using the `propagate=True` flag)
- `Mapper` objects
- the `Mapper` class itself and the `mapper()` function indicate listening for all mappers.

Changed in version 0.8.0: mapper events can be associated with unmapped superclasses of mapped classes.

Mapper events provide hooks into critical sections of the mapper, including those related to object instrumentation, object loading, and object persistence. In particular, the persistence methods `before_insert()`, and `before_update()` are popular places to augment the state being persisted - however, these methods operate with several significant restrictions. The user is encouraged to evaluate the `SessionEvents.before_flush()` and `SessionEvents.after_flush()` methods as more flexible and user-friendly hooks in which to apply additional database state during a flush.

When using `MapperEvents`, several modifiers are available to the `event.listen()` function.

Parameters

- **`propagate=False`** – When True, the event listener should be applied to all inheriting mappers and/or the mappers of inheriting classes, as well as any mapper which is the target of this listener.
- **`raw=False`** – When True, the "target" argument passed to applicable event listener functions will be the instance's `InstanceState` management object, rather than the mapped instance itself.
- **`retval=False`** – when True, the user-defined event function must have a return value, the purpose of which is either to control subsequent event propagation, or to otherwise alter the operation in progress by the mapper. Possible return values are:
  - `sqlalchemy.orm.interfaces.EXT_CONTINUE` - continue event processing normally.
  - `sqlalchemy.orm.interfaces.EXT_STOP` - cancel all subsequent event handlers in the chain.
  - other values - the return value specified by specific listeners.

`after_configured()`

Called after a series of mappers have been configured.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_configured')
def receive_after_configured():
    "listen for the 'after_configured' event"

    # ... (event handling logic) ...
```

The `MapperEvents.after_configured()` event is invoked each time the `orm.configure_mappers()` function is invoked, after the function has completed its work.

orm.configure_mappers() is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

Contrast this event to the MapperEvents.mapper_configured() event, which is called on a per-mapper basis while the configuration operation proceeds; unlike that event, when this event is invoked, all cross-configurations (e.g. backrefs) will also have been made available for any mappers that were pending. Also contrast to MapperEvents.before_configured(), which is invoked before the series of mappers has been configured.

This event can **only** be applied to the Mapper class or mapper() function, and not to individual mappings or mapped classes. It is only invoked for all mappings as a whole:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "after_configured")
def go():
    # ...
```

Theoretically this event is called once per application, but is actually called any time new mappers have been affected by a orm.configure_mappers() call. If new mappings are constructed after existing ones have already been used, this event will likely be called again. To ensure that a particular event is only called once and no further, the once=True argument (new in 0.9.4) can be applied:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "after_configured", once=True)
def go():
    # ...
```

See also:

MapperEvents.mapper_configured()

MapperEvents.before_configured()

after_delete(*mapper*, *connection*, *target*)

Receive an object instance after a DELETE statement has been emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_delete')
def receive_after_delete(mapper, connection, target):
    "listen for the 'after_delete' event"

    # ... (event handling logic) ...
```

This event is used to emit additional SQL statements on the given connection as well as to perform application specific bookkeeping related to a deletion event.

The event is often called for a batch of objects of the same class after their DELETE statements have been emitted at once in a previous step.

Warning: Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given Connection. **Please read fully** the notes at session_persistence_mapper for guidelines on using these methods; generally, the SessionEvents.before_flush() method should be preferred for general on-flush changes.

Parameters

- **mapper** – the Mapper which is the target of this event.
- **connection** – the Connection being used to emit DELETE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being deleted. If the event is configured with raw=True, this will instead be the InstanceState state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

session_persistence_events

after_insert(*mapper*, *connection*, *target*)

Receive an object instance after an INSERT statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_insert')
def receive_after_insert(mapper, connection, target):
    "listen for the 'after_insert' event"

    # ... (event handling logic) ...
```

This event is used to modify in-Python-only state on the instance after an INSERT occurs, as well as to emit additional SQL statements on the given connection.

The event is often called for a batch of objects of the same class after their INSERT statements have been emitted at once in a previous step. In the extremely rare case that this is not desirable, the mapper() can be configured with batch=False, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

Warning: Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given Connection. **Please read fully** the notes at session_persistence_mapper for guidelines on using these methods; generally, the SessionEvents.before_flush() method should be preferred for general on-flush changes.

Parameters

- **mapper** – the Mapper which is the target of this event.
- **connection** – the Connection being used to emit INSERT statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with raw=True, this will instead be the InstanceState state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

session_persistence_events

after_update(*mapper*, *connection*, *target*)

Receive an object instance after an UPDATE statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_update')
def receive_after_update(mapper, connection, target):
    "listen for the 'after_update' event"

    # ... (event handling logic) ...
```

This event is used to modify in-Python-only state on the instance after an UPDATE occurs, as well as to emit additional SQL statements on the given connection.

This method is called for all instances that are marked as "dirty", *even those which have no net changes to their column-based attributes*, and for which no UPDATE statement has proceeded. 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 `after_update()` is *not* a guarantee that an UPDATE statement has been issued.

To detect if the column-based attributes on the object have net changes, and therefore resulted in an UPDATE statement, use `object_session(instance).is_modified(instance, include_collections=False)`.

The event is often called for a batch of objects of the same class after their UPDATE statements have been emitted at once in a previous step. In the extremely rare case that this is not desirable, the `mapper()` can be configured with `batch=False`, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

Warning: Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at session_persistence_mapper for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit UPDATE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

session_persistence_events

before_configured()
Called before a series of mappers have been configured.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_configured')
def receive_before_configured():
    "listen for the 'before_configured' event"

    # ... (event handling logic) ...
```

The MapperEvents.before_configured() event is invoked each time the orm.configure_mappers() function is invoked, before the function has done any of its work. orm.configure_mappers() is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

This event can **only** be applied to the Mapper class or mapper() function, and not to individual mappings or mapped classes. It is only invoked for all mappings as a whole:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "before_configured")
def go():
    # ...
```

Contrast this event to MapperEvents.after_configured(), which is invoked after the series of mappers has been configured, as well as MapperEvents.mapper_configured(), which is invoked on a per-mapper basis as each one is configured to the extent possible.

Theoretically this event is called once per application, but is actually called any time new mappers are to be affected by a orm.configure_mappers() call. If new mappings are constructed after existing ones have already been used, this event will likely be called again. To ensure that a particular event is only called once and no further, the once=True argument (new in 0.9.4) can be applied:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "before_configured", once=True)
def go():
    # ...
```

New in version 0.9.3.

See also:

MapperEvents.mapper_configured()

MapperEvents.after_configured()

before_delete(*mapper*, *connection*, *target*)
Receive an object instance before a DELETE statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_delete')
def receive_before_delete(mapper, connection, target):
    "listen for the 'before_delete' event"
```

```
    # ... (event handling logic) ...
```

This event is used to emit additional SQL statements on the given connection as well as to perform application specific bookkeeping related to a deletion event.

The event is often called for a batch of objects of the same class before their DELETE statements are emitted at once in a later step.

Warning: Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at session_persistence_mapper for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **`mapper`** – the `Mapper` which is the target of this event.
- **`connection`** – the `Connection` being used to emit DELETE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **`target`** – the mapped instance being deleted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

session_persistence_events

`before_insert`(*mapper*, *connection*, *target*)

Receive an object instance before an INSERT statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_insert')
def receive_before_insert(mapper, connection, target):
    "listen for the 'before_insert' event"

    # ... (event handling logic) ...
```

This event is used to modify local, non-object related attributes on the instance before an INSERT occurs, as well as to emit additional SQL statements on the given connection.

The event is often called for a batch of objects of the same class before their INSERT statements are emitted at once in a later step. In the extremely rare case that this is not desirable, the `mapper()` can be configured with `batch=False`, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

Warning: Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at session_persistence_mapper

for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit INSERT statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

session_persistence_events

`before_update`(*mapper*, *connection*, *target*)

Receive an object instance before an UPDATE statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_update')
def receive_before_update(mapper, connection, target):
    "listen for the 'before_update' event"

    # ... (event handling logic) ...
```

This event is used to modify local, non-object related attributes on the instance before an UPDATE occurs, as well as to emit additional SQL statements on the given connection.

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 by modifying attributes so that a net change in value does exist.

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)`.

The event is often called for a batch of objects of the same class before their UPDATE statements are emitted at once in a later step. In the extremely rare case that this is not desirable, the `mapper()` can be configured with `batch=False`, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

Warning: Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at session_persistence_mapper

for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit UPDATE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

session_persistence_events

instrument_class(*mapper*, *class_*)

Receive a class when the mapper is first constructed, before instrumentation is applied to the mapped class.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'instrument_class')
def receive_instrument_class(mapper, class_):
    "listen for the 'instrument_class' event"

    # ... (event handling logic) ...
```

This event is the earliest phase of mapper construction. Most attributes of the mapper are not yet initialized.

This listener can either be applied to the `Mapper` class overall, or to any un-mapped class which serves as a base for classes that will be mapped (using the `propagate=True` flag):

```
Base = declarative_base()

@event.listens_for(Base, "instrument_class", propagate=True)
def on_new_class(mapper, cls_):
    " ... "
```

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **class_** – the mapped class.

mapper_configured(*mapper*, *class_*)

Called when a specific mapper has completed its own configuration within the scope of the `configure_mappers()` call.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
```

```
@event.listens_for(SomeClass, 'mapper_configured')
def receive_mapper_configured(mapper, class_):
    "listen for the 'mapper_configured' event"

    # ... (event handling logic) ...
```

The `MapperEvents.mapper_configured()` event is invoked for each mapper that is encountered when the `orm.configure_mappers()` function proceeds through the current list of not-yet-configured mappers. `orm.configure_mappers()` is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

When the event is called, the mapper should be in its final state, but **not including backrefs** that may be invoked from other mappers; they might still be pending within the configuration operation. Bidirectional relationships that are instead configured via the `orm.relationship.back_populates` argument *will* be fully available, since this style of relationship does not rely upon other possibly-not-configured mappers to know that they exist.

For an event that is guaranteed to have **all** mappers ready to go including backrefs that are defined only on other mappings, use the `MapperEvents.after_configured()` event; this event invokes only after all known mappings have been fully configured.

The `MapperEvents.mapper_configured()` event, unlike `MapperEvents.before_configured()` or `MapperEvents.after_configured()`, is called for each mapper/class individually, and the mapper is passed to the event itself. It also is called exactly once for a particular mapper. The event is therefore useful for configurational steps that benefit from being invoked just once on a specific mapper basis, which don't require that "backref" configurations are necessarily ready yet.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **class_** – the mapped class.

See also:

`MapperEvents.before_configured()`

`MapperEvents.after_configured()`

Instance Events

class sqlalchemy.orm.events.**InstanceEvents**

Define events specific to object lifecycle.

e.g.:

```
from sqlalchemy import event

def my_load_listener(target, context):
    print "on load!"

event.listen(SomeClass, 'load', my_load_listener)
```

Available targets include:

- mapped classes
- unmapped superclasses of mapped or to-be-mapped classes (using the `propagate=True` flag)
- `Mapper` objects
- the `Mapper` class itself and the `mapper()` function indicate listening for all mappers.

Changed in version 0.8.0: instance events can be associated with unmapped superclasses of mapped classes.

Instance events are closely related to mapper events, but are more specific to the instance and its instrumentation, rather than its system of persistence.

When using `InstanceEvents`, several modifiers are available to the `event.listen()` function.

Parameters

- **propagate=False** – When True, the event listener should be applied to all inheriting classes as well as the class which is the target of this listener.
- **raw=False** – When True, the "target" argument passed to applicable event listener functions will be the instance's `InstanceState` management object, rather than the mapped instance itself.

`expire`(*target*, *attrs*)

Receive an object instance after its attributes or some subset have been expired.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'expire')
def receive_expire(target, attrs):
    "listen for the 'expire' event"

    # ... (event handling logic) ...
```

'keys' is a list of attribute names. If None, the entire state was expired.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.
- **attrs** – sequence of attribute names which were expired, or None if all attributes were expired.

`first_init`(*manager*, *cls*)

Called when the first instance of a particular mapping is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'first_init')
def receive_first_init(manager, cls):
    "listen for the 'first_init' event"

    # ... (event handling logic) ...
```

This event is called when the `__init__` method of a class is called the first time for that particular class. The event invokes before `__init__` actually proceeds as well as before the `InstanceEvents.init()` event is invoked.

`init`(*target*, *args*, *kwargs*)

Receive an instance when its constructor is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'init')
def receive_init(target, args, kwargs):
    "listen for the 'init' event"

    # ... (event handling logic) ...
```

This method is only called during a userland construction of an object, in conjunction with the object's constructor, e.g. its `__init__` method. It is not called when an object is loaded from the database; see the `InstanceEvents.load()` event in order to intercept a database load.

The event is called before the actual `__init__` constructor of the object is called. The `kwargs` dictionary may be modified in-place in order to affect what is passed to `__init__`.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.
- **args** – positional arguments passed to the `__init__` method. This is passed as a tuple and is currently immutable.
- **kwargs** – keyword arguments passed to the `__init__` method. This structure *can* be altered in place.

See also:

`InstanceEvents.init_failure()`

`InstanceEvents.load()`

`init_failure`(*target*, *args*, *kwargs*)

Receive an instance when its constructor has been called, and raised an exception.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'init_failure')
def receive_init_failure(target, args, kwargs):
    "listen for the 'init_failure' event"

    # ... (event handling logic) ...
```

This method is only called during a userland construction of an object, in conjunction with the object's constructor, e.g. its `__init__` method. It is not called when an object is loaded from the database.

The event is invoked after an exception raised by the `__init__` method is caught. After the event is invoked, the original exception is re-raised outwards, so that the construction of the object still raises an exception. The actual exception and stack trace raised should be present in `sys.exc_info()`.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.
- **args** – positional arguments that were passed to the `__init__` method.
- **kwargs** – keyword arguments that were passed to the `__init__` method.

See also:

`InstanceEvents.init()`

`InstanceEvents.load()`

`load`(*target*, *context*)

Receive an object instance after it has been created via `__new__`, and after initial attribute population has occurred.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'load')
def receive_load(target, context):
    "listen for the 'load' event"

    # ... (event handling logic) ...
```

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 `InstanceEvents.load()` event is also available in a class-method decorator format called `orm.reconstructor()`.

Parameters

- **`target`** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.
- **`context`** – the `QueryContext` corresponding to the current `Query` in progress. This argument may be `None` if the load does not correspond to a `Query`, such as during `Session.merge()`.

See also:

`InstanceEvents.init()`

`InstanceEvents.refresh()`

`SessionEvents.loaded_as_persistent()`

mapping_constructors

`pickle`(*target*, *state_dict*)

Receive an object instance when its associated state is being pickled.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'pickle')
def receive_pickle(target, state_dict):
    "listen for the 'pickle' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.
- **state_dict** – the dictionary returned by `InstanceState.__getstate__`, containing the state to be pickled.

refresh(*target*, *context*, *attrs*)

Receive an object instance after one or more attributes have been refreshed from a query.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'refresh')
def receive_refresh(target, context, attrs):
    "listen for the 'refresh' event"

    # ... (event handling logic) ...
```

Contrast this to the `InstanceEvents.load()` method, which is invoked when the object is first loaded from a query.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.
- **context** – the `QueryContext` corresponding to the current `Query` in progress.
- **attrs** – sequence of attribute names which were populated, or None if all column-mapped, non-deferred attributes were populated.

See also:

`InstanceEvents.load()`

refresh_flush(*target*, *flush_context*, *attrs*)

Receive an object instance after one or more attributes have been refreshed within the persistence of the object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'refresh_flush')
def receive_refresh_flush(target, flush_context, attrs):
    "listen for the 'refresh_flush' event"

    # ... (event handling logic) ...
```

This event is the same as `InstanceEvents.refresh()` except it is invoked within the unit of work flush process, and the values here typically come from the process of handling an INSERT or UPDATE, such as via the RETURNING clause or from Python-side default values.

New in version 1.0.5.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

- **flush_context** – Internal UOWTransaction object which handles the details of the flush.
- **attrs** – sequence of attribute names which were populated.

unpickle(*target*, *state_dict*)

Receive an object instance after its associated state has been unpickled.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'unpickle')
def receive_unpickle(target, state_dict):
    "listen for the 'unpickle' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the mapped instance. If the event is configured with raw=True, this will instead be the InstanceState state-management object associated with the instance.
- **state_dict** – the dictionary sent to InstanceState.__setstate__, containing the state dictionary which was pickled.

Session Events

class sqlalchemy.orm.events.**SessionEvents**

Define events specific to Session lifecycle.

e.g.:

```
from sqlalchemy import event
from sqlalchemy.orm import sessionmaker

def my_before_commit(session):
    print "before commit!"

Session = sessionmaker()

event.listen(Session, "before_commit", my_before_commit)
```

The listen() function will accept Session objects as well as the return result of sessionmaker() and scoped_session().

Additionally, it accepts the Session class which will apply listeners to all Session instances globally.

after_attach(*session*, *instance*)

Execute after an instance is attached to a session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_attach')
def receive_after_attach(session, instance):
    "listen for the 'after_attach' event"

    # ... (event handling logic) ...
```

This is called after an add, delete or merge.

Note: As of 0.8, this event fires off *after* the item has been fully associated with the session, which is different than previous releases. For event handlers that require the object not yet be part of session state (such as handlers which may autoflush while the target object is not yet complete) consider the new `before_attach()` event.

See also:

`before_attach()`

session_lifecycle_events

after_begin(*session*, *transaction*, *connection*)
Execute after a transaction is begun on a connection

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_begin')
def receive_after_begin(session, transaction, connection):
    "listen for the 'after_begin' event"

    # ... (event handling logic) ...
```

Parameters

- **session** – The target `Session`.
- **transaction** – The `SessionTransaction`.
- **connection** – The `Connection` object which will be used for SQL statements.

See also:

`before_commit()`

`after_commit()`

`after_transaction_create()`

`after_transaction_end()`

after_bulk_delete(*delete_context*)
Execute after a bulk delete operation to the session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style (arguments as of 0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_delete')
def receive_after_bulk_delete(delete_context):
    "listen for the 'after_bulk_delete' event"

    # ... (event handling logic) ...

# legacy calling style (pre-0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_delete')
def receive_after_bulk_delete(session, query, query_context, result):
    "listen for the 'after_bulk_delete' event"

    # ... (event handling logic) ...
```

Changed in version 0.9: The `after_bulk_delete` event now accepts the arguments `delete_context`. Listener functions which accept the previous argument signature(s) listed above will be automatically adapted to the new signature.

This is called as a result of the `Query.delete()` method.

Parameters `delete_context` – a "delete context" object which contains details about the update, including these attributes:

- `session` - the `Session` involved
- `query` -the `Query` object that this update operation was called upon.
- `context` The `QueryContext` object, corresponding to the invocation of an ORM query.
- `result` the `ResultProxy` returned as a result of the bulk DELETE operation.

`after_bulk_update`(*update_context*)

Execute after a bulk update operation to the session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style (arguments as of 0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_update')
def receive_after_bulk_update(update_context):
    "listen for the 'after_bulk_update' event"

    # ... (event handling logic) ...

# legacy calling style (pre-0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_update')
def receive_after_bulk_update(session, query, query_context, result):
    "listen for the 'after_bulk_update' event"

    # ... (event handling logic) ...
```

Changed in version 0.9: The `after_bulk_update` event now accepts the arguments `update_context`. Listener functions which accept the previous argument signature(s) listed above will be automatically adapted to the new signature.

This is called as a result of the `Query.update()` method.

Parameters `update_context` – an "update context" object which contains details about the update, including these attributes:

- `session` - the `Session` involved
- `query` -the `Query` object that this update operation was called upon.
- `context` The `QueryContext` object, corresponding to the invocation of an ORM query.
- `result` the `ResultProxy` returned as a result of the bulk UPDATE operation.

`after_commit`(*session*)

Execute after a commit has occurred.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_commit')
def receive_after_commit(session):
    "listen for the 'after_commit' event"
```

```
    # ... (event handling logic) ...
```

Note: The `after_commit()` hook is *not* per-flush, that is, the `Session` can emit SQL to the database many times within the scope of a transaction. For interception of these events, use the `before_flush()`, `after_flush()`, or `after_flush_postexec()` events.

Note: The `Session` is not in an active transaction when the `after_commit()` event is invoked, and therefore can not emit SQL. To emit SQL corresponding to every transaction, use the `before_commit()` event.

Parameters **session** – The target `Session`.

See also:

`before_commit()`

`after_begin()`

`after_transaction_create()`

`after_transaction_end()`

after_flush(*session*, *flush_context*)
Execute after flush has completed, but before commit has been called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_flush')
def receive_after_flush(session, flush_context):
    "listen for the 'after_flush' event"

    # ... (event handling logic) ...
```

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.

Parameters

- **session** – The target `Session`.
- **flush_context** – Internal `UOWTransaction` object which handles the details of the flush.

See also:

`before_flush()`

`after_flush_postexec()`

session_persistence_events

after_flush_postexec(*session*, *flush_context*)
Execute after flush has completed, and after the post-exec state occurs.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
```

```
@event.listens_for(SomeSessionOrFactory, 'after_flush_postexec')
def receive_after_flush_postexec(session, flush_context):
    "listen for the 'after_flush_postexec' event"

    # ... (event handling logic) ...
```

This will be when the 'new', 'dirty', and 'deleted' lists are in their final state. An actual commit() may or may not have occurred, depending on whether or not the flush started its own transaction or participated in a larger transaction.

Parameters

- **session** – The target `Session`.
- **flush_context** – Internal `UOWTransaction` object which handles the details of the flush.

See also:

`before_flush()`

`after_flush()`

session_persistence_events

after_rollback(*session*)

Execute after a real DBAPI rollback has occurred.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_rollback')
def receive_after_rollback(session):
    "listen for the 'after_rollback' event"

    # ... (event handling logic) ...
```

Note that this event only fires when the *actual* rollback against the database occurs - it does *not* fire each time the `Session.rollback()` method is called, if the underlying DBAPI transaction has already been rolled back. In many cases, the `Session` will not be in an "active" state during this event, as the current transaction is not valid. To acquire a `Session` which is active after the outermost rollback has proceeded, use the `SessionEvents.after_soft_rollback()` event, checking the `Session.is_active` flag.

Parameters **session** – The target `Session`.

after_soft_rollback(*session*, *previous_transaction*)

Execute after any rollback has occurred, including "soft" rollbacks that don't actually emit at the DBAPI level.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_soft_rollback')
def receive_after_soft_rollback(session, previous_transaction):
    "listen for the 'after_soft_rollback' event"

    # ... (event handling logic) ...
```

This corresponds to both nested and outer rollbacks, i.e. the innermost rollback that calls the DBAPI's rollback() method, as well as the enclosing rollback calls that only pop themselves from the transaction stack.

The given Session can be used to invoke SQL and Session.query() operations after an outermost rollback by first checking the Session.is_active flag:

```
@event.listens_for(Session, "after_soft_rollback")
def do_something(session, previous_transaction):
    if session.is_active:
        session.execute("select * from some_table")
```

Parameters

- **session** – The target Session.
- **previous_transaction** – The SessionTransaction transactional marker object which was just closed. The current SessionTransaction for the given Session is available via the Session.transaction attribute.

New in version 0.7.3.

after_transaction_create(*session*, *transaction*)

Execute when a new SessionTransaction is created.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_transaction_create')
def receive_after_transaction_create(session, transaction):
    "listen for the 'after_transaction_create' event"

    # ... (event handling logic) ...
```

This event differs from after_begin() in that it occurs for each SessionTransaction overall, as opposed to when transactions are begun on individual database connections. It is also invoked for nested transactions and subtransactions, and is always matched by a corresponding after_transaction_end() event (assuming normal operation of the Session).

Parameters

- **session** – the target Session.
- **transaction** – the target SessionTransaction.

  To detect if this is the outermost SessionTransaction, as opposed to a "subtransaction" or a SAVEPOINT, test that the SessionTransaction.parent attribute is None:

  ```
  @event.listens_for(session, "after_transaction_create")
  def after_transaction_create(session, transaction):
      if transaction.parent is None:
          # work with top-level transaction
  ```

  To detect if the SessionTransaction is a SAVEPOINT, use the SessionTransaction.nested attribute:

  ```
  @event.listens_for(session, "after_transaction_create")
  def after_transaction_create(session, transaction):
      if transaction.nested:
          # work with SAVEPOINT transaction
  ```

See also:

SessionTransaction

after_transaction_end()

after_transaction_end(*session*, *transaction*)

Execute when the span of a `SessionTransaction` ends.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_transaction_end')
def receive_after_transaction_end(session, transaction):
    "listen for the 'after_transaction_end' event"

    # ... (event handling logic) ...
```

This event differs from `after_commit()` in that it corresponds to all `SessionTransaction` objects in use, including those for nested transactions and subtransactions, and is always matched by a corresponding `after_transaction_create()` event.

Parameters

- **session** – the target `Session`.
- **transaction** – the target `SessionTransaction`.

  To detect if this is the outermost `SessionTransaction`, as opposed to a "sub-transaction" or a SAVEPOINT, test that the `SessionTransaction.parent` attribute is `None`:

  ```
  @event.listens_for(session, "after_transaction_create")
  def after_transaction_end(session, transaction):
      if transaction.parent is None:
          # work with top-level transaction
  ```

  To detect if the `SessionTransaction` is a SAVEPOINT, use the `SessionTransaction.nested` attribute:

  ```
  @event.listens_for(session, "after_transaction_create")
  def after_transaction_end(session, transaction):
      if transaction.nested:
          # work with SAVEPOINT transaction
  ```

See also:

`SessionTransaction`

`after_transaction_create()`

before_attach(*session*, *instance*)

Execute before an instance is attached to a session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'before_attach')
def receive_before_attach(session, instance):
    "listen for the 'before_attach' event"

    # ... (event handling logic) ...
```

This is called before an add, delete or merge causes the object to be part of the session.

New in version 0.8.: Note that `after_attach()` now fires off after the item is part of the session. `before_attach()` is provided for those cases where the item should not yet be part of the session state.

See also:

`after_attach()`

session_lifecycle_events

`before_commit`(*session*)
Execute before commit is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'before_commit')
def receive_before_commit(session):
    "listen for the 'before_commit' event"

    # ... (event handling logic) ...
```

Note: The `before_commit()` hook is *not* per-flush, that is, the `Session` can emit SQL to the database many times within the scope of a transaction. For interception of these events, use the `before_flush()`, `after_flush()`, or `after_flush_postexec()` events.

Parameters `session` – The target `Session`.

See also:

`after_commit()`

`after_begin()`

`after_transaction_create()`

`after_transaction_end()`

`before_flush`(*session*, *flush_context*, *instances*)
Execute before flush process has started.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'before_flush')
def receive_before_flush(session, flush_context, instances):
    "listen for the 'before_flush' event"

    # ... (event handling logic) ...
```

Parameters

- `session` – The target `Session`.
- `flush_context` – Internal `UOWTransaction` object which handles the details of the flush.
- `instances` – Usually `None`, this is the collection of objects which can be passed to the `Session.flush()` method (note this usage is deprecated).

See also:

`after_flush()`

`after_flush_postexec()`

session_persistence_events

deleted_to_detached(*session*, *instance*)
Intercept the "deleted to detached" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'deleted_to_detached')
def receive_deleted_to_detached(session, instance):
    "listen for the 'deleted_to_detached' event"

    # ... (event handling logic) ...
```

This event is invoked when a deleted object is evicted from the session. The typical case when this occurs is when the transaction for a `Session` in which the object was deleted is committed; the object moves from the deleted state to the detached state.

It is also invoked for objects that were deleted in a flush when the `Session.expunge_all()` or `Session.close()` events are called, as well as if the object is individually expunged from its deleted state via `Session.expunge()`.

New in version 1.1.

See also:

session_lifecycle_events

deleted_to_persistent(*session*, *instance*)
Intercept the "deleted to persistent" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'deleted_to_persistent')
def receive_deleted_to_persistent(session, instance):
    "listen for the 'deleted_to_persistent' event"

    # ... (event handling logic) ...
```

This transition occurs only when an object that's been deleted successfully in a flush is restored due to a call to `Session.rollback()`. The event is not called under any other circumstances.

New in version 1.1.

See also:

session_lifecycle_events

detached_to_persistent(*session*, *instance*)
Intercept the "detached to persistent" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'detached_to_persistent')
def receive_detached_to_persistent(session, instance):
    "listen for the 'detached_to_persistent' event"

    # ... (event handling logic) ...
```

This event is a specialization of the `SessionEvents.after_attach()` event which is only invoked for this specific transition. It is invoked typically during the `Session.add()` call, as well as during the `Session.delete()` call if the object was not previously associated with the `Session` (note that an object marked as "deleted" remains in the "persistent" state until the flush proceeds).

Note: If the object becomes persistent as part of a call to `Session.delete()`, the object is **not** yet marked as deleted when this event is called. To detect deleted objects, check the `deleted` flag sent to the `SessionEvents.persistent_to_detached()` to event after the flush proceeds, or check the `Session.deleted` collection within the `SessionEvents.before_flush()` event if deleted objects need to be intercepted before the flush.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

session_lifecycle_events

`loaded_as_persistent`(*session*, *instance*)

Intercept the "loaded as persistent" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'loaded_as_persistent')
def receive_loaded_as_persistent(session, instance):
    "listen for the 'loaded_as_persistent' event"

    # ... (event handling logic) ...
```

This event is invoked within the ORM loading process, and is invoked very similarly to the `InstanceEvents.load()` event. However, the event here is linkable to a `Session` class or instance, rather than to a mapper or class hierarchy, and integrates with the other session lifecycle events smoothly. The object is guaranteed to be present in the session's identity map when this event is called.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

session_lifecycle_events

`pending_to_persistent`(*session*, *instance*)

Intercept the "pending to persistent"" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'pending_to_persistent')
```

```
def receive_pending_to_persistent(session, instance):
    "listen for the 'pending_to_persistent' event"

    # ... (event handling logic) ...
```

This event is invoked within the flush process, and is similar to scanning the `Session.new` collection within the `SessionEvents.after_flush()` event. However, in this case the object has already been moved to the persistent state when the event is called.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

session_lifecycle_events

pending_to_transient(*session*, *instance*)
Intercept the "pending to transient" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'pending_to_transient')
def receive_pending_to_transient(session, instance):
    "listen for the 'pending_to_transient' event"

    # ... (event handling logic) ...
```

This less common transition occurs when an pending object that has not been flushed is evicted from the session; this can occur when the `Session.rollback()` method rolls back the transaction, or when the `Session.expunge()` method is used.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

session_lifecycle_events

persistent_to_deleted(*session*, *instance*)
Intercept the "persistent to deleted" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'persistent_to_deleted')
def receive_persistent_to_deleted(session, instance):
    "listen for the 'persistent_to_deleted' event"

    # ... (event handling logic) ...
```

This event is invoked when a persistent object's identity is deleted from the database within a flush, however the object still remains associated with the `Session` until the transaction completes.

If the transaction is rolled back, the object moves again to the persistent state, and the `SessionEvents.deleted_to_persistent()` event is called. If the transaction is committed, the object becomes detached, which will emit the `SessionEvents.deleted_to_detached()` event.

Note that while the `Session.delete()` method is the primary public interface to mark an object as deleted, many objects get deleted due to cascade rules, which are not always determined until flush time. Therefore, there's no way to catch every object that will be deleted until the flush has proceeded. the `SessionEvents.persistent_to_deleted()` event is therefore invoked at the end of a flush.

New in version 1.1.

See also:

session_lifecycle_events

`persistent_to_detached`(*session*, *instance*)
Intercept the "persistent to detached" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'persistent_to_detached')
def receive_persistent_to_detached(session, instance):
    "listen for the 'persistent_to_detached' event"

    # ... (event handling logic) ...
```

This event is invoked when a persistent object is evicted from the session. There are many conditions that cause this to happen, including:

- using a method such as `Session.expunge()` or `Session.close()`
- Calling the `Session.rollback()` method, when the object was part of an INSERT statement for that session's transaction

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.
- **deleted** – boolean. If True, indicates this object moved to the detached state because it was marked as deleted and flushed.

New in version 1.1.

See also:

session_lifecycle_events

`persistent_to_transient`(*session*, *instance*)
Intercept the "persistent to transient" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'persistent_to_transient')
def receive_persistent_to_transient(session, instance):
    "listen for the 'persistent_to_transient' event"

    # ... (event handling logic) ...
```

This less common transition occurs when an pending object that has has been flushed is evicted from the session; this can occur when the `Session.rollback()` method rolls back the transaction.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

session_lifecycle_events

`transient_to_pending`(*session*, *instance*)
Intercept the "transient to pending" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'transient_to_pending')
def receive_transient_to_pending(session, instance):
    "listen for the 'transient_to_pending' event"

    # ... (event handling logic) ...
```

This event is a specialization of the `SessionEvents.after_attach()` event which is only invoked for this specific transition. It is invoked typically during the `Session.add()` call.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

session_lifecycle_events

Query Events

`class sqlalchemy.orm.events.QueryEvents`
Represent events within the construction of a `Query` object.

The events here are intended to be used with an as-yet-unreleased inspection system for `Query`. Some very basic operations are possible now, however the inspection system is intended to allow complex query manipulations to be automated.

New in version 1.0.0.

`before_compile`(*query*)
Receive the `Query` object before it is composed into a core `Select` object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeQuery, 'before_compile')
def receive_before_compile(query):
    "listen for the 'before_compile' event"
```

```
    # ... (event handling logic) ...
```

This event is intended to allow changes to the query given:

```
@event.listens_for(Query, "before_compile", retval=True)
def no_deleted(query):
    for desc in query.column_descriptions:
        if desc['type'] is User:
            entity = desc['entity']
            query = query.filter(entity.deleted == False)
    return query
```

The event should normally be listened with the `retval=True` parameter set, so that the modified query may be returned.

Instrumentation Events

Defines SQLAlchemy's system of class instrumentation.

This module is usually not directly visible to user applications, but defines a large part of the ORM's interactivity.

instrumentation.py deals with registration of end-user classes for state tracking. It interacts closely with state.py and attributes.py which establish per-instance and per-class-attribute instrumentation, respectively.

The class instrumentation system can be customized on a per-class or global basis using the `sqlalchemy.ext.instrumentation` module, which provides the means to build and specify alternate instrumentation forms.

class sqlalchemy.orm.events.**InstrumentationEvents**

Events related to class instrumentation events.

The listeners here support being established against any new style class, that is any object that is a subclass of 'type'. Events will then be fired off for events against that class. If the "propagate=True" flag is passed to event.listen(), the event will fire off for subclasses of that class as well.

The Python `type` builtin is also accepted as a target, which when used has the effect of events being emitted for all classes.

Note the "propagate" flag here is defaulted to `True`, unlike the other class level events where it defaults to `False`. This means that new subclasses will also be the subject of these events, when a listener is established on a superclass.

Changed in version 0.8: - events here will emit based on comparing the incoming class to the type of class passed to `event.listen()`. Previously, the event would fire for any class unconditionally regardless of what class was sent for listening, despite documentation which stated the contrary.

attribute_instrument(*cls*, *key*, *inst*)

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeBaseClass, 'attribute_instrument')
def receive_attribute_instrument(cls, key, inst):
    "listen for the 'attribute_instrument' event"

    # ... (event handling logic) ...
```

Called when an attribute is instrumented.

`class_instrument`(*cls*)
Called after the given class is instrumented.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeBaseClass, 'class_instrument')
def receive_class_instrument(cls):
    "listen for the 'class_instrument' event"

    # ... (event handling logic) ...
```

To get at the `ClassManager`, use `manager_of_class()`.

`class_uninstrument`(*cls*)
Called before the given class is uninstrumented.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeBaseClass, 'class_uninstrument')
def receive_class_uninstrument(cls):
    "listen for the 'class_uninstrument' event"

    # ... (event handling logic) ...
```

To get at the `ClassManager`, use `manager_of_class()`.

2.6.2 ORM Internals

Key ORM constructs, not otherwise covered in other sections, are listed here.

`class sqlalchemy.orm.state.AttributeState`(*state*, *key*)
Provide an inspection interface corresponding to a particular attribute on a particular mapped object.

The `AttributeState` object is accessed via the `InstanceState.attrs` collection of a particular `InstanceState`:

```
from sqlalchemy import inspect

insp = inspect(some_mapped_object)
attr_state = insp.attrs.some_attribute
```

`history`
Return the current pre-flush change history for this attribute, via the `History` interface.

This method will **not** emit loader callables if the value of the attribute is unloaded.

See also:

`AttributeState.load_history()` - retrieve history using loader callables if the value is not locally present.

`attributes.get_history()` - underlying function

`load_history()`
Return the current pre-flush change history for this attribute, via the `History` interface.

This method **will** emit loader callables if the value of the attribute is unloaded.

See also:

`AttributeState.history`

`attributes.get_history()` - underlying function

New in version 0.9.0.

loaded_value
The current value of this attribute as loaded from the database.

If the value has not been loaded, or is otherwise not present in the object's dictionary, returns NO_VALUE.

value
Return the value of this attribute.

This operation is equivalent to accessing the object's attribute directly or via `getattr()`, and will fire off any pending loader callables if needed.

class sqlalchemy.orm.util.CascadeOptions
Keeps track of the options sent to relationship().cascade

class sqlalchemy.orm.instrumentation.ClassManager(*class_*)
tracks state information at the class level.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

dispose()
Dissasociate this manager from its class.

fromkeys()
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_parent(*state*, *key*, *optimistic=False*)
TODO

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

manage()
Mark this instance as the manager for its class.

original_init
Initialize self. See help(type(self)) for accurate signature.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

state_getter()
Return a (instance) -> InstanceState callable.

"state getter" callables should raise either KeyError or AttributeError if no InstanceState could be found for the instance.

unregister()
remove all instrumentation established by this ClassManager.

update(*E*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`values()` → an object providing a view on D's values

`class sqlalchemy.orm.properties.ColumnProperty(*columns, **kwargs)`
Describes an object attribute that corresponds to a table column.

Public constructor is the `orm.column_property()` function.

`class Comparator(prop, parentmapper, adapt_to_entity=None)`
Produce boolean, comparison, and other operators for `ColumnProperty` attributes.

See the documentation for `PropComparator` for a brief overview.

See also:

`PropComparator`

`ColumnOperators`

types_operators

`TypeEngine.comparator_factory`

`adapt_to_entity(adapt_to_entity)`
Return a copy of this PropComparator which will use the given `AliasedInsp` to produce corresponding expressions.

`adapter`
Produce a callable that adapts column expressions to suit an aliased version of this comparator.

`all_()`
Produce a `all_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == mytable.c.somearray.all_()

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().all_()
```

See also:

`all_()` - standalone version

`any_()` - ANY operator

New in version 1.1.

`any(criterion=None, **kwargs)`
Return true if this collection contains any member that meets the given criterion.

The usual implementation of `any()` is `RelationshipProperty.Comparator.any()`.

Parameters

- **`criterion`** – an optional ClauseElement formulated against the member class' table or attributes.
- **`**kwargs`** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

`any_()`
Produce a `any_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == mytable.c.somearray.any_()
```

```
# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().any_()
```

See also:

`any_()` - standalone version

`all_()` - ALL operator

New in version 1.1.

`asc()`
Produce a `asc()` clause against the parent object.

`between(`*cleft*, *cright*, *symmetric=False*`)`
Produce a `between()` clause against the parent object, given the lower and upper range.

`bool_op(`*opstring*, *precedence=0*`)`
Return a custom boolean operator.

This method is shorthand for calling `Operators.op()` and passing the `Operators.op.is_comparison` flag with True.

New in version 1.2.0b3.

See also:

`Operators.op()`

`collate(`*collation*`)`
Produce a `collate()` clause against the parent object, given the collation string.

See also:

`collate()`

`concat(`*other*`)`
Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

`contains(`*other*, ***kwargs*`)`
Implement the 'contains' operator.

Produces a LIKE expression that tests against a match for the middle of a string value:

```
column LIKE '%' || <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.contains("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.contains.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.contains.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.contains.autoescape` flag is set to True.

- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.contains("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.contains.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.contains.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.contains("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.contains.autoescape:

  ```
  somecolumn.contains("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.endswith()

ColumnOperators.like()

desc()
Produce a desc() clause against the parent object.

distinct()
Produce a distinct() clause against the parent object.

endswith(*other*, ***kwargs*)
Implement the 'endswith' operator.

Produces a LIKE expression that tests against a match for the end of a string value:

```
column LIKE '%' || <other>
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.endswith("foobar"))
```

Since the operator uses LIKE, wildcard characters "%" and "_" that are present inside the <other> expression will behave like wildcards as well. For literal string values, the ColumnOperators.endswith.autoescape flag may be set to True to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the ColumnOperators.endswith.escape parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.endswith.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.endswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.endswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.endswith.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.endswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.endswith.autoescape:

  ```
  somecolumn.endswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.contains()

ColumnOperators.like()

has(*criterion=None*, ***kwargs*)
Return true if this element references a member which meets the given criterion.

The usual implementation of `has()` is `RelationshipProperty.Comparator.has()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

ilike(*other*, *escape=None*)
Implement the `ilike` operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

```
lower(a) LIKE lower(other)
```

Or on backends that support the ILIKE operator:

```
a ILIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

`ColumnOperators.like()`

in_(*other*)
Implement the `in` operator.

In a column context, produces the clause `a IN other`. "other" may be a tuple/list of column expressions, or a `select()` construct.

In the case that `other` is an empty sequence, the compiler produces an "empty in" expression. This defaults to the expression "1 != 1" to produce false in all cases. The `create_engine.empty_in_strategy` may be used to alter this behavior.

Changed in version 1.2: The `ColumnOperators.in_()` and `ColumnOperators.notin_()` operators now produce a "static" expression for an empty IN sequence by default.

is_(*other*)
Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

`ColumnOperators.isnot()`

is_distinct_from(*other*)
Implement the `IS DISTINCT FROM` operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

New in version 1.1.

isnot(*other*)
Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

isnot_distinct_from(*other*)
Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

New in version 1.1.

like(*other*, *escape=None*)
Implement the like operator.

In a column context, produces the expression:

```
a LIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.like("%foobar%"))
```

Parameters
- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)
Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:
- PostgreSQL - renders x @@ to_tsquery(y)
- MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
- Oracle - renders CONTAINS(x, y)
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
implement the NOT ILIKE operator.

This is equivalent to using negation with ColumnOperators.ilike(), i.e. ~x.ilike(y).

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)
implement the NOT IN operator.

This is equivalent to using negation with ColumnOperators.in_(), i.e. ~x.in_(y).

In the case that other is an empty sequence, the compiler produces an "empty not in" expression. This defaults to the expression "1 = 1" to produce true in all cases. The create_engine.empty_in_strategy may be used to alter this behavior.

Changed in version 1.2: The ColumnOperators.in_() and ColumnOperators.notin_() operators now produce a "static" expression for an empty IN sequence by default.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)
implement the NOT LIKE operator.

This is equivalent to using negation with ColumnOperators.like(), i.e. ~x.like(y).

New in version 0.8.

See also:

ColumnOperators.like()

nullsfirst()
Produce a nullsfirst() clause against the parent object.

nullslast()
Produce a nullslast() clause against the parent object.

of_type(*class_*)
Redefine this object in terms of a polymorphic subclass.

Returns a new PropComparator from which further criterion can be evaluated.

e.g.:

```
query.join(Company.employees.of_type(Engineer)).\
   filter(Engineer.name=='foo')
```

Parameters **class_** – a class or mapper indicating that criterion will be against this specific subclass.

See also:

inheritance_of_type

op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.

- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like `==`, `>`, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.
- **return_type** – a `TypeEngine` class or object that will force the return type of an expression produced by this operator to be of that type. By default, operators that specify `Operators.op.is_comparison` will resolve to `Boolean`, and those that do not will be of the same type as the left-hand operand.

  New in version 1.2.0b3: - added the `Operators.op.return_type` argument.

See also:

types_operators

relationship_custom_operator

startswith(*other*, ***kwargs*)

Implement the `startswith` operator.

Produces a LIKE expression that tests against a match for the start of a string value:

```
column LIKE <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.startswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.startswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.startswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.startswith.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.startswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '/'
  ```

With the value of :param as "foo/%bar".

New in version 1.2.

Changed in version 1.2.0: The ColumnOperators.startswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.startswith.escape parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

An expression such as:

```
somecolumn.startswith("foo/%bar", escape="^")
```

Will render as:

```
somecolumn LIKE :param || '%' ESCAPE '^'
```

The parameter may also be combined with ColumnOperators.startswith.autoescape:

```
somecolumn.startswith("foo%bar^bat", escape="^", autoescape=True)
```

Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.endswith()

ColumnOperators.contains()

ColumnOperators.like()

cascade_iterator(*type_*, *state*, *visited_instances=None*, *halt_on=None*)

Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade_iterator is called.

This method typically only applies to RelationshipProperty.

class_attribute

Return the class-bound descriptor corresponding to this MapperProperty.

This is basically a getattr() call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this MapperProperty were named addresses, and the class to which it is mapped is User, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
>>> User.addresses.property is addresses_property
True
```

expression
Return the primary column or expression for this ColumnProperty.

extension_type = symbol('NOT_EXTENSION')

init()
Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

set_parent(*parent*, *init*)
Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

class sqlalchemy.orm.properties.ComparableProperty(*comparator_factory*, *descriptor=None*, *doc=None*, *info=None*)
Instruments a Python property for use in query expressions.

class sqlalchemy.orm.descriptor_props.CompositeProperty(*class_*, **attrs*, ***kwargs*)
Defines a "composite" mapped attribute, representing a collection of columns as one attribute.

CompositeProperty is constructed using the composite() function.

See also:

mapper_composite

class Comparator(*prop*, *parentmapper*, *adapt_to_entity=None*)
Produce boolean, comparison, and other operators for CompositeProperty attributes.

See the example in composite_operations for an overview of usage , as well as the documentation for PropComparator.

See also:

PropComparator

ColumnOperators

types_operators

TypeEngine.comparator_factory

do_init()
Initialization which occurs after the CompositeProperty has been associated with its parent mapper.

get_history(*state*, *dict_*, *passive=symbol('PASSIVE_OFF')*)
Provided for userland code that uses attributes.get_history().

class sqlalchemy.orm.attributes.Event(*attribute_impl*, *op*)
A token propagated throughout the course of a chain of attribute events.

Serves as an indicator of the source of the event and also provides a means of controlling propagation across a chain of attribute operations.

The Event object is sent as the initiator argument when dealing with events such as AttributeEvents.append(), AttributeEvents.set(), and AttributeEvents.remove().

The Event object is currently interpreted by the backref event handlers, and is used to control the propagation of operations across two mutually-dependent attributes.

New in version 0.9.0.

Variables

- **impl** – The AttributeImpl which is the current event initiator.
- **op** – The symbol OP_APPEND, OP_REMOVE, OP_REPLACE, or OP_BULK_REPLACE, indicating the source operation.

class sqlalchemy.orm.identity.IdentityMap

check_modified()
return True if any InstanceStates present have been marked as 'modified'.

class sqlalchemy.orm.base.InspectionAttr
A base class applied to all ORM objects that can be returned by the inspect() function.

The attributes defined here allow the usage of simple boolean checks to test basic facts about the object returned.

While the boolean checks here are basically the same as using the Python isinstance() function, the flags here can be used without the need to import all of these classes, and also such that the SQLAlchemy class system can change while leaving the flags here intact for forwards-compatibility.

extension_type = symbol('NOT_EXTENSION')
The extension type, if any. Defaults to interfaces.NOT_EXTENSION

New in version 0.8.0.

See also:

HYBRID_METHOD

HYBRID_PROPERTY

ASSOCIATION_PROXY

is_aliased_class = False
True if this object is an instance of AliasedClass.

is_attribute = False
True if this object is a Python descriptor.

This can refer to one of many types. Usually a QueryableAttribute which handles attributes events on behalf of a MapperProperty. But can also be an extension type such as AssociationProxy or hybrid_property. The InspectionAttr.extension_type will refer to a constant identifying the specific subtype.

See also:

Mapper.all_orm_descriptors

is_clause_element = False
True if this object is an instance of ClauseElement.

is_instance = False
True if this object is an instance of InstanceState.

is_mapper = False
True if this object is an instance of Mapper.

is_property = False
True if this object is an instance of MapperProperty.

is_selectable = False
Return True if this object is an instance of Selectable.

class sqlalchemy.orm.base.InspectionAttrInfo
Adds the .info attribute to InspectionAttr.

The rationale for InspectionAttr vs. InspectionAttrInfo is that the former is compatible as a mixin for classes that specify __slots__; this is essentially an implementation artifact.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this InspectionAttr.

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the `column_property()`, `relationship()`, or `composite()` functions.

New in version 0.8: Added support for .info to all `MapperProperty` subclasses.

Changed in version 1.0.0: `MapperProperty.info` is also available on extension types via the `InspectionAttrInfo.info` attribute, so that it can apply to a wider variety of ORM and extension constructs.

See also:

`QueryableAttribute.info`

`SchemaItem.info`

`class sqlalchemy.orm.state.InstanceState(obj, manager)`

tracks state information at the instance level.

The `InstanceState` is a key object used by the SQLAlchemy ORM in order to track the state of an object; it is created the moment an object is instantiated, typically as a result of instrumentation which SQLAlchemy applies to the `__init__()` method of the class.

`InstanceState` is also a semi-public object, available for runtime inspection as to the state of a mapped instance, including information such as its current status within a particular `Session` and details about data on individual attributes. The public API in order to acquire a `InstanceState` object is to use the `inspect()` system:

```
>>> from sqlalchemy import inspect
>>> insp = inspect(some_mapped_object)
```

See also:

core_inspection_toplevel

attrs

Return a namespace representing each attribute on the mapped object, including its current value and history.

The returned object is an instance of `AttributeState`. This object allows inspection of the current data within an attribute as well as attribute history since the last flush.

callables = ()

A namespace where a per-state loader callable can be associated.

In SQLAlchemy 1.0, this is only used for lazy loaders / deferred loaders that were set up via query option.

Previously, callables was used also to indicate expired attributes by storing a link to the InstanceState itself in this dictionary. This role is now handled by the expired_attributes set.

deleted

Return true if the object is deleted.

An object that is in the deleted state is guaranteed to not be within the `Session.identity_map` of its parent `Session`; however if the session's transaction is rolled back, the object will be restored to the persistent state and the identity map.

Note: The `InstanceState.deleted` attribute refers to a specific state of the object that occurs between the "persistent" and "detached" states; once the object is detached, the `InstanceState.deleted` attribute **no longer returns True**; in order to detect that a state was deleted, regardless of whether or not the object is associated with a `Session`, use the `InstanceState.was_deleted` accessor.

See also:

session_object_states

`detached`
Return true if the object is detached.

See also:

session_object_states

`dict`
Return the instance dict used by the object.

Under normal circumstances, this is always synonymous with the `__dict__` attribute of the mapped object, unless an alternative instrumentation system has been configured.

In the case that the actual object has been garbage collected, this accessor returns a blank dictionary.

`expired_attributes = None`
The set of keys which are 'expired' to be loaded by the manager's deferred scalar loader, assuming no pending changes.

see also the `unmodified` collection which is intersected against this set when a refresh operation occurs.

`has_identity`
Return `True` if this object has an identity key.

This should always have the same value as the expression `state.persistent or state.detached`.

`identity`
Return the mapped identity of the mapped object. This is the primary key identity as persisted by the ORM which can always be passed directly to `Query.get()`.

Returns `None` if the object has no primary key identity.

Note: An object which is transient or pending does **not** have a mapped identity until it is flushed, even if its attributes include primary key values.

`identity_key`
Return the identity key for the mapped object.

This is the key used to locate the object within the `Session.identity_map` mapping. It contains the identity as returned by `identity` within it.

`mapper`
Return the `Mapper` used for this mapepd object.

`object`
Return the mapped object represented by this `InstanceState`.

`pending`
Return true if the object is pending.

See also:

session_object_states

`persistent`
Return true if the object is persistent.

An object that is in the persistent state is guaranteed to be within the `Session.identity_map` of its parent `Session`.

Changed in version 1.1: The `InstanceState.persistent` accessor no longer returns True for an object that was "deleted" within a flush; use the `InstanceState.deleted` accessor to detect this state. This allows the "persistent" state to guarantee membership in the identity map.

See also:

session_object_states

`session`
Return the owning `Session` for this instance, or `None` if none available.

Note that the result here can in some cases be *different* from that of `obj in session`; an object that's been deleted will report as not `in session`, however if the transaction is still in progress, this attribute will still refer to that session. Only when the transaction is completed does the object become fully detached under normal circumstances.

`transient`
Return true if the object is transient.

See also:

session_object_states

`unloaded`
Return the set of keys which do not have a loaded value.

This includes expired attributes and any other attribute that was never populated or modified.

`unloaded_expirable`
Return the set of keys which do not have a loaded value.

This includes expired attributes and any other attribute that was never populated or modified.

`unmodified`
Return the set of keys which have no uncommitted changes

`unmodified_intersection`(*keys*)
Return self.unmodified.intersection(keys).

`was_deleted`
Return True if this object is or was previously in the "deleted" state and has not been reverted to persistent.

This flag returns True once the object was deleted in flush. When the object is expunged from the session either explicitly or via transaction commit and enters the "detached" state, this flag will continue to report True.

New in version 1.1: - added a local method form of `orm.util.was_deleted()`.

See also:

`InstanceState.deleted` - refers to the "deleted" state

`orm.util.was_deleted()` - standalone function

session_object_states

`class sqlalchemy.orm.attributes.InstrumentedAttribute`(*class_*, *key*, *impl=None*, *comparator=None*, *parententity=None*, *of_type=None*)
Class bound instrumented attribute which adds basic descriptor methods.

See `QueryableAttribute` for a description of most features.

`__delete__`(*instance*)

`__get__`(*instance*, *owner*)

`__set__`(*instance*, *value*)

`sqlalchemy.orm.interfaces.MANYTOONE = symbol('MANYTOONE')`
Indicates the many-to-one direction for a `relationship()`.

This symbol is typically used by the internals but may be exposed within certain API features.

sqlalchemy.orm.interfaces.MANYTOMANY = symbol('MANYTOMANY')
Indicates the many-to-many direction for a relationship().

This symbol is typically used by the internals but may be exposed within certain API features.

class sqlalchemy.orm.interfaces.MapperProperty
Represent a particular class attribute mapped by Mapper.

The most common occurrences of MapperProperty are the mapped Column, which is represented in a mapping as an instance of ColumnProperty, and a reference to another class produced by relationship(), represented in the mapping as an instance of RelationshipProperty.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this InspectionAttr.

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the column_property(), relationship(), or composite() functions.

New in version 0.8: Added support for .info to all MapperProperty subclasses.

Changed in version 1.0.0: InspectionAttr.info moved from MapperProperty so that it can apply to a wider variety of ORM and extension constructs.

See also:

QueryableAttribute.info

SchemaItem.info

cascade = frozenset()
The set of 'cascade' attribute names.

This collection is checked before the 'cascade_iterator' method is called.

The collection typically only applies to a RelationshipProperty.

cascade_iterator(*type_*, *state*, *visited_instances=None*, *halt_on=None*)
Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade_iterator is called.

This method typically only applies to RelationshipProperty.

class_attribute
Return the class-bound descriptor corresponding to this MapperProperty.

This is basically a getattr() call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this MapperProperty were named addresses, and the class to which it is mapped is User, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
>>> User.addresses.property is addresses_property
True
```

create_row_processor(*context*, *path*, *mapper*, *result*, *adapter*, *populators*)
Produce row processing functions and append to the given set of populators lists.

`do_init()`
Perform subclass-specific initialization post-mapper-creation steps.

This is a template method called by the `MapperProperty` object's init() method.

`init()`
Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

`instrument_class(`*mapper*`)`
Hook called by the Mapper to the property to initiate instrumentation of the class attribute managed by this MapperProperty.

The MapperProperty here will typically call out to the attributes module to set up an InstrumentedAttribute.

This step is the first of two steps to set up an InstrumentedAttribute, and is called early in the mapper setup process.

The second step is typically the init_class_attribute step, called from StrategizedProperty via the post_instrument_class() hook. This step assigns additional state to the InstrumentedAttribute (specifically the "impl") which has been determined after the MapperProperty has determined what kind of persistence management it needs to do (e.g. scalar, object, collection, etc).

`is_property = True`
Part of the InspectionAttr interface; states this object is a mapper property.

`merge(`*session*, *source_state*, *source_dict*, *dest_state*, *dest_dict*, *load*, *_recursive*, *_resolve_conflict_map*`)`
Merge the attribute represented by this `MapperProperty` from source to destination object.

`post_instrument_class(`*mapper*`)`
Perform instrumentation adjustments that need to occur after init() has completed.

The given Mapper is the Mapper invoking the operation, which may not be the same Mapper as self.parent in an inheritance scenario; however, Mapper will always at least be a sub-mapper of self.parent.

This method is typically used by StrategizedProperty, which delegates it to LoaderStrategy.init_class_attribute() to perform final setup on the class-bound InstrumentedAttribute.

`set_parent(`*parent*, *init*`)`
Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

`setup(`*context*, *entity*, *path*, *adapter*, ***kwargs*`)`
Called by Query for the purposes of constructing a SQL statement.

Each MapperProperty associated with the target mapper processes the statement referenced by the query context, adding columns and/or criterion as appropriate.

`sqlalchemy.orm.interfaces.NOT_EXTENSION = symbol('NOT_EXTENSION')`
Symbol indicating an `InspectionAttr` that's not part of sqlalchemy.ext.

Is assigned to the `InspectionAttr.extension_type` attibute.

`sqlalchemy.orm.interfaces.ONETOMANY = symbol('ONETOMANY')`
Indicates the one-to-many direction for a `relationship()`.

This symbol is typically used by the internals but may be exposed within certain API features.

`class sqlalchemy.orm.interfaces.PropComparator(`*prop*, *parentmapper*, *adapt_to_entity=None*`)`
Defines SQL operators for `MapperProperty` objects.

SQLAlchemy allows for operators to be redefined at both the Core and ORM level. `PropComparator` is the base class of operator redefinition for ORM-level operations, including those of `ColumnProperty`, `RelationshipProperty`, and `CompositeProperty`.

Note: With the advent of Hybrid properties introduced in SQLAlchemy 0.7, as well as Core-level operator redefinition in SQLAlchemy 0.8, the use case for user-defined `PropComparator` instances is extremely rare. See hybrids_toplevel as well as types_operators.

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

```
# definition of custom PropComparator subclasses

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

class MyColumnComparator(ColumnProperty.Comparator):
    def __eq__(self, other):
        return self.__clause_element__() == other

class MyRelationshipComparator(RelationshipProperty.Comparator):
    def any(self, expression):
        "define the 'any' operation"
        # ...

class MyCompositeComparator(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__())])


# application of custom PropComparator subclasses

from sqlalchemy.orm import column_property, relationship, composite
from sqlalchemy import Column, String

class SomeMappedClass(Base):
    some_column = column_property(Column("some_column", String),
                        comparator_factory=MyColumnComparator)

    some_relationship = relationship(SomeOtherClass,
                        comparator_factory=MyRelationshipComparator)

    some_composite = composite(
            Column("a", String), Column("b", String),
            comparator_factory=MyCompositeComparator
        )
```

Note that for column-level operator redefinition, it's usually simpler to define the operators at the Core level, using the `TypeEngine.comparator_factory` attribute. See types_operators for more detail.

See also:

`ColumnProperty.Comparator`

`RelationshipProperty.Comparator`

`CompositeProperty.Comparator`

`ColumnOperators`

types_operators

`TypeEngine.comparator_factory`

adapt_to_entity(*adapt_to_entity*)
Return a copy of this PropComparator which will use the given `AliasedInsp` to produce corresponding expressions.

adapter
Produce a callable that adapts column expressions to suit an aliased version of this comparator.

all_()
Produce a `all_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == mytable.c.somearray.all_()

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().all_()
```

See also:

`all_()` - standalone version

`any_()` - ANY operator

New in version 1.1.

any(*criterion=None*, ***kwargs*)
Return true if this collection contains any member that meets the given criterion.

The usual implementation of `any()` is `RelationshipProperty.Comparator.any()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

any_()
Produce a `any_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == mytable.c.somearray.any_()

# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().any_()
```

See also:

`any_()` - standalone version

all_() - ALL operator

New in version 1.1.

asc()
Produce a asc() clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)
Produce a between() clause against the parent object, given the lower and upper range.

bool_op(*opstring*, *precedence=0*)
Return a custom boolean operator.

This method is shorthand for calling Operators.op() and passing the Operators.op.is_comparison flag with True.

New in version 1.2.0b3.

See also:

Operators.op()

collate(*collation*)
Produce a collate() clause against the parent object, given the collation string.

See also:

collate()

concat(*other*)
Implement the 'concat' operator.

In a column context, produces the clause a || b, or uses the concat() operator on MySQL.

contains(*other*, ***kwargs*)
Implement the 'contains' operator.

Produces a LIKE expression that tests against a match for the middle of a string value:

```
column LIKE '%' || <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.contains("foobar"))
```

Since the operator uses LIKE, wildcard characters "%" and "_" that are present inside the <other> expression will behave like wildcards as well. For literal string values, the ColumnOperators.contains.autoescape flag may be set to True to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the ColumnOperators.contains.escape parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.contains.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

```
somecolumn.contains("foo%bar", autoescape=True)
```

Will render as:

```
somecolumn LIKE '%' || :param || '%' ESCAPE '/'
```

With the value of :param as "foo/%bar".

New in version 1.2.

Changed in version 1.2.0: The ColumnOperators.contains.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.contains.escape parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

```
somecolumn.contains("foo/%bar", escape="^")
```

  Will render as:

```
somecolumn LIKE '%' || :param || '%' ESCAPE '^'
```

  The parameter may also be combined with ColumnOperators.contains.autoescape:

```
somecolumn.contains("foo%bar^bat", escape="^", autoescape=True)
```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.endswith()

ColumnOperators.like()

desc()
Produce a desc() clause against the parent object.

distinct()
Produce a distinct() clause against the parent object.

endswith(*other*, ***kwargs*)
Implement the 'endswith' operator.

Produces a LIKE expression that tests against a match for the end of a string value:

```
column LIKE '%' || <other>
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.endswith("foobar"))
```

Since the operator uses LIKE, wildcard characters "%" and "_" that are present inside the <other> expression will behave like wildcards as well. For literal string values, the

ColumnOperators.endswith.autoescape flag may be set to True to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the ColumnOperators.endswith.escape parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.endswith.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.endswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.endswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.endswith.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.endswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.endswith.autoescape:

  ```
  somecolumn.endswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.contains()

ColumnOperators.like()

has(*criterion=None*, ***kwargs*)
Return true if this element references a member which meets the given criterion.

The usual implementation of `has()` is `RelationshipProperty.Comparator.has()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

`ilike`(*other*, *escape=None*)

Implement the `ilike` operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

```
lower(a) LIKE lower(other)
```

Or on backends that support the ILIKE operator:

```
a ILIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

  ```
  somecolumn.ilike("foo/%bar", escape="/")
  ```

See also:

`ColumnOperators.like()`

`in_`(*other*)

Implement the `in` operator.

In a column context, produces the clause `a IN other`. "other" may be a tuple/list of column expressions, or a `select()` construct.

In the case that `other` is an empty sequence, the compiler produces an "empty in" expression. This defaults to the expression "1 != 1" to produce false in all cases. The `create_engine.empty_in_strategy` may be used to alter this behavior.

Changed in version 1.2: The `ColumnOperators.in_()` and `ColumnOperators.notin_()` operators now produce a "static" expression for an empty IN sequence by default.

`is_`(*other*)

Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

`ColumnOperators.isnot()`

`is_distinct_from`(*other*)

Implement the `IS DISTINCT FROM` operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

New in version 1.1.

isnot(*other*)
Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

`ColumnOperators.is_()`

isnot_distinct_from(*other*)
Implement the `IS NOT DISTINCT FROM` operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

New in version 1.1.

like(*other*, *escape=None*)
Implement the `like` operator.

In a column context, produces the expression:

```
a LIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

  ```
  somecolumn.like("foo/%bar", escape="/")
  ```

See also:

`ColumnOperators.ilike()`

match(*other*, ***kwargs*)
Implements a database-specific 'match' operator.

`match()` attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- PostgreSQL - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
implement the NOT ILIKE operator.

This is equivalent to using negation with ColumnOperators.ilike(), i.e. ~x.ilike(y).

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)
implement the NOT IN operator.

This is equivalent to using negation with ColumnOperators.in_(), i.e. ~x.in_(y).

In the case that other is an empty sequence, the compiler produces an "empty not in" expression. This defaults to the expression "1 = 1" to produce true in all cases. The create_engine.empty_in_strategy may be used to alter this behavior.

Changed in version 1.2: The ColumnOperators.in_() and ColumnOperators.notin_() operators now produce a "static" expression for an empty IN sequence by default.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)
implement the NOT LIKE operator.

This is equivalent to using negation with ColumnOperators.like(), i.e. ~x.like(y).

New in version 0.8.

See also:

ColumnOperators.like()

nullsfirst()
Produce a nullsfirst() clause against the parent object.

nullslast()
Produce a nullslast() clause against the parent object.

of_type(*class_*)
Redefine this object in terms of a polymorphic subclass.

Returns a new PropComparator from which further criterion can be evaluated.

e.g.:

```
query.join(Company.employees.of_type(Engineer)).\
   filter(Engineer.name=='foo')
```

Parameters **class_** – a class or mapper indicating that criterion will be against this specific subclass.

See also:

inheritance_of_type

op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like `==`, `>`, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.
- **return_type** – a `TypeEngine` class or object that will force the return type of an expression produced by this operator to be of that type. By default, operators that specify `Operators.op.is_comparison` will resolve to `Boolean`, and those that do not will be of the same type as the left-hand operand.

  New in version 1.2.0b3: - added the `Operators.op.return_type` argument.

See also:

types_operators

relationship_custom_operator

operate(*op*, **other*, ***kwargs*)

Operate on an argument.

This is the lowest level of operation, raises `NotImplementedError` by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding `ColumnOperators` to apply `func.lower()` to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as `ColumnOperators.contains()`.

reverse_operate(*op*, *other*, ***kwargs*)
Reverse operate on an argument.

Usage is the same as `operate()`.

startswith(*other*, ***kwargs*)
Implement the `startswith` operator.

Produces a LIKE expression that tests against a match for the start of a string value:

```
column LIKE <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.startswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.startswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.startswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.startswith.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.startswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '/'
  ```

  With the value of :param as `"foo/%bar"`.

  New in version 1.2.

  Changed in version 1.2.0: The `ColumnOperators.startswith.autoescape` parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the `ColumnOperators.startswith.escape` parameter.
- **escape** – a character which when given will render with the `ESCAPE` keyword to establish that character as the escape character. This character can then be placed preceding occurrences of `%` and `_` to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.startswith("foo/%bar", escape="^")
  ```

  Will render as:

```
somecolumn LIKE :param || '%' ESCAPE '^'
```

The parameter may also be combined with `ColumnOperators.startswith.autoescape`:

```
somecolumn.startswith("foo%bar^bat", escape="^", autoescape=True)
```

Where above, the given literal parameter will be converted to `"foo^%bar^^bat"` before being passed to the database.

See also:

`ColumnOperators.endswith()`

`ColumnOperators.contains()`

`ColumnOperators.like()`

class sqlalchemy.orm.properties.**RelationshipProperty**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign_keys=None*, *uselist=None*, *order_by=False*, *backref=None*, *back_populates=None*, *post_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy='select'*, *collection_class=None*, *passive_deletes=False*, *passive_updates=True*, *remote_side=None*, *enable_typechecks=True*, *join_depth=None*, *comparator_factory=None*, *single_parent=False*, *innerjoin=False*, *distinct_target_key=None*, *doc=None*, *active_history=False*, *cascade_backrefs=True*, *load_on_pending=False*, *bake_queries=True*, *_local_remote_pairs=None*, *query_class=None*, *info=None*)

Describes an object property that holds a single item or list of items that correspond to a related database table.

Public constructor is the `orm.relationship()` function.

See also:

relationship_config_toplevel

class Comparator(*prop*, *parentmapper*, *adapt_to_entity=None*, *of_type=None*)

Produce boolean, comparison, and other operators for `RelationshipProperty` attributes.

See the documentation for `PropComparator` for a brief overview of ORM level operator definition.

See also:

`PropComparator`

`ColumnProperty.Comparator`

`ColumnOperators`

types_operators

TypeEngine.comparator_factory

adapter
Produce a callable that adapts column expressions to suit an aliased version of this comparator.

all_()
Produce a all_() clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == mytable.c.somearray.all_()

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().all_()
```

See also:

all_() - standalone version

any_() - ANY operator

New in version 1.1.

any(*criterion=None*, ***kwargs*)
Produce an expression that tests a collection against particular criterion, using EXISTS.

An expression like:

```
session.query(MyClass).filter(
    MyClass.somereference.any(SomeRelated.x==2)
)
```

Will produce a query like:

```
SELECT * FROM my_table WHERE
EXISTS (SELECT 1 FROM related WHERE related.my_id=my_table.id
AND related.x=2)
```

Because any() uses a correlated subquery, its performance is not nearly as good when compared against large target tables as that of using a join.

any() is particularly useful for testing for empty collections:

```
session.query(MyClass).filter(
    ~MyClass.somereference.any()
)
```

will produce:

```
SELECT * FROM my_table WHERE
NOT EXISTS (SELECT 1 FROM related WHERE
related.my_id=my_table.id)
```

any() is only valid for collections, i.e. a relationship() that has uselist=True. For scalar references, use has().

any_()
Produce a any_() clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == mytable.c.somearray.any_()

# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().any_()
```

See also:

`any_()` - standalone version

`all_()` - ALL operator

New in version 1.1.

`asc()`
Produce a `asc()` clause against the parent object.

`between`(*cleft*, *cright*, *symmetric=False*)
Produce a `between()` clause against the parent object, given the lower and upper range.

`bool_op`(*opstring*, *precedence=0*)
Return a custom boolean operator.

This method is shorthand for calling `Operators.op()` and passing the `Operators.op.is_comparison` flag with True.

New in version 1.2.0b3.

See also:

`Operators.op()`

`collate`(*collation*)
Produce a `collate()` clause against the parent object, given the collation string.

See also:

`collate()`

`concat`(*other*)
Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

`contains`(*other*, ***kwargs*)
Return a simple expression that tests a collection for containment of a particular item.

`contains()` is only valid for a collection, i.e. a `relationship()` that implements one-to-many or many-to-many with `uselist=True`.

When used in a simple one-to-many context, an expression like:

```
MyClass.contains(other)
```

Produces a clause like:

```
mytable.id == <some id>
```

Where `<some id>` is the value of the foreign key attribute on `other` which refers to the primary key of its parent object. From this it follows that `contains()` is very useful when used with simple one-to-many operations.

For many-to-many operations, the behavior of `contains()` has more caveats. The association table will be rendered in the statement, producing an "implicit" join, that is, includes multiple tables in the FROM clause which are equated in the WHERE clause:

```
query(MyClass).filter(MyClass.contains(other))
```

Produces a query like:

```
SELECT * FROM my_table, my_association_table AS
my_association_table_1 WHERE
my_table.id = my_association_table_1.parent_id
AND my_association_table_1.child_id = <some id>
```

Where `<some id>` would be the primary key of `other`. From the above, it is clear that `contains()` will **not** work with many-to-many collections when used in queries that move beyond simple AND conjunctions, such as multiple `contains()` expressions joined by OR. In such cases subqueries or explicit "outer joins" will need to be used instead. See `any()` for a less-performant alternative using EXISTS, or refer to `Query.outerjoin()` as well as ormtutorial_joins for more details on constructing outer joins.

`desc()`
Produce a `desc()` clause against the parent object.

`distinct()`
Produce a `distinct()` clause against the parent object.

`endswith`(*other*, ***kwargs*)
Implement the 'endswith' operator.

Produces a LIKE expression that tests against a match for the end of a string value:

```
column LIKE '%' || <other>
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.endswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.endswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.endswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.endswith.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.endswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '/'
  ```

  With the value of :param as `"foo/%bar"`.

  New in version 1.2.

Changed in version 1.2.0: The `ColumnOperators.endswith.autoescape` parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the `ColumnOperators.endswith.escape` parameter.

- **escape** – a character which when given will render with the `ESCAPE` keyword to establish that character as the escape character. This character can then be placed preceding occurrences of `%` and `_` to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.endswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '^'
  ```

  The parameter may also be combined with `ColumnOperators.endswith.autoescape`:

  ```
  somecolumn.endswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to `"foo^%bar^^bat"` before being passed to the database.

See also:

`ColumnOperators.startswith()`

`ColumnOperators.contains()`

`ColumnOperators.like()`

has(*criterion=None*, ***kwargs*)

Produce an expression that tests a scalar reference against particular criterion, using EXISTS.

An expression like:

```
session.query(MyClass).filter(
    MyClass.somereference.has(SomeRelated.x==2)
)
```

Will produce a query like:

```
SELECT * FROM my_table WHERE
EXISTS (SELECT 1 FROM related WHERE
related.id==my_table.related_id AND related.x=2)
```

Because `has()` uses a correlated subquery, its performance is not nearly as good when compared against large target tables as that of using a join.

`has()` is only valid for scalar references, i.e. a `relationship()` that has `uselist=False`. For collection references, use `any()`.

ilike(*other*, *escape=None*)

Implement the `ilike` operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

```
lower(a) LIKE lower(other)
```

Or on backends that support the ILIKE operator:

```
a ILIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)
Produce an IN clause - this is not implemented for relationship()-based attributes at this time.

is_(*other*)
Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

is_distinct_from(*other*)
Implement the IS DISTINCT FROM operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

New in version 1.1.

isnot(*other*)
Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

isnot_distinct_from(*other*)
Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

New in version 1.1.

like(*other*, *escape=None*)
Implement the like operator.

In a column context, produces the expression:

```
a LIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

mapper

The target Mapper referred to by this RelationshipProperty.Comparator.

This is the "target" or "remote" side of the relationship().

match(*other*, ***kwargs*)

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- PostgreSQL - renders x @@ to_tsquery(y)
- MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
- Oracle - renders CONTAINS(x, y)
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)

implement the NOT ILIKE operator.

This is equivalent to using negation with ColumnOperators.ilike(), i.e. ~x.ilike(y).

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)

implement the NOT IN operator.

This is equivalent to using negation with ColumnOperators.in_(), i.e. ~x.in_(y).

In the case that **other** is an empty sequence, the compiler produces an "empty not in" expression. This defaults to the expression "1 = 1" to produce true in all cases. The create_engine.empty_in_strategy may be used to alter this behavior.

Changed in version 1.2: The ColumnOperators.in_() and ColumnOperators.notin_() operators now produce a "static" expression for an empty IN sequence by default.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)

implement the NOT LIKE operator.

This is equivalent to using negation with ColumnOperators.like(), i.e. ~x.like(y).

New in version 0.8.

See also:

ColumnOperators.like()

`nullsfirst()`
Produce a `nullsfirst()` clause against the parent object.

`nullslast()`
Produce a `nullslast()` clause against the parent object.

`of_type(`*cls*`)`
Redefine this object in terms of a polymorphic subclass.

See `PropComparator.of_type()` for an example.

`op(`*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*`)`
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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`.

Parameters

- **`operator`** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **`precedence`** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **`is_comparison`** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like `==`, `>`, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.
- **`return_type`** – a `TypeEngine` class or object that will force the return type of an expression produced by this operator to be of that type. By default, operators that specify `Operators.op.is_comparison` will resolve to `Boolean`, and those that do not will be of the same type as the left-hand operand.

  New in version 1.2.0b3: - added the `Operators.op.return_type` argument.

See also:

types_operators

relationship_custom_operator

`operate(`*op*, **other*, ***kwargs*`)`
Operate on an argument.

This is the lowest level of operation, raises `NotImplementedError` by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding `ColumnOperators` to apply `func.lower()` to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as `ColumnOperators.contains()`.

reverse_operate(*op*, *other*, ***kwargs*)

Reverse operate on an argument.

Usage is the same as `operate()`.

startswith(*other*, ***kwargs*)

Implement the `startswith` operator.

Produces a LIKE expression that tests against a match for the start of a string value:

```
column LIKE <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.startswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.startswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.startswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.startswith.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.startswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '/'
  ```

  With the value of :param as `"foo/%bar"`.

  New in version 1.2.

  Changed in version 1.2.0: The `ColumnOperators.startswith.autoescape` parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the `ColumnOperators.startswith.escape` parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.startswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.startswith.autoescape:

  ```
  somecolumn.startswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.endswith()

ColumnOperators.contains()

ColumnOperators.like()

cascade
Return the current cascade setting for this RelationshipProperty.

class_attribute
Return the class-bound descriptor corresponding to this MapperProperty.

This is basically a getattr() call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this MapperProperty were named addresses, and the class to which it is mapped is User, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
>>> User.addresses.property is addresses_property
True
```

extension_type = symbol('NOT_EXTENSION')

init()
Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

mapper
Return the targeted Mapper for this RelationshipProperty.

This is a lazy-initializing static attribute.

set_parent(*parent*, *init*)
Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

table
Return the selectable linked to this `RelationshipProperty` object's target `Mapper`.

Deprecated since version 0.7: Use .target

class sqlalchemy.orm.descriptor_props.SynonymProperty(*name*, *map_column=None*, *descriptor=None*, *comparator_factory=None*, *doc=None*, *info=None*)

cascade_iterator(*type_*, *state*, *visited_instances=None*, *halt_on=None*)
Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade_iterator is called.

This method typically only applies to RelationshipProperty.

class_attribute
Return the class-bound descriptor corresponding to this `MapperProperty`.

This is basically a `getattr()` call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this `MapperProperty` were named `addresses`, and the class to which it is mapped is `User`, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
>>> User.addresses.property is addresses_property
True
```

create_row_processor(*context*, *path*, *mapper*, *result*, *adapter*, *populators*)
Produce row processing functions and append to the given set of populators lists.

do_init()
Perform subclass-specific initialization post-mapper-creation steps.

This is a template method called by the `MapperProperty` object's init() method.

extension_type = symbol('NOT_EXTENSION')

init()
Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

merge(*session*, *source_state*, *source_dict*, *dest_state*, *dest_dict*, *load*, *_recursive*, *_resolve_conflict_map*)
Merge the attribute represented by this `MapperProperty` from source to destination object.

post_instrument_class(*mapper*)
Perform instrumentation adjustments that need to occur after init() has completed.

The given Mapper is the Mapper invoking the operation, which may not be the same Mapper as self.parent in an inheritance scenario; however, Mapper will always at least be a sub-mapper of self.parent.

This method is typically used by StrategizedProperty, which delegates it to LoaderStrategy.init_class_attribute() to perform final setup on the class-bound InstrumentedAttribute.

setup(*context*, *entity*, *path*, *adapter*, ***kwargs*)
Called by Query for the purposes of constructing a SQL statement.

Each MapperProperty associated with the target mapper processes the statement referenced by the query context, adding columns and/or criterion as appropriate.

class sqlalchemy.orm.query.**QueryContext**(*query*)

class sqlalchemy.orm.attributes.**QueryableAttribute**(*class_*, *key*, *impl=None*, *comparator=None*, *parententity=None*, *of_type=None*)
Base class for descriptor objects that intercept attribute events on behalf of a `MapperProperty` object. The actual `MapperProperty` is accessible via the `QueryableAttribute.property` attribute.

See also:

`InstrumentedAttribute`

`MapperProperty`

`Mapper.all_orm_descriptors`

`Mapper.attrs`

adapter
Produce a callable that adapts column expressions to suit an aliased version of this comparator.

all_()
Produce a `all_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == mytable.c.somearray.all_()

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().all_()
```

See also:

`all_()` - standalone version

`any_()` - ANY operator

New in version 1.1.

any(*criterion=None*, ***kwargs*)
Return true if this collection contains any member that meets the given criterion.

The usual implementation of `any()` is `RelationshipProperty.Comparator.any()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

any_()
Produce a `any_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == mytable.c.somearray.any_()
```

```
# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().any_()
```

See also:

`any_()` - standalone version

`all_()` - ALL operator

New in version 1.1.

`asc()`
Produce a `asc()` clause against the parent object.

`between`(*cleft*, *cright*, *symmetric=False*)
Produce a `between()` clause against the parent object, given the lower and upper range.

`bool_op`(*opstring*, *precedence=0*)
Return a custom boolean operator.

This method is shorthand for calling `Operators.op()` and passing the `Operators.op.is_comparison` flag with True.

New in version 1.2.0b3.

See also:

`Operators.op()`

`collate`(*collation*)
Produce a `collate()` clause against the parent object, given the collation string.

See also:

`collate()`

`concat`(*other*)
Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

`contains`(*other*, ***kwargs*)
Implement the 'contains' operator.

Produces a LIKE expression that tests against a match for the middle of a string value:

```
column LIKE '%' || <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.contains("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.contains.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.contains.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **`other`** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.contains.autoescape` flag is set to True.

- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.contains("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.contains.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.contains.escape parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.contains("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.contains.autoescape:

  ```
  somecolumn.contains("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.endswith()

ColumnOperators.like()

desc()
Produce a desc() clause against the parent object.

distinct()
Produce a distinct() clause against the parent object.

endswith(*other*, ***kwargs*)
Implement the 'endswith' operator.

Produces a LIKE expression that tests against a match for the end of a string value:

```
column LIKE '%' || <other>
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.endswith("foobar"))
```

Since the operator uses LIKE, wildcard characters "%" and "_" that are present inside the <other> expression will behave like wildcards as well. For literal string values, the ColumnOperators.endswith.autoescape flag may be set to True to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the ColumnOperators.endswith.escape parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.endswith.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.endswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.endswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.endswith.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.endswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.endswith.autoescape:

  ```
  somecolumn.endswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.contains()

ColumnOperators.like()

extension_type = symbol('NOT_EXTENSION')

has(*criterion=None*, ***kwargs*)
Return true if this element references a member which meets the given criterion.

The usual implementation of **has()** is **RelationshipProperty.Comparator.has()**.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

ilike(*other*, *escape=None*)
Implement the **ilike** operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

```
lower(a) LIKE lower(other)
```

Or on backends that support the ILIKE operator:

```
a ILIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the **ESCAPE** keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)
Implement the **in** operator.

In a column context, produces the clause **a IN other**. "other" may be a tuple/list of column expressions, or a **select()** construct.

In the case that **other** is an empty sequence, the compiler produces an "empty in" expression. This defaults to the expression "1 != 1" to produce false in all cases. The **create_engine.empty_in_strategy** may be used to alter this behavior.

Changed in version 1.2: The **ColumnOperators.in_()** and **ColumnOperators.notin_()** operators now produce a "static" expression for an empty IN sequence by default.

info
Return the 'info' dictionary for the underlying SQL element.

The behavior here is as follows:

- If the attribute is a column-mapped property, i.e. **ColumnProperty**, which is mapped directly to a schema-level **Column** object, this attribute will return the **SchemaItem.info** dictionary associated with the core-level **Column** object.

- If the attribute is a `ColumnProperty` but is mapped to any other kind of SQL expression other than a `Column`, the attribute will refer to the `MapperProperty.info` dictionary associated directly with the `ColumnProperty`, assuming the SQL expression itself does not have its own `.info` attribute (which should be the case, unless a user-defined SQL construct has defined one).
- If the attribute refers to any other kind of `MapperProperty`, including `RelationshipProperty`, the attribute will refer to the `MapperProperty.info` dictionary associated with that `MapperProperty`.
- To access the `MapperProperty.info` dictionary of the `MapperProperty` unconditionally, including for a `ColumnProperty` that's associated directly with a `schema.Column`, the attribute can be referred to using `QueryableAttribute.property` attribute, as `MyClass.someattribute.property.info`.

New in version 0.8.0.

See also:

`SchemaItem.info`

`MapperProperty.info`

`is_`(*other*)
Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

`ColumnOperators.isnot()`

`is_distinct_from`(*other*)
Implement the `IS DISTINCT FROM` operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

New in version 1.1.

`isnot`(*other*)
Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

`ColumnOperators.is_()`

`isnot_distinct_from`(*other*)
Implement the `IS NOT DISTINCT FROM` operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

New in version 1.1.

`like`(*other*, *escape=None*)
Implement the `like` operator.

In a column context, produces the expression:

```
a LIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)
Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- PostgreSQL - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
implement the NOT ILIKE operator.

This is equivalent to using negation with ColumnOperators.ilike(), i.e. ~x.ilike(y).

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)
implement the NOT IN operator.

This is equivalent to using negation with ColumnOperators.in_(), i.e. ~x.in_(y).

In the case that other is an empty sequence, the compiler produces an "empty not in" expression. This defaults to the expression "1 = 1" to produce true in all cases. The create_engine.empty_in_strategy may be used to alter this behavior.

Changed in version 1.2: The ColumnOperators.in_() and ColumnOperators.notin_() operators now produce a "static" expression for an empty IN sequence by default.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)
implement the NOT LIKE operator.

This is equivalent to using negation with ColumnOperators.like(), i.e. ~x.like(y).

New in version 0.8.

See also:

`ColumnOperators.like()`

nullsfirst()
Produce a `nullsfirst()` clause against the parent object.

nullslast()
Produce a `nullslast()` clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like `==`, `>`, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.
- **return_type** – a `TypeEngine` class or object that will force the return type of an expression produced by this operator to be of that type. By default, operators that specify `Operators.op.is_comparison` will resolve to `Boolean`, and those that do not will be of the same type as the left-hand operand.

  New in version 1.2.0b3: - added the `Operators.op.return_type` argument.

See also:

types_operators

relationship_custom_operator

parent
Return an inspection instance representing the parent.

This will be either an instance of `Mapper` or `AliasedInsp`, depending upon the nature of the parent entity which this attribute is associated with.

property
Return the MapperProperty associated with this QueryableAttribute.

Return values here will commonly be instances of ColumnProperty or RelationshipProperty.

startswith(*other*, ***kwargs*)
Implement the startswith operator.

Produces a LIKE expression that tests against a match for the start of a string value:

```
column LIKE <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.startswith("foobar"))
```

Since the operator uses LIKE, wildcard characters "%" and "_" that are present inside the <other> expression will behave like wildcards as well. For literal string values, the ColumnOperators.startswith.autoescape flag may be set to True to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the ColumnOperators.startswith.escape parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.startswith.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.startswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.startswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.startswith.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.startswith("foo/%bar", escape="^")
  ```

  Will render as:

```
somecolumn LIKE :param || '%' ESCAPE '^'
```

The parameter may also be combined with `ColumnOperators.startswith.autoescape`:

```
somecolumn.startswith("foo%bar^bat", escape="^", autoescape=True)
```

Where above, the given literal parameter will be converted to `"foo^%bar^^bat"` before being passed to the database.

See also:

`ColumnOperators.endswith()`

`ColumnOperators.contains()`

`ColumnOperators.like()`

class sqlalchemy.orm.session.**UOWTransaction**(*session*)

filter_states_for_dep(*dep*, *states*)
Filter the given list of InstanceStates to those relevant to the given DependencyProcessor.

finalize_flush_changes()
mark processed objects as clean / deleted after a successful flush().

this method is called within the flush() method after the execute() method has succeeded and the transaction has been committed.

get_attribute_history(*state*, *key*, *passive=symbol('PASSIVE_NO_INITIALIZE')*)
facade to attributes.get_state_history(), including caching of results.

is_deleted(*state*)
return true if the given state is marked as deleted within this uowtransaction.

remove_state_actions(*state*)
remove pending actions for a state from the uowtransaction.

was_already_deleted(*state*)
return true if the given state is expired and was deleted previously.

2.6.3 ORM Exceptions

SQLAlchemy ORM exceptions.

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

exception sqlalchemy.orm.exc.**DetachedInstanceError**(**arg*, ***kw*)
An attempt to access unloaded attributes on a mapped instance that is detached.

exception sqlalchemy.orm.exc.**FlushError**(**arg*, ***kw*)
A invalid condition was detected during flush().

exception sqlalchemy.orm.exc.**MultipleResultsFound**(**arg*, ***kw*)
A single database result was required but more than one were found.

sqlalchemy.orm.exc.**NO_STATE = (<class 'AttributeError'>, <class 'KeyError'>)**
Exception types that may be raised by instrumentation implementations.

exception sqlalchemy.orm.exc.**NoResultFound**(**arg*, ***kw*)
A database result was required but none was found.

exception sqlalchemy.orm.exc.ObjectDeletedError(*base*, *state*, *msg=None*)
A refresh operation failed to retrieve the database row corresponding to an object's known primary key identity.

A refresh operation proceeds when an expired attribute is accessed on an object, or when `Query.get()` is used to retrieve an object which is, upon retrieval, detected as expired. A SELECT is emitted for the target row based on primary key; if no row is returned, this exception is raised.

The true meaning of this exception is simply that no row exists for the primary key identifier associated with a persistent object. The row may have been deleted, or in some cases the primary key updated to a new value, outside of the ORM's management of the target object.

exception sqlalchemy.orm.exc.ObjectDereferencedError(**arg*, ***kw*)
An operation cannot complete due to an object being garbage collected.

exception sqlalchemy.orm.exc.StaleDataError(**arg*, ***kw*)
An operation encountered database state that is unaccounted for.

Conditions which cause this to happen include:

- 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.
- A object is detached from its parent object, however the object was previously attached to a different parent identity which was garbage collected, and a decision cannot be made if the new parent was really the most recent "parent".

  New in version 0.7.4.

exception sqlalchemy.orm.exc.UnmappedClassError(*cls*, *msg=None*)
An mapping operation was requested for an unknown class.

exception sqlalchemy.orm.exc.UnmappedColumnError(**arg*, ***kw*)
Mapping operation was requested on an unknown column.

exception sqlalchemy.orm.exc.UnmappedError(**arg*, ***kw*)
Base for exceptions that involve expected mappings not present.

exception sqlalchemy.orm.exc.UnmappedInstanceError(*base*, *obj*, *msg=None*)
An mapping operation was requested for an unknown instance.

2.6.4 Deprecated ORM Event Interfaces

This section describes the class-based ORM event interface which first existed in SQLAlchemy 0.1, which progressed with more kinds of events up until SQLAlchemy 0.5. The non-ORM analogue is described at dep_interfaces_core_toplevel.

Deprecated since version 0.7: As of SQLAlchemy 0.7, the new event system described in event_toplevel replaces the extension/proxy/listener system, providing a consistent interface to all events without the need for subclassing.

Mapper Events

class sqlalchemy.orm.interfaces.MapperExtension
Base implementation for `Mapper` event hooks.

Note: `MapperExtension` is deprecated. Please refer to `event.listen()` as well as `MapperEvents`.

New extension classes subclass `MapperExtension` and are specified using the `extension` mapper() argument, which is a single `MapperExtension` or a list of such:

```
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())
```

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:

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

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.

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.

init_failed(*mapper*, *class_*, *oldinit*, *instance*, *args*, *kwargs*)
Receive an instance when its 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 its 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.

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.

Session Events

class sqlalchemy.orm.interfaces.**SessionExtension**
Base implementation for Session event hooks.

Note: SessionExtension is deprecated. Please refer to event.listen() as well as SessionEvents.

Subclasses may be installed into a Session (or sessionmaker) using the extension keyword argument:

```
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.

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 occurred.

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 occurred, 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 occurred.

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.

Attribute Events

class sqlalchemy.orm.interfaces.AttributeExtension
Base implementation for `AttributeImpl` event hooks, events that fire upon attribute mutations in user code.

Note: `AttributeExtension` is deprecated. Please refer to `event.listen()` as well as `AttributeEvents`.

`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.

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

active_history = True
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.

2.7 ORM Extensions

SQLAlchemy has a variety of ORM extensions available, which add additional functionality to the core behavior.

The extensions build almost entirely on public core and ORM APIs and users should be encouraged to read their source code to further their understanding of their behavior. In particular the "Horizontal Sharding", "Hybrid Attributes", and "Mutation Tracking" extensions are very succinct.

2.7.1 Association Proxy

`associationproxy` is used to create a read/write view of a target attribute across a relationship. It essentially conceals the usage of a "middle" attribute between two endpoints, and can be used to cherry-pick fields from a collection of related objects or to reduce the verbosity of using the association object pattern. Applied creatively, the association proxy allows the construction of sophisticated collections and dictionary views of virtually any geometry, persisted to the database using standard, transparently configured relational patterns.

Simplifying Scalar Collections

Consider a many-to-many mapping between two classes, `User` and `Keyword`. Each `User` can have any number of `Keyword` objects, and vice-versa (the many-to-many pattern is described at relationships_many_to_many):

```
from sqlalchemy import Column, Integer, String, ForeignKey, Table
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))
    kw = relationship("Keyword", secondary=lambda: userkeywords_table)

    def __init__(self, name):
        self.name = name

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword

userkeywords_table = Table('userkeywords', Base.metadata,
    Column('user_id', Integer, ForeignKey("user.id"),
```

```
          primary_key=True),
    Column('keyword_id', Integer, ForeignKey("keyword.id"),
          primary_key=True)
)
```

Reading and manipulating the collection of "keyword" strings associated with `User` requires traversal from each collection element to the `.keyword` attribute, which can be awkward:

```
>>> user = User('jek')
>>> user.kw.append(Keyword('cheese inspector'))
>>> print(user.kw)
[<__main__.Keyword object at 0x12bf830>]
>>> print(user.kw[0].keyword)
cheese inspector
>>> print([keyword.keyword for keyword in user.kw])
['cheese inspector']
```

The `association_proxy` is applied to the `User` class to produce a "view" of the `kw` relationship, which only exposes the string value of `.keyword` associated with each `Keyword` object:

```
from sqlalchemy.ext.associationproxy import association_proxy

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))
    kw = relationship("Keyword", secondary=lambda: userkeywords_table)

    def __init__(self, name):
        self.name = name

    # proxy the 'keyword' attribute from the 'kw' relationship
    keywords = association_proxy('kw', 'keyword')
```

We can now reference the `.keywords` collection as a listing of strings, which is both readable and writable. New `Keyword` objects are created for us transparently:

```
>>> user = User('jek')
>>> user.keywords.append('cheese inspector')
>>> user.keywords
['cheese inspector']
>>> user.keywords.append('snack ninja')
>>> user.kw
[<__main__.Keyword object at 0x12cdd30>, <__main__.Keyword object at 0x12cde30>]
```

The `AssociationProxy` object produced by the `association_proxy()` function is an instance of a Python descriptor. It is always declared with the user-defined class being mapped, regardless of whether Declarative or classical mappings via the `mapper()` function are used.

The proxy functions by operating upon the underlying mapped attribute or collection in response to operations, and changes made via the proxy are immediately apparent in the mapped attribute, as well as vice versa. The underlying attribute remains fully accessible.

When first accessed, the association proxy performs introspection operations on the target collection so that its behavior corresponds correctly. Details such as if the locally proxied attribute is a collection (as is typical) or a scalar reference, as well as if the collection acts like a set, list, or dictionary is taken into account, so that the proxy should act just like the underlying collection or attribute does.

Creation of New Values

When a list append() event (or set add(), dictionary __setitem__(), or scalar assignment event) is intercepted by the association proxy, it instantiates a new instance of the "intermediary" object using its constructor, passing as a single argument the given value. In our example above, an operation like:

```
user.keywords.append('cheese inspector')
```

Is translated by the association proxy into the operation:

```
user.kw.append(Keyword('cheese inspector'))
```

The example works here because we have designed the constructor for `Keyword` to accept a single positional argument, `keyword`. For those cases where a single-argument constructor isn't feasible, the association proxy's creational behavior can be customized using the `creator` argument, which references a callable (i.e. Python function) that will produce a new object instance given the singular argument. Below we illustrate this using a lambda as is typical:

```
class User(Base):
    # ...

    # use Keyword(keyword=kw) on append() events
    keywords = association_proxy('kw', 'keyword',
                    creator=lambda kw: Keyword(keyword=kw))
```

The `creator` function accepts a single argument in the case of a list- or set- based collection, or a scalar attribute. In the case of a dictionary-based collection, it accepts two arguments, "key" and "value". An example of this is below in proxying_dictionaries.

Simplifying Association Objects

The "association object" pattern is an extended form of a many-to-many relationship, and is described at association_pattern. Association proxies are useful for keeping "association objects" out of the way during regular use.

Suppose our `userkeywords` table above had additional columns which we'd like to map explicitly, but in most cases we don't require direct access to these attributes. Below, we illustrate a new mapping which introduces the `UserKeyword` class, which is mapped to the `userkeywords` table illustrated earlier. This class adds an additional column `special_key`, a value which we occasionally want to access, but not in the usual case. We create an association proxy on the `User` class called `keywords`, which will bridge the gap from the `user_keywords` collection of `User` to the `.keyword` attribute present on each `UserKeyword`:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref

from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))

    # association proxy of "user_keywords" collection
    # to "keyword" attribute
    keywords = association_proxy('user_keywords', 'keyword')

    def __init__(self, name):
        self.name = name
```

```
class UserKeyword(Base):
    __tablename__ = 'user_keyword'
    user_id = Column(Integer, ForeignKey('user.id'), primary_key=True)
    keyword_id = Column(Integer, ForeignKey('keyword.id'), primary_key=True)
    special_key = Column(String(50))

    # bidirectional attribute/collection of "user"/"user_keywords"
    user = relationship(User,
                backref=backref("user_keywords",
                                cascade="all, delete-orphan")
            )

    # reference to the "Keyword" object
    keyword = relationship("Keyword")

    def __init__(self, keyword=None, user=None, special_key=None):
        self.user = user
        self.keyword = keyword
        self.special_key = special_key

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword

    def __repr__(self):
        return 'Keyword(%s)' % repr(self.keyword)
```

With the above configuration, we can operate upon the `.keywords` collection of each `User` object, and the usage of `UserKeyword` is concealed:

```
>>> user = User('log')
>>> for kw in (Keyword('new_from_blammo'), Keyword('its_big')):
...     user.keywords.append(kw)
...
>>> print(user.keywords)
[Keyword('new_from_blammo'), Keyword('its_big')]
```

Where above, each `.keywords.append()` operation is equivalent to:

```
>>> user.user_keywords.append(UserKeyword(Keyword('its_heavy')))
```

The `UserKeyword` association object has two attributes here which are populated; the `.keyword` attribute is populated directly as a result of passing the `Keyword` object as the first argument. The `.user` argument is then assigned as the `UserKeyword` object is appended to the `User.user_keywords` collection, where the bidirectional relationship configured between `User.user_keywords` and `UserKeyword.user` results in a population of the `UserKeyword.user` attribute. The `special_key` argument above is left at its default value of `None`.

For those cases where we do want `special_key` to have a value, we create the `UserKeyword` object explicitly. Below we assign all three attributes, where the assignment of `.user` has the effect of the `UserKeyword` being appended to the `User.user_keywords` collection:

```
>>> UserKeyword(Keyword('its_wood'), user, special_key='my special key')
```

The association proxy returns to us a collection of `Keyword` objects represented by all these operations:

```
>>> user.keywords
[Keyword('new_from_blammo'), Keyword('its_big'), Keyword('its_heavy'), Keyword('its_wood')]
```

Proxying to Dictionary Based Collections

The association proxy can proxy to dictionary based collections as well. SQLAlchemy mappings usually use the `attribute_mapped_collection()` collection type to create dictionary collections, as well as the extended techniques described in dictionary_collections.

The association proxy adjusts its behavior when it detects the usage of a dictionary-based collection. When new values are added to the dictionary, the association proxy instantiates the intermediary object by passing two arguments to the creation function instead of one, the key and the value. As always, this creation function defaults to the constructor of the intermediary class, and can be customized using the `creator` argument.

Below, we modify our `UserKeyword` example such that the `User.user_keywords` collection will now be mapped using a dictionary, where the `UserKeyword.special_key` argument will be used as the key for the dictionary. We then apply a `creator` argument to the `User.keywords` proxy so that these values are assigned appropriately when new elements are added to the dictionary:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm.collections import attribute_mapped_collection

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))

    # proxy to 'user_keywords', instantiating UserKeyword
    # assigning the new key to 'special_key', values to
    # 'keyword'.
    keywords = association_proxy('user_keywords', 'keyword',
                    creator=lambda k, v:
                                UserKeyword(special_key=k, keyword=v)
                )

    def __init__(self, name):
        self.name = name

class UserKeyword(Base):
    __tablename__ = 'user_keyword'
    user_id = Column(Integer, ForeignKey('user.id'), primary_key=True)
    keyword_id = Column(Integer, ForeignKey('keyword.id'), primary_key=True)
    special_key = Column(String)

    # bidirectional user/user_keywords relationships, mapping
    # user_keywords with a dictionary against "special_key" as key.
    user = relationship(User, backref=backref(
                    "user_keywords",
                    collection_class=attribute_mapped_collection("special_key"),
                    cascade="all, delete-orphan"
                    )
                )
    keyword = relationship("Keyword")

class Keyword(Base):
```

```
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword

    def __repr__(self):
        return 'Keyword(%s)' % repr(self.keyword)
```

We illustrate the `.keywords` collection as a dictionary, mapping the `UserKeyword.string_key` value to `Keyword` objects:

```
>>> user = User('log')

>>> user.keywords['sk1'] = Keyword('kw1')
>>> user.keywords['sk2'] = Keyword('kw2')

>>> print(user.keywords)
{'sk1': Keyword('kw1'), 'sk2': Keyword('kw2')}
```

Composite Association Proxies

Given our previous examples of proxying from relationship to scalar attribute, proxying across an association object, and proxying dictionaries, we can combine all three techniques together to give `User` a `keywords` dictionary that deals strictly with the string value of `special_key` mapped to the string `keyword`. Both the `UserKeyword` and `Keyword` classes are entirely concealed. This is achieved by building an association proxy on `User` that refers to an association proxy present on `UserKeyword`:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref

from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm.collections import attribute_mapped_collection

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))

    # the same 'user_keywords'->'keyword' proxy as in
    # the basic dictionary example
    keywords = association_proxy(
                'user_keywords',
                'keyword',
                creator=lambda k, v:
                            UserKeyword(special_key=k, keyword=v)
                )

    def __init__(self, name):
        self.name = name

class UserKeyword(Base):
    __tablename__ = 'user_keyword'
    user_id = Column(Integer, ForeignKey('user.id'), primary_key=True)
    keyword_id = Column(Integer, ForeignKey('keyword.id'),
                                                    primary_key=True)
    special_key = Column(String)
```

```
    user = relationship(User, backref=backref(
            "user_keywords",
            collection_class=attribute_mapped_collection("special_key"),
            cascade="all, delete-orphan"
            )
        )

    # the relationship to Keyword is now called
    # 'kw'
    kw = relationship("Keyword")

    # 'keyword' is changed to be a proxy to the
    # 'keyword' attribute of 'Keyword'
    keyword = association_proxy('kw', 'keyword')

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword
```

`User.keywords` is now a dictionary of string to string, where `UserKeyword` and `Keyword` objects are created and removed for us transparently using the association proxy. In the example below, we illustrate usage of the assignment operator, also appropriately handled by the association proxy, to apply a dictionary value to the collection at once:

```
>>> user = User('log')
>>> user.keywords = {
...     'sk1':'kw1',
...     'sk2':'kw2'
... }
>>> print(user.keywords)
{'sk1': 'kw1', 'sk2': 'kw2'}

>>> user.keywords['sk3'] = 'kw3'
>>> del user.keywords['sk2']
>>> print(user.keywords)
{'sk1': 'kw1', 'sk3': 'kw3'}

>>> # illustrate un-proxied usage
... print(user.user_keywords['sk3'].kw)
<__main__.Keyword object at 0x12ceb90>
```

One caveat with our example above is that because `Keyword` objects are created for each dictionary set operation, the example fails to maintain uniqueness for the `Keyword` objects on their string name, which is a typical requirement for a tagging scenario such as this one. For this use case the recipe UniqueObject, or a comparable creational strategy, is recommended, which will apply a "lookup first, then create" strategy to the constructor of the `Keyword` class, so that an already existing `Keyword` is returned if the given name is already present.

Querying with Association Proxies

The `AssociationProxy` features simple SQL construction capabilities which relate down to the underlying `relationship()` in use as well as the target attribute. For example, the `RelationshipProperty.Comparator.any()` and `RelationshipProperty.Comparator.has()` operations are available, and will produce a "nested" EXISTS clause, such as in our basic association object example:

```
>>> print(session.query(User).filter(User.keywords.any(keyword='jek')))
SELECT user.id AS user_id, user.name AS user_name
FROM user
WHERE EXISTS (SELECT 1
FROM user_keyword
WHERE user.id = user_keyword.user_id AND (EXISTS (SELECT 1
FROM keyword
WHERE keyword.id = user_keyword.keyword_id AND keyword.keyword = :keyword_1)))
```

For a proxy to a scalar attribute, `__eq__()` is supported:

```
>>> print(session.query(UserKeyword).filter(UserKeyword.keyword == 'jek'))
SELECT user_keyword.*
FROM user_keyword
WHERE EXISTS (SELECT 1
    FROM keyword
    WHERE keyword.id = user_keyword.keyword_id AND keyword.keyword = :keyword_1)
```

and `.contains()` is available for a proxy to a scalar collection:

```
>>> print(session.query(User).filter(User.keywords.contains('jek')))
SELECT user.*
FROM user
WHERE EXISTS (SELECT 1
FROM userkeywords, keyword
WHERE user.id = userkeywords.user_id
    AND keyword.id = userkeywords.keyword_id
    AND keyword.keyword = :keyword_1)
```

`AssociationProxy` can be used with `Query.join()` somewhat manually using the `attr` attribute in a star-args context:

```
q = session.query(User).join(*User.keywords.attr)
```

New in version 0.7.3: `attr` attribute in a star-args context.

`attr` is composed of `AssociationProxy.local_attr` and `AssociationProxy.remote_attr`, which are just synonyms for the actual proxied attributes, and can also be used for querying:

```
uka = aliased(UserKeyword)
ka = aliased(Keyword)
q = session.query(User).\
        join(uka, User.keywords.local_attr).\
        join(ka, User.keywords.remote_attr)
```

New in version 0.7.3: `AssociationProxy.local_attr` and `AssociationProxy.remote_attr`, synonyms for the actual proxied attributes, and usable for querying.

API Documentation

sqlalchemy.ext.associationproxy.association_proxy(*target_collection*, *attr*, ***kw*)

Return a Python property implementing a view of a target attribute which references an attribute on members of the target.

The returned value is an instance of `AssociationProxy`.

Implements a Python property representing a relationship as a collection of simpler values, or a scalar value. 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 attribute we'll proxy to. This attribute is typically mapped by `relationship()` to link to a target collection, but can also be a many-to-one or non-scalar relationship.
- **attr** – Attribute on the associated instance or 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*, *info=None*`)`

A descriptor that presents a read/write view of an object attribute.

`any(`*criterion=None*, ***kwargs*`)`

Produce a proxied 'any' expression using EXISTS.

This expression will be a composed product using the `RelationshipProperty.Comparator.any()` and/or `RelationshipProperty.Comparator.has()` operators of the underlying proxied attributes.

`attr`

Return a tuple of `(local_attr, remote_attr)`.

This attribute is convenient when specifying a join using `Query.join()` across two relationships:

```
sess.query(Parent).join(*Parent.proxied.attr)
```

New in version 0.7.3.

See also:

`AssociationProxy.local_attr`

`AssociationProxy.remote_attr`

`contains(`*obj*`)`

Produce a proxied 'contains' expression using EXISTS.

This expression will be a composed product using the RelationshipProperty.Comparator.any() , RelationshipProperty.Comparator.has(), and/or RelationshipProperty.Comparator.contains() operators of the underlying proxied attributes.

extension_type = symbol('ASSOCIATION_PROXY')

has(*criterion=None*, ***kwargs*)
Produce a proxied 'has' expression using EXISTS.

This expression will be a composed product using the RelationshipProperty.Comparator.any() and/or RelationshipProperty.Comparator.has() operators of the underlying proxied attributes.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this InspectionAttr.

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the column_property(), relationship(), or composite() functions.

New in version 0.8: Added support for .info to all MapperProperty subclasses.

Changed in version 1.0.0: MapperProperty.info is also available on extension types via the InspectionAttrInfo.info attribute, so that it can apply to a wider variety of ORM and extension constructs.

See also:

QueryableAttribute.info

SchemaItem.info

is_aliased_class = False

is_attribute = False

is_clause_element = False

is_instance = False

is_mapper = False

is_property = False

is_selectable = False

local_attr
The 'local' MapperProperty referenced by this AssociationProxy.

New in version 0.7.3.

See also:

AssociationProxy.attr

AssociationProxy.remote_attr

remote_attr
The 'remote' MapperProperty referenced by this AssociationProxy.

New in version 0.7.3.

See also:

AssociationProxy.attr

AssociationProxy.local_attr

scalar
Return True if this AssociationProxy proxies a scalar relationship on the local side.

target_class
The intermediary class handled by this AssociationProxy.

Intercepted append/set/assignment events will result in the generation of new instances of this class.

sqlalchemy.ext.associationproxy.ASSOCIATION_PROXY = symbol('ASSOCIATION_PROXY')

2.7.2 Automap

Define an extension to the sqlalchemy.ext.declarative system which automatically generates mapped classes and relationships from a database schema, typically though not necessarily one which is reflected.

New in version 0.9.1: Added sqlalchemy.ext.automap.

It is hoped that the AutomapBase system provides a quick and modernized solution to the problem that the very famous SQLSoup also tries to solve, that of generating a quick and rudimentary object model from an existing database on the fly. By addressing the issue strictly at the mapper configuration level, and integrating fully with existing Declarative class techniques, AutomapBase seeks to provide a well-integrated approach to the issue of expediently auto-generating ad-hoc mappings.

Basic Use

The simplest usage is to reflect an existing database into a new model. We create a new AutomapBase class in a similar manner as to how we create a declarative base class, using automap_base(). We then call AutomapBase.prepare() on the resulting base class, asking it to reflect the schema and produce mappings:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

Base = automap_base()

# engine, suppose it has two tables 'user' and 'address' set up
engine = create_engine("sqlite:///mydatabase.db")

# reflect the tables
Base.prepare(engine, reflect=True)

# mapped classes are now created with names by default
# matching that of the table name.
User = Base.classes.user
Address = Base.classes.address

session = Session(engine)

# rudimentary relationships are produced
session.add(Address(email_address="foo@bar.com", user=User(name="foo")))
session.commit()

# collection-based relationships are by default named
# "<classname>_collection"
print (u1.address_collection)
```

Above, calling AutomapBase.prepare() while passing along the AutomapBase.prepare.reflect parameter indicates that the MetaData.reflect() method will be called on this declarative base classes' MetaData collection; then, each **viable** Table within the MetaData will get a new mapped class generated automatically. The ForeignKeyConstraint objects which link the various tables together will be used to produce new, bidirectional relationship() objects between classes. The classes and relationships

follow along a default naming scheme that we can customize. At this point, our basic mapping consisting of related `User` and `Address` classes is ready to use in the traditional way.

Note: By **viable**, we mean that for a table to be mapped, it must specify a primary key. Additionally, if the table is detected as being a pure association table between two other tables, it will not be directly mapped and will instead be configured as a many-to-many table between the mappings for the two referring tables.

Generating Mappings from an Existing MetaData

We can pass a pre-declared `MetaData` object to `automap_base()`. This object can be constructed in any way, including programmatically, from a serialized file, or from itself being reflected using `MetaData.reflect()`. Below we illustrate a combination of reflection and explicit table declaration:

```
from sqlalchemy import create_engine, MetaData, Table, Column, ForeignKey
from sqlalchemy.ext.automap import automap_base
engine = create_engine("sqlite:///mydatabase.db")

# produce our own MetaData object
metadata = MetaData()

# we can reflect it ourselves from a database, using options
# such as 'only' to limit what tables we look at...
metadata.reflect(engine, only=['user', 'address'])

# ... or just define our own Table objects with it (or combine both)
Table('user_order', metadata,
                Column('id', Integer, primary_key=True),
                Column('user_id', ForeignKey('user.id'))
            )

# we can then produce a set of mappings from this MetaData.
Base = automap_base(metadata=metadata)

# calling prepare() just sets up mapped classes and relationships.
Base.prepare()

# mapped classes are ready
User, Address, Order = Base.classes.user, Base.classes.address,\
    Base.classes.user_order
```

Specifying Classes Explicitly

The `sqlalchemy.ext.automap` extension allows classes to be defined explicitly, in a way similar to that of the `DeferredReflection` class. Classes that extend from `AutomapBase` act like regular declarative classes, but are not immediately mapped after their construction, and are instead mapped when we call `AutomapBase.prepare()`. The `AutomapBase.prepare()` method will make use of the classes we've established based on the table name we use. If our schema contains tables `user` and `address`, we can define one or both of the classes to be used:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy import create_engine

# automap base
Base = automap_base()

# pre-declare User for the 'user' table
class User(Base):
```

```
    __tablename__ = 'user'

    # override schema elements like Columns
    user_name = Column('name', String)

    # override relationships too, if desired.
    # we must use the same name that automap would use for the
    # relationship, and also must refer to the class name that automap will
    # generate for "address"
    address_collection = relationship("address", collection_class=set)

# reflect
engine = create_engine("sqlite:///mydatabase.db")
Base.prepare(engine, reflect=True)

# we still have Address generated from the tablename "address",
# but User is the same as Base.classes.User now

Address = Base.classes.address

u1 = session.query(User).first()
print (u1.address_collection)

# the backref is still there:
a1 = session.query(Address).first()
print (a1.user)
```

Above, one of the more intricate details is that we illustrated overriding one of the `relationship()` objects that automap would have created. To do this, we needed to make sure the names match up with what automap would normally generate, in that the relationship name would be `User.address_collection` and the name of the class referred to, from automap's perspective, is called `address`, even though we are referring to it as `Address` within our usage of this class.

Overriding Naming Schemes

`sqlalchemy.ext.automap` is tasked with producing mapped classes and relationship names based on a schema, which means it has decision points in how these names are determined. These three decision points are provided using functions which can be passed to the `AutomapBase.prepare()` method, and are known as `classname_for_table()`, `name_for_scalar_relationship()`, and `name_for_collection_relationship()`. Any or all of these functions are provided as in the example below, where we use a "camel case" scheme for class names and a "pluralizer" for collection names using the Inflect package:

```
import re
import inflect

def camelize_classname(base, tablename, table):
    "Produce a 'camelized' class name, e.g. "
    "'words_and_underscores' -> 'WordsAndUnderscores'"

    return str(tablename[0].upper() + \
            re.sub(r'_([a-z])', lambda m: m.group(1).upper(), tablename[1:]))

_pluralizer = inflect.engine()
def pluralize_collection(base, local_cls, referred_cls, constraint):
    "Produce an 'uncamelized', 'pluralized' class name, e.g. "
    "'SomeTerm' -> 'some_terms'"

    referred_name = referred_cls.__name__
    uncamelized = re.sub(r'[A-Z]',
```

```
                        lambda m: "_%s" % m.group(0).lower(),
                        referred_name)[1:]
    pluralized = _pluralizer.plural(uncamelized)
    return pluralized

from sqlalchemy.ext.automap import automap_base

Base = automap_base()

engine = create_engine("sqlite:///mydatabase.db")

Base.prepare(engine, reflect=True,
             classname_for_table=camelize_classname,
             name_for_collection_relationship=pluralize_collection
    )
```

From the above mapping, we would now have classes `User` and `Address`, where the collection from `User` to `Address` is called `User.addresses`:

```
User, Address = Base.classes.User, Base.classes.Address

u1 = User(addresses=[Address(email="foo@bar.com")])
```

Relationship Detection

The vast majority of what automap accomplishes is the generation of `relationship()` structures based on foreign keys. The mechanism by which this works for many-to-one and one-to-many relationships is as follows:

1. A given `Table`, known to be mapped to a particular class, is examined for `ForeignKeyConstraint` objects.
2. From each `ForeignKeyConstraint`, the remote `Table` object present is matched up to the class to which it is to be mapped, if any, else it is skipped.
3. As the `ForeignKeyConstraint` we are examining corresponds to a reference from the immediate mapped class, the relationship will be set up as a many-to-one referring to the referred class; a corresponding one-to-many backref will be created on the referred class referring to this class.
4. If any of the columns that are part of the `ForeignKeyConstraint` are not nullable (e.g. `nullable=False`), a `cascade` keyword argument of `all, delete-orphan` will be added to the keyword arguments to be passed to the relationship or backref. If the `ForeignKeyConstraint` reports that `ForeignKeyConstraint.ondelete` is set to `CASCADE` for a not null or `SET NULL` for a nullable set of columns, the option `passive_deletes` flag is set to `True` in the set of relationship keyword arguments. Note that not all backends support reflection of ON DELETE.

   New in version 1.0.0: - automap will detect non-nullable foreign key constraints when producing a one-to-many relationship and establish a default cascade of `all, delete-orphan` if so; additionally, if the constraint specifies `ForeignKeyConstraint.ondelete` of `CASCADE` for non-nullable or `SET NULL` for nullable columns, the `passive_deletes=True` option is also added.
5. The names of the relationships are determined using the `AutomapBase.prepare.name_for_scalar_relationship` and `AutomapBase.prepare.name_for_collection_relationship` callable functions. It is important to note that the default relationship naming derives the name from the **the actual class name**. If you've given a particular class an explicit name by declaring it, or specified an alternate class naming scheme, that's the name from which the relationship name will be derived.
6. The classes are inspected for an existing mapped property matching these names. If one is detected on one side, but none on the other side, `AutomapBase` attempts to create a relationship on the missing side, then uses the `relationship.back_populates` parameter in order to point the new relationship to the other side.

7. In the usual case where no relationship is on either side, `AutomapBase.prepare()` produces a `relationship()` on the "many-to-one" side and matches it to the other using the `relationship.backref` parameter.
8. Production of the `relationship()` and optionally the `backref()` is handed off to the `AutomapBase.prepare.generate_relationship` function, which can be supplied by the end-user in order to augment the arguments passed to `relationship()` or `backref()` or to make use of custom implementations of these functions.

Custom Relationship Arguments

The `AutomapBase.prepare.generate_relationship` hook can be used to add parameters to relationships. For most cases, we can make use of the existing `automap.generate_relationship()` function to return the object, after augmenting the given keyword dictionary with our own arguments.

Below is an illustration of how to send `relationship.cascade` and `relationship.passive_deletes` options along to all one-to-many relationships:

```
from sqlalchemy.ext.automap import generate_relationship

def _gen_relationship(base, direction, return_fn,
                              attrname, local_cls, referred_cls, **kw):
    if direction is interfaces.ONETOMANY:
        kw['cascade'] = 'all, delete-orphan'
        kw['passive_deletes'] = True
    # make use of the built-in function to actually return
    # the result.
    return generate_relationship(base, direction, return_fn,
                                 attrname, local_cls, referred_cls, **kw)

from sqlalchemy.ext.automap import automap_base
from sqlalchemy import create_engine

# automap base
Base = automap_base()

engine = create_engine("sqlite:///mydatabase.db")
Base.prepare(engine, reflect=True,
             generate_relationship=_gen_relationship)
```

Many-to-Many relationships

`sqlalchemy.ext.automap` will generate many-to-many relationships, e.g. those which contain a `secondary` argument. The process for producing these is as follows:

1. A given `Table` is examined for `ForeignKeyConstraint` objects, before any mapped class has been assigned to it.
2. If the table contains two and exactly two `ForeignKeyConstraint` objects, and all columns within this table are members of these two `ForeignKeyConstraint` objects, the table is assumed to be a "secondary" table, and will **not be mapped directly**.
3. The two (or one, for self-referential) external tables to which the `Table` refers to are matched to the classes to which they will be mapped, if any.
4. If mapped classes for both sides are located, a many-to-many bi-directional `relationship()` / `backref()` pair is created between the two classes.
5. The override logic for many-to-many works the same as that of one-to-many/ many-to-one; the `generate_relationship()` function is called upon to generate the strucures and existing attributes will be maintained.

Relationships with Inheritance

`sqlalchemy.ext.automap` will not generate any relationships between two classes that are in an inheritance relationship. That is, with two classes given as follows:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    type = Column(String(50))
    __mapper_args__ = {
         'polymorphic_identity':'employee', 'polymorphic_on': type
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    __mapper_args__ = {
        'polymorphic_identity':'engineer',
    }
```

The foreign key from `Engineer` to `Employee` is used not for a relationship, but to establish joined inheritance between the two classes.

Note that this means automap will not generate *any* relationships for foreign keys that link from a subclass to a superclass. If a mapping has actual relationships from subclass to superclass as well, those need to be explicit. Below, as we have two separate foreign keys from `Engineer` to `Employee`, we need to set up both the relationship we want as well as the `inherit_condition`, as these are not things SQLAlchemy can guess:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    type = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'employee', 'polymorphic_on':type
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    favorite_employee_id = Column(Integer, ForeignKey('employee.id'))

    favorite_employee = relationship(Employee,
                                     foreign_keys=favorite_employee_id)

    __mapper_args__ = {
        'polymorphic_identity':'engineer',
        'inherit_condition': id == Employee.id
    }
```

Handling Simple Naming Conflicts

In the case of naming conflicts during mapping, override any of `classname_for_table()`, `name_for_scalar_relationship()`, and `name_for_collection_relationship()` as needed. For example, if automap is attempting to name a many-to-one relationship the same as an existing column, an alternate convention can be conditionally selected. Given a schema:

```
CREATE TABLE table_a (
    id INTEGER PRIMARY KEY
```

```
);

CREATE TABLE table_b (
    id INTEGER PRIMARY KEY,
    table_a INTEGER,
    FOREIGN KEY(table_a) REFERENCES table_a(id)
);
```

The above schema will first automap the `table_a` table as a class named `table_a`; it will then automap a relationship onto the class for `table_b` with the same name as this related class, e.g. `table_a`. This relationship name conflicts with the mapping column `table_b.table_a`, and will emit an error on mapping.

We can resolve this conflict by using an underscore as follows:

```
def name_for_scalar_relationship(base, local_cls, referred_cls, constraint):
    name = referred_cls.__name__.lower()
    local_table = local_cls.__table__
    if name in local_table.columns:
        newname = name + "_"
        warnings.warn(
            "Already detected name %s present.  using %s" %
            (name, newname))
        return newname
    return name


Base.prepare(engine, reflect=True,
    name_for_scalar_relationship=name_for_scalar_relationship)
```

Alternatively, we can change the name on the column side. The columns that are mapped can be modified using the technique described at mapper_column_distinct_names, by assigning the column explicitly to a new name:

```
Base = automap_base()

class TableB(Base):
    __tablename__ = 'table_b'
    _table_a = Column('table_a', ForeignKey('table_a.id'))

Base.prepare(engine, reflect=True)
```

Using Automap with Explicit Declarations

As noted previously, automap has no dependency on reflection, and can make use of any collection of `Table` objects within a `MetaData` collection. From this, it follows that automap can also be used generate missing relationships given an otherwise complete model that fully defines table metadata:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy import Column, Integer, String, ForeignKey

Base = automap_base()

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String)

class Address(Base):
    __tablename__ = 'address'
```

```
    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(ForeignKey('user.id'))

# produce relationships
Base.prepare()

# mapping is complete, with "address_collection" and
# "user" relationships
a1 = Address(email='u1')
a2 = Address(email='u2')
u1 = User(address_collection=[a1, a2])
assert a1.user is u1
```

Above, given mostly complete `User` and `Address` mappings, the `ForeignKey` which we defined on `Address.user_id` allowed a bidirectional relationship pair `Address.user` and `User.address_collection` to be generated on the mapped classes.

Note that when subclassing `AutomapBase`, the `AutomapBase.prepare()` method is required; if not called, the classes we've declared are in an un-mapped state.

API Reference

`sqlalchemy.ext.automap.automap_base`(*declarative_base=None*, ***kw*)

Produce a declarative automap base.

This function produces a new base class that is a product of the `AutomapBase` class as well a declarative base produced by `declarative.declarative_base()`.

All parameters other than `declarative_base` are keyword arguments that are passed directly to the `declarative.declarative_base()` function.

Parameters

- **`declarative_base`** – an existing class produced by `declarative.declarative_base()`. When this is passed, the function no longer invokes `declarative.declarative_base()` itself, and all other keyword arguments are ignored.
- **`**kw`** – keyword arguments are passed along to `declarative.declarative_base()`.

`class sqlalchemy.ext.automap.AutomapBase`

Base class for an "automap" schema.

The `AutomapBase` class can be compared to the "declarative base" class that is produced by the `declarative.declarative_base()` function. In practice, the `AutomapBase` class is always used as a mixin along with an actual declarative base.

A new subclassable `AutomapBase` is typically instantated using the `automap_base()` function.

See also:

automap_toplevel

`classes = None`

An instance of `util.Properties` containing classes.

This object behaves much like the `.c` collection on a table. Classes are present under the name they were given, e.g.:

```
Base = automap_base()
Base.prepare(engine=some_engine, reflect=True)
```

```
User, Address = Base.classes.User, Base.classes.Address
```

classmethod prepare(*engine=None*, *reflect=False*, *schema=None*, *classname_for_table=<function classname_for_table>*, *collection_class=<class 'list'>*, *name_for_scalar_relationship=<function name_for_scalar_relationship>*, *name_for_collection_relationship=<function name_for_collection_relationship>*, *generate_relationship=<function generate_relationship>*)

Extract mapped classes and relationships from the `MetaData` and perform mappings.

Parameters

- **engine** – an `Engine` or `Connection` with which to perform schema reflection, if specified. If the `AutomapBase.prepare.reflect` argument is False, this object is not used.
- **reflect** – if True, the `MetaData.reflect()` method is called on the `MetaData` associated with this `AutomapBase`. The `Engine` passed via `AutomapBase.prepare.engine` will be used to perform the reflection if present; else, the `MetaData` should already be bound to some engine else the operation will fail.
- **classname_for_table** – callable function which will be used to produce new class names, given a table name. Defaults to `classname_for_table()`.
- **name_for_scalar_relationship** – callable function which will be used to produce relationship names for scalar relationships. Defaults to `name_for_scalar_relationship()`.
- **name_for_collection_relationship** – callable function which will be used to produce relationship names for collection-oriented relationships. Defaults to `name_for_collection_relationship()`.
- **generate_relationship** – callable function which will be used to actually generate `relationship()` and `backref()` constructs. Defaults to `generate_relationship()`.
- **collection_class** – the Python collection class that will be used when a new `relationship()` object is created that represents a collection. Defaults to `list`.
- **schema** – When present in conjunction with the `AutomapBase.prepare.reflect` flag, is passed to `MetaData.reflect()` to indicate the primary schema where tables should be reflected from. When omitted, the default schema in use by the database connection is used.

  New in version 1.1.

sqlalchemy.ext.automap.**classname_for_table**(*base*, *tablename*, *table*)

Return the class name that should be used, given the name of a table.

The default implementation is:

```
return str(tablename)
```

Alternate implementations can be specified using the `AutomapBase.prepare.classname_for_table` parameter.

Parameters

- **base** – the `AutomapBase` class doing the prepare.
- **tablename** – string name of the `Table`.
- **table** – the `Table` object itself.

Returns

a string class name.

Note: In Python 2, the string used for the class name **must** be a non-Unicode object, e.g. a `str()` object. The `.name` attribute of `Table` is typically a Python unicode subclass, so the `str()` function should be applied to this name, after accounting for any non-ASCII characters.

`sqlalchemy.ext.automap.name_for_scalar_relationship(base, local_cls, referred_cls, constraint)`

Return the attribute name that should be used to refer from one class to another, for a scalar object reference.

The default implementation is:

```
return referred_cls.__name__.lower()
```

Alternate implementations can be specified using the `AutomapBase.prepare.name_for_scalar_relationship` parameter.

Parameters

- **base** – the `AutomapBase` class doing the prepare.
- **local_cls** – the class to be mapped on the local side.
- **referred_cls** – the class to be mapped on the referring side.
- **constraint** – the `ForeignKeyConstraint` that is being inspected to produce this relationship.

`sqlalchemy.ext.automap.name_for_collection_relationship(base, local_cls, referred_cls, constraint)`

Return the attribute name that should be used to refer from one class to another, for a collection reference.

The default implementation is:

```
return referred_cls.__name__.lower() + "_collection"
```

Alternate implementations can be specified using the `AutomapBase.prepare.name_for_collection_relationship` parameter.

Parameters

- **base** – the `AutomapBase` class doing the prepare.
- **local_cls** – the class to be mapped on the local side.
- **referred_cls** – the class to be mapped on the referring side.
- **constraint** – the `ForeignKeyConstraint` that is being inspected to produce this relationship.

`sqlalchemy.ext.automap.generate_relationship(base, direction, return_fn, attrname, local_cls, referred_cls, **kw)`

Generate a `relationship()` or `backref()` on behalf of two mapped classes.

An alternate implementation of this function can be specified using the `AutomapBase.prepare.generate_relationship` parameter.

The default implementation of this function is as follows:

```
if return_fn is backref:
    return return_fn(attrname, **kw)
elif return_fn is relationship:
```

```
    return return_fn(referred_cls, **kw)
else:
    raise TypeError("Unknown relationship function: %s" % return_fn)
```

Parameters

- **base** – the `AutomapBase` class doing the prepare.
- **direction** – indicate the "direction" of the relationship; this will be one of `ONETOMANY`, `MANYTOONE`, `MANYTOMANY`.
- **return_fn** – the function that is used by default to create the relationship. This will be either `relationship()` or `backref()`. The `backref()` function's result will be used to produce a new `relationship()` in a second step, so it is critical that user-defined implementations correctly differentiate between the two functions, if a custom relationship function is being used.
- **attrname** – the attribute name to which this relationship is being assigned. If the value of `generate_relationship.return_fn` is the `backref()` function, then this name is the name that is being assigned to the backref.
- **local_cls** – the "local" class to which this relationship or backref will be locally present.
- **referred_cls** – the "referred" class to which the relationship or backref refers to.
- ****kw** – all additional keyword arguments are passed along to the function.

Returns a `relationship()` or `backref()` construct, as dictated by the `generate_relationship.return_fn` parameter.

2.7.3 Baked Queries

`baked` provides an alternative creational pattern for `Query` objects, which allows for caching of the object's construction and string-compilation steps. This means that for a particular `Query` building scenario that is used more than once, all of the Python function invocation involved in building the query from its initial construction up through generating a SQL string will only occur **once**, rather than for each time that query is built up and executed.

The rationale for this system is to greatly reduce Python interpreter overhead for everything that occurs **before the SQL is emitted**. The caching of the "baked" system does **not** in any way reduce SQL calls or cache the **return results** from the database. A technique that demonstrates the caching of the SQL calls and result sets themselves is available in examples_caching.

New in version 1.0.0.

Note: The `sqlalchemy.ext.baked` extension is **not for beginners**. Using it correctly requires a good high level understanding of how SQLAlchemy, the database driver, and the backend database interact with each other. This extension presents a very specific kind of optimization that is not ordinarily needed. As noted above, it **does not cache queries**, only the string formulation of the SQL itself.

Synopsis

Usage of the baked system starts by producing a so-called "bakery", which represents storage for a particular series of query objects:

```
from sqlalchemy.ext import baked

bakery = baked.bakery()
```

The above "bakery" will store cached data in an LRU cache that defaults to 200 elements, noting that an ORM query will typically contain one entry for the ORM query as invoked, as well as one entry per database dialect for the SQL string.

The bakery allows us to build up a `Query` object by specifying its construction as a series of Python callables, which are typically lambdas. For succinct usage, it overrides the `+=` operator so that a typical query build-up looks like the following:

```
from sqlalchemy import bindparam

def search_for_user(session, username, email=None):

    baked_query = bakery(lambda session: session.query(User))
    baked_query += lambda q: q.filter(User.name == bindparam('username'))

    baked_query += lambda q: q.order_by(User.id)

    if email:
        baked_query += lambda q: q.filter(User.email == bindparam('email'))

    result = baked_query(session).params(username=username, email=email).all()

    return result
```

Following are some observations about the above code:

1. The `baked_query` object is an instance of `BakedQuery`. This object is essentially the "builder" for a real orm `Query` object, but it is not itself the *actual* `Query` object.
2. The actual `Query` object is not built at all, until the very end of the function when `Result.all()` is called.
3. The steps that are added to the `baked_query` object are all expressed as Python functions, typically lambdas. The first lambda given to the `bakery()` function receives a `Session` as its argument. The remaining lambdas each receive a `Query` as their argument.
4. In the above code, even though our application may call upon `search_for_user()` many times, and even though within each invocation we build up an entirely new `BakedQuery` object, *all of the lambdas are only called once*. Each lambda is **never** called a second time for as long as this query is cached in the bakery.
5. The caching is achieved by storing references to the **lambda objects themselves** in order to formulate a cache key; that is, the fact that the Python interpreter assigns an in-Python identity to these functions is what determines how to identify the query on successive runs. For those invocations of `search_for_user()` where the `email` parameter is specified, the callable `lambda q: q.filter(User.email == bindparam('email'))` will be part of the cache key that's retrieved; when `email` is `None`, this callable is not part of the cache key.
6. Because the lambdas are all called only once, it is essential that no variables which may change across calls are referenced **within** the lambdas; instead, assuming these are values to be bound into the SQL string, we use `bindparam()` to construct named parameters, where we apply their actual values later using `Result.params()`.

Performance

The baked query probably looks a little odd, a little bit awkward and a little bit verbose. However, the savings in Python performance for a query which is invoked lots of times in an application are

very dramatic. The example suite `short_selects` demonstrated in examples_performance illustrates a comparison of queries which each return only one row, such as the following regular query:

```
session = Session(bind=engine)
for id_ in random.sample(ids, n):
    session.query(Customer).filter(Customer.id == id_).one()
```

compared to the equivalent "baked" query:

```
bakery = baked.bakery()
s = Session(bind=engine)
for id_ in random.sample(ids, n):
    q = bakery(lambda s: s.query(Customer))
    q += lambda q: q.filter(Customer.id == bindparam('id'))
    q(s).params(id=id_).one()
```

The difference in Python function call count for an iteration of 10000 calls to each block are:

```
test_baked_query : test a baked query of the full entity.
                   (10000 iterations); total fn calls 1951294

test_orm_query :   test a straight ORM query of the full entity.
                   (10000 iterations); total fn calls 7900535
```

In terms of number of seconds on a powerful laptop, this comes out as:

```
test_baked_query : test a baked query of the full entity.
                   (10000 iterations); total time 2.174126 sec

test_orm_query :   test a straight ORM query of the full entity.
                   (10000 iterations); total time 7.958516 sec
```

Note that this test very intentionally features queries that only return one row. For queries that return many rows, the performance advantage of the baked query will have less and less of an impact, proportional to the time spent fetching rows. It is critical to keep in mind that the **baked query feature only applies to building the query itself, not the fetching of results**. Using the baked feature is by no means a guarantee to a much faster application; it is only a potentially useful feature for those applications that have been measured as being impacted by this particular form of overhead.

Measure twice, cut once

For background on how to profile a SQLAlchemy application, please see the section faq_performance. It is essential that performance measurement techniques are used when attempting to improve the performance of an application.

Rationale

The "lambda" approach above is a superset of what would be a more traditional "parameterized" approach. Suppose we wished to build a simple system where we build a `Query` just once, then store it in a dictionary for re-use. This is possible right now by just building up the query, and removing its `Session` by calling `my_cached_query = query.with_session(None)`:

```
my_simple_cache = {}

def lookup(session, id_argument):
    if "my_key" not in my_simple_cache:
        query = session.query(Model).filter(Model.id == bindparam('id'))
        my_simple_cache["my_key"] = query.with_session(None)
    else:
```

```
        query = my_simple_cache["my_key"].with_session(session)

    return query.params(id=id_argument).all()
```

The above approach gets us a very minimal performance benefit. By re-using a `Query`, we save on the Python work within the `session.query(Model)` constructor as well as calling upon `filter(Model.id == bindparam('id'))`, which will skip for us the building up of the Core expression as well as sending it to `Query.filter()`. However, the approach still regenerates the full `Select` object every time when `Query.all()` is called and additionally this brand new `Select` is sent off to the string compilation step every time, which for a simple case like the above is probably about 70% of the overhead.

To reduce the additional overhead, we need some more specialized logic, some way to memoize the construction of the select object and the construction of the SQL. There is an example of this on the wiki in the section BakedQuery, a precursor to this feature, however in that system, we aren't caching the *construction* of the query. In order to remove all the overhead, we need to cache both the construction of the query as well as the SQL compilation. Let's assume we adapted the recipe in this way and made ourselves a method `.bake()` that pre-compiles the SQL for the query, producing a new object that can be invoked with minimal overhead. Our example becomes:

```
my_simple_cache = {}

def lookup(session, id_argument):

    if "my_key" not in my_simple_cache:
        query = session.query(Model).filter(Model.id == bindparam('id'))
        my_simple_cache["my_key"] = query.with_session(None).bake()
    else:
        query = my_simple_cache["my_key"].with_session(session)

    return query.params(id=id_argument).all()
```

Above, we've fixed the performance situation, but we still have this string cache key to deal with.

We can use the "bakery" approach to re-frame the above in a way that looks less unusual than the "building up lambdas" approach, and more like a simple improvement upon the simple "reuse a query" approach:

```
bakery = baked.bakery()

def lookup(session, id_argument):
    def create_model_query(session):
        return session.query(Model).filter(Model.id == bindparam('id'))

    parameterized_query = bakery.bake(create_model_query)
    return parameterized_query(session).params(id=id_argument).all()
```

Above, we use the "baked" system in a manner that is very similar to the simplistic "cache a query" system. However, it uses two fewer lines of code, does not need to manufacture a cache key of "my_key", and also includes the same feature as our custom "bake" function that caches 100% of the Python invocation work from the constructor of the query, to the filter call, to the production of the `Select` object, to the string compilation step.

From the above, if we ask ourselves, "what if lookup needs to make conditional decisions as to the structure of the query?", this is where hopefully it becomes apparent why "baked" is the way it is. Instead of a parameterized query building off from exactly one function (which is how we thought baked might work originally), we can build it from *any number* of functions. Consider our naive example, if we needed to have an additional clause in our query on a conditional basis:

```
my_simple_cache = {}

def lookup(session, id_argument, include_frobnizzle=False):
    if include_frobnizzle:
```

```
        cache_key = "my_key_with_frobnizzle"
    else:
        cache_key = "my_key_without_frobnizzle"

    if cache_key not in my_simple_cache:
        query = session.query(Model).filter(Model.id == bindparam('id'))
        if include_frobnizzle:
            query = query.filter(Model.frobnizzle == True)

        my_simple_cache[cache_key] = query.with_session(None).bake()
    else:
        query = my_simple_cache[cache_key].with_session(session)

    return query.params(id=id_argument).all()
```

Our "simple" parameterized system must now be tasked with generating cache keys which take into account whether or not the "include_frobnizzle" flag was passed, as the presence of this flag means that the generated SQL would be entirely different. It should be apparent that as the complexity of query building goes up, the task of caching these queries becomes burdensome very quickly. We can convert the above example into a direct use of "bakery" as follows:

```
bakery = baked.bakery()

def lookup(session, id_argument, include_frobnizzle=False):
    def create_model_query(session):
        return session.query(Model).filter(Model.id == bindparam('id'))

    parameterized_query = bakery.bake(create_model_query)

    if include_frobnizzle:
        def include_frobnizzle_in_query(query):
            return query.filter(Model.frobnizzle == True)

        parameterized_query = parameterized_query.with_criteria(
            include_frobnizzle_in_query)

    return parameterized_query(session).params(id=id_argument).all()
```

Above, we again cache not just the query object but all the work it needs to do in order to generate SQL. We also no longer need to deal with making sure we generate a cache key that accurately takes into account all of the structural modifications we've made; this is now handled automatically and without the chance of mistakes.

This code sample is a few lines shorter than the naive example, removes the need to deal with cache keys, and has the vast performance benefits of the full so-called "baked" feature. But still a little verbose! Hence we take methods like `BakedQuery.add_criteria()` and `BakedQuery.with_criteria()` and shorten them into operators, and encourage (though certainly not require!) using simple lambdas, only as a means to reduce verbosity:

```
bakery = baked.bakery()

def lookup(session, id_argument, include_frobnizzle=False):
    parameterized_query = bakery.bake(
        lambda s: s.query(Model).filter(Model.id == bindparam('id'))
      )

    if include_frobnizzle:
        parameterized_query += lambda q: q.filter(Model.frobnizzle == True)

    return parameterized_query(session).params(id=id_argument).all()
```

Where above, the approach is simpler to implement and much more similar in code flow to what a

non-cached querying function would look like, hence making code easier to port.

The above description is essentially a summary of the design process used to arrive at the current "baked" approach. Starting from the "normal" approaches, the additional issues of cache key construction and management, removal of all redundant Python execution, and queries built up with conditionals needed to be addressed, leading to the final approach.

Disabling Baked Queries Session-wide

The flag `Session.enable_baked_queries` may be set to False, causing all baked queries to not use the cache when used against that `Session`:

```
session = Session(engine, enable_baked_queries=False)
```

Like all session flags, it is also accepted by factory objects like `sessionmaker` and methods like `sessionmaker.configure()`.

The immediate rationale for this flag is to reduce memory use in the case that the query baking used by relationship loaders and other loaders is not desirable. It also can be used in the case that an application which is seeing issues potentially due to cache key conflicts from user-defined baked queries or other baked query issues can turn the behavior off, in order to identify or eliminate baked queries as the cause of an issue.

New in version 1.2.

Lazy Loading Integration

The baked query system is integrated into SQLAlchemy's lazy loader feature as used by `relationship()`, and will cache queries for most lazy load conditions. A small subset of "lazy loads" may not be cached; these involve query options in conjunction with ad-hoc `aliased` structures that cannot produce a repeatable cache key.

Changed in version 1.2: "baked" queries are now the foundation of the lazy-loader feature of `relationship()`.

Opting out with the bake_queries flag

The `relationship()` construct includes a flag `relationship.bake_queries` which when set to False will cause that relationship to opt out of caching queries. Additionally, the `Session.enable_baked_queries` setting can be used to disable all "baked query" use. These flags can be useful to conserve memory, when memory conservation is more important than performance for a particular relationship or for the application overall.

API Documentation

`sqlalchemy.ext.baked.bakery`(*size=200*, *_size_alert=None*)
Construct a new bakery.

Returns an instance of `Bakery`

`class sqlalchemy.ext.baked.BakedQuery`(*bakery*, *initial_fn*, *args=()*)
A builder object for `query.Query` objects.

`add_criteria`(*fn*, **args*)
Add a criteria function to this `BakedQuery`.

This is equivalent to using the `+=` operator to modify a `BakedQuery` in-place.

`classmethod bakery`(*size=200*, *_size_alert=None*)
Construct a new bakery.

Returns an instance of `Bakery`

`for_session`(*session*)
Return a `Result` object for this `BakedQuery`.

This is equivalent to calling the `BakedQuery` as a Python callable, e.g. `result = my_baked_query(session)`.

`spoil`(*full=False*)
Cancel any query caching that will occur on this BakedQuery object.

The BakedQuery can continue to be used normally, however additional creational functions will not be cached; they will be called on every invocation.

This is to support the case where a particular step in constructing a baked query disqualifies the query from being cacheable, such as a variant that relies upon some uncacheable value.

Parameters `full` – if False, only functions added to this `BakedQuery` object subsequent to the spoil step will be non-cached; the state of the `BakedQuery` up until this point will be pulled from the cache. If True, then the entire `Query` object is built from scratch each time, with all creational functions being called on each invocation.

`with_criteria`(*fn*, **args*)
Add a criteria function to a `BakedQuery` cloned from this one.

This is equivalent to using the `+` operator to produce a new `BakedQuery` with modifications.

`class sqlalchemy.ext.baked.Bakery`(*cls_*, *cache*)
Callable which returns a `BakedQuery`.

This object is returned by the class method `BakedQuery.bakery()`. It exists as an object so that the "cache" can be easily inspected.

New in version 1.2.

`class sqlalchemy.ext.baked.Result`(*bq*, *session*)
Invokes a `BakedQuery` against a `Session`.

The `Result` object is where the actual `query.Query` object gets created, or retrieved from the cache, against a target `Session`, and is then invoked for results.

`all`()
Return all rows.

Equivalent to `Query.all()`.

`count`()
return the 'count'.

Equivalent to `Query.count()`.

Note this uses a subquery to ensure an accurate count regardless of the structure of the original statement.

New in version 1.1.6.

`first`()
Return the first row.

Equivalent to `Query.first()`.

`get`(*ident*)
Retrieve an object based on identity.

Equivalent to `Query.get()`.

`one`()
Return exactly one result or raise an exception.

Equivalent to `Query.one()`.

`one_or_none()`
Return one or zero results, or raise an exception for multiple rows.

Equivalent to `Query.one_or_none()`.

New in version 1.0.9.

`params(*args, **kw)`
Specify parameters to be replaced into the string SQL statement.

`scalar()`
Return the first element of the first result or None if no rows present. If multiple rows are returned, raises MultipleResultsFound.

Equivalent to `Query.scalar()`.

New in version 1.1.6.

`with_post_criteria(fn)`
Add a criteria function that will be applied post-cache.

This adds a function that will be run against the `Query` object after it is retrieved from the cache. Functions here can be used to alter the query in ways that **do not affect the SQL output**, such as execution options and shard identifiers (when using a shard-enabled query object)

> **Warning:** `Result.with_post_criteria()` functions are applied to the `Query` object **after** the query's SQL statement object has been retrieved from the cache. Any operations here which intend to modify the SQL should ensure that `BakedQuery.spoil()` was called first.

New in version 1.2.

2.7.4 Declarative

The Declarative system is the typically used system provided by the SQLAlchemy ORM in order to define classes mapped to relational database tables. However, as noted in classical_mapping, Declarative is in fact a series of extensions that ride on top of the SQLAlchemy `mapper()` construct.

While the documentation typically refers to Declarative for most examples, the following sections will provide detailed information on how the Declarative API interacts with the basic `mapper()` and Core `Table` systems, as well as how sophisticated patterns can be built using systems such as mixins.

Basic Use

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 import Column, Integer, String
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)
```

`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)
```

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')
```

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()
                }
```

Defining SQL Expressions

See mapper_sql_expressions for examples on declaratively mapping attributes to SQL expressions.

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")
```

For the case where more than one module contains a class of the same name, string class names can also be specified as module-qualified paths within any of these string expressions:

```
class User(Base):
    # ....
    addresses = relationship("myapp.model.address.Address",
                          order_by="desc(myapp.model.address.Address.email)",
                          primaryjoin="myapp.model.address.Address.user_id=="
                                      "myapp.model.user.User.id")
```

The qualified path can be any partial path that removes ambiguity between the names. For example, to disambiguate between `myapp.model.address.Address` and `myapp.model.lookup.Address`, we can specify `address.Address` or `lookup.Address`:

```
class User(Base):
    # ....
    addresses = relationship("address.Address",
                          order_by="desc(address.Address.email)",
                          primaryjoin="address.Address.user_id=="
                                      "User.id")
```

New in version 0.8: module-qualified paths can be used when specifying string arguments with Declarative, in order to specify specific modules.

Two alternatives also exist to using string-based attributes. A lambda can also be used, which will be evaluated after all mappers have been configured:

```
class User(Base):
    # ...
    addresses = relationship(lambda: Address,
                          order_by=lambda: desc(Address.email),
                          primaryjoin=lambda: Address.user_id==User.id)
```

Or, the relationship can be added to the class explicitly after the classes are available:

```
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)
```

Like other `relationship()` arguments, a string is accepted as well, passing the string name of the table as defined in the `Base.metadata.tables` collection:

```
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.

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, where each argument is positional (usually constraints):

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = (
            ForeignKeyConstraint(['id'], ['remote_table.id']),
            UniqueConstraint('foo'),
            )
```

Keyword arguments can be specified with the above form by specifying the last argument as a dictionary:

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = (
            ForeignKeyConstraint(['id'], ['remote_table.id']),
            UniqueConstraint('foo'),
            {'autoload':True}
            )
```

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 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.

Note that when the __table__ approach is used, the object is immediately usable as a plain Table within the class declaration body itself, as a Python class is only another syntactical block. Below this is illustrated by using the id column in the primaryjoin condition of a relationship():

```
class MyClass(Base):
    __table__ = Table('my_table', Base.metadata,
        Column('id', Integer, primary_key=True),
        Column('name', String(50))
    )

    widgets = relationship(Widget,
                primaryjoin=Widget.myclass_id==__table__.c.id)
```

Similarly, mapped attributes which refer to __table__ can be placed inline, as below where we assign the name column to the attribute _name, generating a synonym for name:

```
from sqlalchemy.ext.declarative import synonym_for

class MyClass(Base):
    __table__ = Table('my_table', Base.metadata,
        Column('id', Integer, primary_key=True),
        Column('name', String(50))
    )

    _name = __table__.c.name

    @synonym_for("_name")
    def name(self):
        return "Name: %s" % _name
```

Using Reflection with Declarative

It's easy to set up a Table that uses autoload=True in conjunction with a mapped class:

```
class MyClass(Base):
    __table__ = Table('mytable', Base.metadata,
                      autoload=True, autoload_with=some_engine)
```

However, one improvement that can be made here is to not require the Engine to be available when

classes are being first declared. To achieve this, use the `DeferredReflection` mixin, which sets up mappings only after a special `prepare(engine)` step is called:

```
from sqlalchemy.ext.declarative import declarative_base, DeferredReflection

Base = declarative_base(cls=DeferredReflection)

class Foo(Base):
    __tablename__ = 'foo'
    bars = relationship("Bar")

class Bar(Base):
    __tablename__ = 'bar'

    # illustrate overriding of "bar.foo_id" to have
    # a foreign key constraint otherwise not
    # reflected, such as when using MySQL
    foo_id = Column(Integer, ForeignKey('foo.id'))

Base.prepare(e)
```

New in version 0.8: Added `DeferredReflection`.

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__`.

See also:

inheritance_toplevel - general introduction to inheritance mapping with Declarative.

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, with the "Engineer.id" column taking precedence if queried directly. 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__`.

Resolving Column Conflicts

Note above that the `primary_language` and `golf_swing` columns are "moved up" to be applied to `Person.__table__`, as a result of their declaration on a subclass that has no table of its own. A tricky case comes up when two subclasses want to specify *the same* column, as below:

```
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'}
    start_date = Column(DateTime)

class Manager(Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}
    start_date = Column(DateTime)
```

Above, the `start_date` column declared on both `Engineer` and `Manager` will result in an error:

```
sqlalchemy.exc.ArgumentError: Column 'start_date' on class
<class '__main__.Manager'> conflicts with existing
column 'people.start_date'
```

In a situation like this, Declarative can't be sure of the intent, especially if the `start_date` columns had, for example, different types. A situation like this can be resolved by using `declared_attr` to define the

Column conditionally, taking care to return the **existing column** via the parent __table__ if it already exists:

```
from sqlalchemy.ext.declarative import declared_attr

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'}

    @declared_attr
    def start_date(cls):
        "Start date column, if not present already."
        return Person.__table__.c.get('start_date', Column(DateTime))

class Manager(Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}

    @declared_attr
    def start_date(cls):
        "Start date column, if not present already."
        return Person.__table__.c.get('start_date', Column(DateTime))
```

Above, when Manager is mapped, the start_date column is already present on the Person class. Declarative lets us return that Column as a result in this case, where it knows to skip re-assigning the same column. If the mapping is mis-configured such that the start_date column is accidentally re-assigned to a different table (such as, if we changed Manager to be joined inheritance without fixing start_date), an error is raised which indicates an existing Column is trying to be re-assigned to a different owning Table.

New in version 0.8: declared_attr can be used on a non-mixin class, and the returned Column or other mapped attribute will be applied to the mapping as any other attribute. Previously, the resulting attribute would be ignored, and also result in a warning being emitted when a subclass was created.

New in version 0.8: declared_attr, when used either with a mixin or non-mixin declarative class, can return an existing Column already assigned to the parent Table, to indicate that the re-assignment of the Column should be skipped, however should still be mapped on the target class, in order to resolve duplicate column conflicts.

The same concept can be used with mixin classes (see declarative_mixins):

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class HasStartDate(object):
    @declared_attr
    def start_date(cls):
        return cls.__table__.c.get('start_date', Column(DateTime))

class Engineer(HasStartDate, Person):
    __mapper_args__ = {'polymorphic_identity': 'engineer'}

class Manager(HasStartDate, Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}
```

The above mixin checks the local __table__ attribute for the column. Because we're using single table

inheritance, we're sure that in this case, `cls.__table__` refers to `Person.__table__`. If we were mixing joined- and single-table inheritance, we might want our mixin to check more carefully if `cls.__table__` is really the `Table` we're looking for.

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()`, which needs to be created with the `Table` objects before the class is built:

```
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}
```

The helper classes `AbstractConcreteBase` and `ConcreteBase` provide automation for the above system of creating a polymorphic union. See the documentation for these helpers as well as the main ORM documentation on concrete inheritance for details.

See also:

concrete_inheritance

inheritance_concrete_helpers

Mixin and Custom Base Classes

A common need when using `declarative` is to share some functionality, such as a set of common columns, some common table options, or other mapped properties, across many classes. The standard Python idioms for this is to have the classes inherit from a base which includes these common features.

When using `declarative`, this idiom is allowed via the usage of a custom declarative base class, as well as a "mixin" class which is inherited from in addition to the primary base. Declarative includes several helper features to make this work in terms of how mappings are declared. An example of some commonly mixed-in idioms is below:

```
from sqlalchemy.ext.declarative import declared_attr

class MyMixin(object):

    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

    __table_args__ = {'mysql_engine': 'InnoDB'}
    __mapper_args__= {'always_refresh': True}

    id =  Column(Integer, primary_key=True)

class MyModel(MyMixin, Base):
    name = Column(String(1000))
```

Where above, the class `MyModel` will contain an "id" column as the primary key, a `__tablename__` attribute that derives from the name of the class itself, as well as `__table_args__` and `__mapper_args__` defined by the `MyMixin` mixin class.

There's no fixed convention over whether `MyMixin` precedes `Base` or not. Normal Python method resolution rules apply, and the above example would work just as well with:

```
class MyModel(Base, MyMixin):
    name = Column(String(1000))
```

This works because `Base` here doesn't define any of the variables that `MyMixin` defines, i.e. `__tablename__`, `__table_args__`, `id`, etc. If the `Base` did define an attribute of the same name, the class placed first in the inherits list would determine which attribute is used on the newly defined class.

Augmenting the Base

In addition to using a pure mixin, most of the techniques in this section can also be applied to the base class itself, for patterns that should apply to all classes derived from a particular base. This is achieved using the `cls` argument of the `declarative_base()` function:

```
from sqlalchemy.ext.declarative import declared_attr

class Base(object):
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

    __table_args__ = {'mysql_engine': 'InnoDB'}

    id =  Column(Integer, primary_key=True)

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base(cls=Base)
```

```
class MyModel(Base):
    name = Column(String(1000))
```

Where above, `MyModel` and all other classes that derive from `Base` will have a table name derived from the class name, an `id` primary key column, as well as the "InnoDB" engine for MySQL.

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(TimestampMixin, Base):
    __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 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(ReferenceAddressMixin, Base):
    __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.

Changed in version 0.6.5: Rename `sqlalchemy.util.classproperty` into `declared_attr`.

Columns generated by `declared_attr` can also be referenced by `__mapper_args__` to a limited degree, currently by `polymorphic_on` and `version_id_col`; 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(MyMixin, Base):
    __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(RefTargetMixin, Base):
    __tablename__ = 'foo'
    id = Column(Integer, primary_key=True)

class Bar(RefTargetMixin, Base):
    __tablename__ = 'bar'
    id = Column(Integer, primary_key=True)

class Target(Base):
    __tablename__ = 'target'
    id = Column(Integer, primary_key=True)
```

Using Advanced Relationship Arguments (e.g. `primaryjoin`, etc.)

`relationship()` definitions which require explicit primaryjoin, order_by etc. expressions should in all but the most simplistic cases use **late bound** forms for these arguments, meaning, using either the string form or a lambda. The reason for this is that the related `Column` objects which are to be configured using `@declared_attr` are not available to another `@declared_attr` attribute; while the methods will work and return new `Column` objects, those are not the `Column` objects that Declarative will be using as it calls the methods on its own, thus using *different* `Column` objects.

The canonical example is the primaryjoin condition that depends upon another mixed-in column:

```
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==cls.target_id   # this is *incorrect*
        )
```

Mapping a class using the above mixin, we will get an error like:

```
sqlalchemy.exc.InvalidRequestError: this ForeignKey's parent column is not
yet associated with a Table.
```

This is because the `target_id` `Column` we've called upon in our `target()` method is not the same `Column` that declarative is actually going to map to our table.

The condition above is resolved using a lambda:

```
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=lambda: Target.id==cls.target_id
        )
```

or alternatively, the string form (which ultimately generates a lambda):

```
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(), and other MapperProperty classes

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(SomethingMixin, Base):
    __tablename__ = "something"
```

The `column_property()` or other construct may refer to other columns from the mixin. These are copied ahead of time before the `declared_attr` is invoked:

```
class SomethingMixin(object):
    x = Column(Integer)

    y = Column(Integer)
```

```
    @declared_attr
    def x_plus_y(cls):
        return column_property(cls.x + cls.y)
```

Changed in version 1.0.0: mixin columns are copied to the final mapped class so that `declared_attr` methods can access the actual column that will be mapped.

Mixing in Association Proxy and Other Attributes

Mixins can specify user-defined attributes as well as other extension units such as `association_proxy()`. The usage of `declared_attr` is required in those cases where the attribute must be tailored specifically to the target subclass. An example is when constructing multiple `association_proxy()` attributes which each target a different type of child object. Below is an `association_proxy()` / mixin example which provides a scalar list of string values to an implementing class:

```
from sqlalchemy import Column, Integer, ForeignKey, String
from sqlalchemy.orm import relationship
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base, declared_attr

Base = declarative_base()

class HasStringCollection(object):
    @declared_attr
    def _strings(cls):
        class StringAttribute(Base):
            __tablename__ = cls.string_table_name
            id = Column(Integer, primary_key=True)
            value = Column(String(50), nullable=False)
            parent_id = Column(Integer,
                               ForeignKey('%s.id' % cls.__tablename__),
                               nullable=False)
            def __init__(self, value):
                self.value = value

        return relationship(StringAttribute)

    @declared_attr
    def strings(cls):
        return association_proxy('_strings', 'value')

class TypeA(HasStringCollection, Base):
    __tablename__ = 'type_a'
    string_table_name = 'type_a_strings'
    id = Column(Integer(), primary_key=True)

class TypeB(HasStringCollection, Base):
    __tablename__ = 'type_b'
    string_table_name = 'type_b_strings'
    id = Column(Integer(), primary_key=True)
```

Above, the `HasStringCollection` mixin produces a `relationship()` which refers to a newly generated class called `StringAttribute`. The `StringAttribute` class is generated with its own `Table` definition which is local to the parent class making usage of the `HasStringCollection` mixin. It also produces an `association_proxy()` object which proxies references to the `strings` attribute onto the `value` attribute of each `StringAttribute` instance.

`TypeA` or `TypeB` can be instantiated given the constructor argument `strings`, a list of strings:

```
ta = TypeA(strings=['foo', 'bar'])
tb = TypeA(strings=['bat', 'bar'])
```

This list will generate a collection of `StringAttribute` objects, which are persisted into a table that's local to either the `type_a_strings` or `type_b_strings` table:

```
>>> print(ta._strings)
[<__main__.StringAttribute object at 0x10151cd90>,
    <__main__.StringAttribute object at 0x10151ce10>]
```

When constructing the `association_proxy()`, the `declared_attr` decorator must be used so that a distinct `association_proxy()` object is created for each of the `TypeA` and `TypeB` classes.

New in version 0.8: `declared_attr` is usable with non-mapped attributes, including user-defined attributes as well as `association_proxy()`.

Controlling table inheritance with mixins

The `__tablename__` attribute may be used to provide a function that will determine the name of the table used for each class in an inheritance hierarchy, as well as whether a class has its own distinct table.

This is achieved using the `declared_attr` indicator in conjunction with a method named `__tablename__()`. Declarative will always invoke `declared_attr` for the special names `__tablename__`, `__mapper_args__` and `__table_args__` function **for each mapped class in the hierarchy, except if overridden in a subclass**. The function therefore needs to expect to receive each class individually and to provide the correct answer for each.

For example, to create a mixin that gives every class a simple table name based on class name:

```
from sqlalchemy.ext.declarative import declared_attr

class Tablename:
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

class Person(Tablename, Base):
    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, we can modify our `__tablename__` function to return `None` for subclasses, using `has_inherited_table()`. This has the effect of those subclasses being mapped with single table inheritance against the parent:

```
from sqlalchemy.ext.declarative import declared_attr
from sqlalchemy.ext.declarative import has_inherited_table

class Tablename(object):
    @declared_attr
    def __tablename__(cls):
        if has_inherited_table(cls):
            return None
        return cls.__name__.lower()

class Person(Tablename, Base):
```

```
    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'}
```

Mixing in Columns in Inheritance Scenarios

In contrast to how `__tablename__` and other special names are handled when used with `declared_attr`, when we mix in columns and properties (e.g. relationships, column properties, etc.), the function is invoked for the **base class only** in the hierarchy. Below, only the `Person` class will receive a column called `id`; the mapping will fail on `Engineer`, which is not given a primary key:

```
class HasId(object):
    @declared_attr
    def id(cls):
        return Column('id', Integer, primary_key=True)

class Person(HasId, Base):
    __tablename__ = 'person'
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = 'engineer'
    primary_language = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

It is usually the case in joined-table inheritance that we want distinctly named columns on each subclass. However in this case, we may want to have an `id` column on every table, and have them refer to each other via foreign key. We can achieve this as a mixin by using the `declared_attr.cascading` modifier, which indicates that the function should be invoked **for each class in the hierarchy**, in *almost* (see warning below) the same way as it does for `__tablename__`:

```
class HasIdMixin(object):
    @declared_attr.cascading
    def id(cls):
        if has_inherited_table(cls):
            return Column(ForeignKey('person.id'), primary_key=True)
        else:
            return Column(Integer, primary_key=True)

class Person(HasIdMixin, Base):
    __tablename__ = 'person'
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = 'engineer'
    primary_language = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

Warning: The `declared_attr.cascading` feature currently does **not** allow for a subclass to override the attribute with a different function or value. This is a current limitation in the mechanics of how `@declared_attr` is resolved, and a warning is emitted if this condition is detected. This

limitation does **not** exist for the special attribute names such as `__tablename__`, which resolve in a different way internally than that of `declared_attr.cascading`.

New in version 1.0.0: added `declared_attr.cascading`.

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(object):
    __table_args__ = {'mysql_engine':'InnoDB'}

class MyOtherMixin(object):
    __table_args__ = {'info':'foo'}

class MyModel(MySQLSettings, MyOtherMixin, Base):
    __tablename__='my_model'

    @declared_attr
    def __table_args__(cls):
        args = dict()
        args.update(MySQLSettings.__table_args__)
        args.update(MyOtherMixin.__table_args__)
        return args

    id =  Column(Integer, primary_key=True)
```

Creating Indexes with Mixins

To define a named, potentially multicolumn `Index` that applies to all tables derived from a mixin, use the "inline" form of `Index` and establish it as part of `__table_args__`:

```
class MyMixin(object):
    a =  Column(Integer)
    b =  Column(Integer)

    @declared_attr
    def __table_args__(cls):
        return (Index('test_idx_%s' % cls.__tablename__, 'a', 'b'),)

class MyModel(MyMixin, Base):
    __tablename__ = 'atable'
    c =  Column(Integer,primary_key=True)
```

Declarative API

API Reference

sqlalchemy.ext.declarative.**declarative_base**(*bind=None*, *metadata=None*, *mapper=None*, *cls=<class 'object'>*, *name='Base'*, *constructor=<function _declarative_constructor>*, *class_registry=None*, *metaclass=<class 'sqlalchemy.ext.declarative.api.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.
- **class_registry** – optional dictionary that will serve as the registry of class names-> mapped classes when string names are used to identify classes inside of `relationship()` and others. Allows two or more declarative base classes to share the same registry of class names for simplified inter-base relationships.
- **metaclass** – Defaults to `DeclarativeMeta`. A metaclass or ___metaclass___ compatible callable to use as the meta type of the generated declarative base class.

Changed in version 1.1: if `declarative_base.cls` is a single class (rather than a tuple), the constructed base class will inherit its docstring.

See also:

`as_declarative()`

sqlalchemy.ext.declarative.**as_declarative**(***kw*)

Class decorator for `declarative_base()`.

Provides a syntactical shortcut to the `cls` argument sent to `declarative_base()`, allowing the base class to be converted in-place to a "declarative" base:

```
from sqlalchemy.ext.declarative import as_declarative

@as_declarative()
class Base(object):
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()
```

```
    id = Column(Integer, primary_key=True)

class MyMappedClass(Base):
    # ...
```

All keyword arguments passed to `as_declarative()` are passed along to `declarative_base()`.

New in version 0.8.3.

See also:

`declarative_base()`

class sqlalchemy.ext.declarative.**declared_attr**(*fget*, *cascading=False*)

Mark a class-level method as representing the definition of a mapped property or special declarative member name.

@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__}
```

Changed in version 0.8: `declared_attr` can be used with non-ORM or extension attributes, such as user-defined attributes or `association_proxy()` objects, which will be assigned to the class at class construction time.

cascading

Mark a `declared_attr` as cascading.

This is a special-use modifier which indicates that a column or MapperProperty-based declared attribute should be configured distinctly per mapped subclass, within a mapped-inheritance scenario.

Warning: The `declared_attr.cascading` modifier has several limitations:

- The flag **only** applies to the use of `declared_attr` on declarative mixin classes and `__abstract__` classes; it currently has no effect when used on a mapped class directly.
- The flag **only** applies to normally-named attributes, e.g. not any special underscore attributes such as `__tablename__`. On these attributes it has **no** effect.
- The flag currently **does not allow further overrides** down the class hierarchy; if a subclass tries to override the attribute, a warning is emitted and the overridden attribute is skipped. This is a limitation that it is hoped will be resolved at some point.

Below, both MyClass as well as MySubClass will have a distinct `id` Column object established:

```
class HasIdMixin(object):
    @declared_attr.cascading
    def id(cls):
        if has_inherited_table(cls):
            return Column(ForeignKey('myclass.id'), primary_key=True)
        else:
            return Column(Integer, primary_key=True)

class MyClass(HasIdMixin, Base):
    __tablename__ = 'myclass'
    # ...

class MySubClass(MyClass):
    ""
    # ...
```

The behavior of the above configuration is that `MySubClass` will refer to both its own `id` column as well as that of `MyClass` underneath the attribute named `some_id`.

See also:

declarative_inheritance

mixin_inheritance_columns

`sqlalchemy.ext.declarative.api._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.

This is used in declarative mixins to build attributes that behave differently for the base class vs. a subclass in an inheritance hierarchy.

See also:

decl_mixin_inheritance

`sqlalchemy.ext.declarative.synonym_for`(*name*, *map_column=False*)
Decorator that produces an `orm.synonym()` attribute in conjunction with a Python descriptor.

The function being decorated is passed to `orm.synonym()` as the `orm.synonym.descriptor` parameter:

```
class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    _job_status = Column("job_status", String(50))

    @synonym_for("job_status")
    @property
    def job_status(self):
        return "Status: %s" % self._job_status
```

The hybrid properties feature of SQLAlchemy is typically preferred instead of synonyms, which is a more legacy feature.

See also:

synonyms - Overview of synonyms

`orm.synonym()` - the mapper-level function

mapper_hybrids - The Hybrid Attribute extension provides an updated approach to augmenting attribute behavior more flexibly than can be achieved with synonyms.

`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.

`class sqlalchemy.ext.declarative.AbstractConcreteBase`
A helper class for 'concrete' declarative mappings.

`AbstractConcreteBase` will use the `polymorphic_union()` function automatically, against all tables mapped as a subclass to this class. The function is called via the `__declare_last__()` function, which is essentially a hook for the `after_configured()` event.

`AbstractConcreteBase` does produce a mapped class for the base class, however it is not persisted to any table; it is instead mapped directly to the "polymorphic" selectable directly and is only used for selecting. Compare to `ConcreteBase`, which does create a persisted table for the base class.

Example:

```
from sqlalchemy.ext.declarative import AbstractConcreteBase

class Employee(AbstractConcreteBase, Base):
    pass

class Manager(Employee):
    __tablename__ = 'manager'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

```
    manager_data = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity':'manager',
        'concrete':True}
```

The abstract base class is handled by declarative in a special way; at class configuration time, it behaves like a declarative mixin or an `__abstract__` base class. Once classes are configured and mappings are produced, it then gets mapped itself, but after all of its decscendants. This is a very unique system of mapping not found in any other SQLAlchemy system.

Using this approach, we can specify columns and properties that will take place on mapped subclasses, in the way that we normally do as in declarative_mixins:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)

class Employee(AbstractConcreteBase, Base):
    employee_id = Column(Integer, primary_key=True)

    @declared_attr
    def company_id(cls):
        return Column(ForeignKey('company.id'))

    @declared_attr
    def company(cls):
        return relationship("Company")

class Manager(Employee):
    __tablename__ = 'manager'

    name = Column(String(50))
    manager_data = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity':'manager',
        'concrete':True}
```

When we make use of our mappings however, both `Manager` and `Employee` will have an independently usable `.company` attribute:

```
session.query(Employee).filter(Employee.company.has(id=5))
```

Changed in version 1.0.0: - The mechanics of `AbstractConcreteBase` have been reworked to support relationships established directly on the abstract base, without any special configurational steps.

See also:

`ConcreteBase`

concrete_inheritance

inheritance_concrete_helpers

`class sqlalchemy.ext.declarative.ConcreteBase`

A helper class for 'concrete' declarative mappings.

`ConcreteBase` will use the `polymorphic_union()` function automatically, against all tables mapped as a subclass to this class. The function is called via the `__declare_last__()` function, which is essentially a hook for the `after_configured()` event.

`ConcreteBase` produces a mapped table for the class itself. Compare to `AbstractConcreteBase`, which does not.

Example:

```
from sqlalchemy.ext.declarative import ConcreteBase

class Employee(ConcreteBase, Base):
    __tablename__ = 'employee'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    __mapper_args__ = {
                    'polymorphic_identity':'employee',
                    'concrete':True}

class Manager(Employee):
    __tablename__ = 'manager'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))
    __mapper_args__ = {
                    'polymorphic_identity':'manager',
                    'concrete':True}
```

See also:

`AbstractConcreteBase`

concrete_inheritance

inheritance_concrete_helpers

class sqlalchemy.ext.declarative.**DeferredReflection**

A helper class for construction of mappings based on a deferred reflection step.

Normally, declarative can be used with reflection by setting a `Table` object using autoload=True as the `__table__` attribute on a declarative class. The caveat is that the `Table` must be fully reflected, or at the very least have a primary key column, at the point at which a normal declarative mapping is constructed, meaning the `Engine` must be available at class declaration time.

The `DeferredReflection` mixin moves the construction of mappers to be at a later point, after a specific method is called which first reflects all `Table` objects created so far. Classes can define it as such:

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

class MyClass(DeferredReflection, Base):
    __tablename__ = 'mytable'
```

Above, `MyClass` is not yet mapped. After a series of classes have been defined in the above fashion, all tables can be reflected and mappings created using `prepare()`:

```
engine = create_engine("someengine://...")
DeferredReflection.prepare(engine)
```

The `DeferredReflection` mixin can be applied to individual classes, used as the base for the declarative base itself, or used in a custom abstract class. Using an abstract base allows that only a subset of classes to be prepared for a particular prepare step, which is necessary for applications that use more than one engine. For example, if an application has two engines, you might use two bases, and prepare each separately, e.g.:

```
class ReflectedOne(DeferredReflection, Base):
    __abstract__ = True

class ReflectedTwo(DeferredReflection, Base):
```

```
    __abstract__ = True

class MyClass(ReflectedOne):
    __tablename__ = 'mytable'

class MyOtherClass(ReflectedOne):
    __tablename__ = 'myothertable'

class YetAnotherClass(ReflectedTwo):
    __tablename__ = 'yetanothertable'

# ... etc.
```

Above, the class hierarchies for `ReflectedOne` and `ReflectedTwo` can be configured separately:

```
ReflectedOne.prepare(engine_one)
ReflectedTwo.prepare(engine_two)
```

New in version 0.8.

classmethod prepare(*engine*)
Reflect all `Table` objects for all current `DeferredReflection` subclasses

Special Directives

`__declare_last__()`

The `__declare_last__()` hook allows definition of a class level function that is automatically called by the `MapperEvents.after_configured()` event, which occurs after mappings are assumed to be completed and the 'configure' step has finished:

```
class MyClass(Base):
    @classmethod
    def __declare_last__(cls):
        ""
        # do something with mappings
```

`__declare_first__()`

Like `__declare_last__()`, but is called at the beginning of mapper configuration via the `MapperEvents.before_configured()` event:

```
class MyClass(Base):
    @classmethod
    def __declare_first__(cls):
        ""
        # do something before mappings are configured
```

New in version 0.9.3.

`__abstract__`

`__abstract__` causes declarative to skip the production of a table or mapper for the class entirely. A class can be added within a hierarchy in the same way as mixin (see declarative_mixins), allowing subclasses to extend just from the special class:

```
class SomeAbstractBase(Base):
    __abstract__ = True

    def some_helpful_method(self):
        ""

    @declared_attr
    def __mapper_args__(cls):
        return {"helpful mapper arguments":True}

class MyMappedClass(SomeAbstractBase):
    ""
```

One possible use of `__abstract__` is to use a distinct `MetaData` for different bases:

```
Base = declarative_base()

class DefaultBase(Base):
    __abstract__ = True
    metadata = MetaData()

class OtherBase(Base):
    __abstract__ = True
    metadata = MetaData()
```

Above, classes which inherit from `DefaultBase` will use one `MetaData` as the registry of tables, and those which inherit from `OtherBase` will use a different one. The tables themselves can then be created perhaps within distinct databases:

```
DefaultBase.metadata.create_all(some_engine)
OtherBase.metadata_create_all(some_other_engine)
```

`__table_cls__`

Allows the callable / class used to generate a `Table` to be customized. This is a very open-ended hook that can allow special customizations to a `Table` that one generates here:

```
class MyMixin(object):
    @classmethod
    def __table_cls__(cls, name, metadata, *arg, **kw):
        return Table(
            "my_" + name,
            metadata, *arg, **kw
        )
```

The above mixin would cause all `Table` objects generated to include the prefix `"my_"`, followed by the name normally specified using the `__tablename__` attribute.

`__table_cls__` also supports the case of returning `None`, which causes the class to be considered as single-table inheritance vs. its subclass. This may be useful in some customization schemes to determine that single-table inheritance should take place based on the arguments for the table itself, such as, define as single-inheritance if there is no primary key present:

```
class AutoTable(object):
    @declared_attr
    def __tablename__(cls):
        return cls.__name__

    @classmethod
    def __table_cls__(cls, *arg, **kw):
```

```
        for obj in arg[1:]:
            if (isinstance(obj, Column) and obj.primary_key) or \
                    isinstance(obj, PrimaryKeyConstraint):
                return Table(*arg, **kw)

        return None

class Person(AutoTable, Base):
    id = Column(Integer, primary_key=True)

class Employee(Person):
    employee_name = Column(String)
```

The above `Employee` class would be mapped as single-table inheritance against `Person`; the `employee_name` column would be added as a member of the `Person` table.

New in version 1.0.0.

2.7.5 Mutation Tracking

Provide support for tracking of in-place changes to scalar values, which are propagated into ORM change events on owning parent objects.

New in version 0.7: `sqlalchemy.ext.mutable` replaces SQLAlchemy's legacy approach to in-place mutations of scalar values; see 07_migration_mutation_extension.

Establishing Mutability on Scalar Column Values

A typical example of a "mutable" structure is a Python dictionary. Following the example introduced in types_toplevel, we begin with a custom type that marshals Python dictionaries into JSON strings before being persisted:

```
from sqlalchemy.types import TypeDecorator, VARCHAR
import json

class JSONEncodedDict(TypeDecorator):
    "Represents an immutable structure as a json-encoded string."

    impl = VARCHAR

    def process_bind_param(self, value, dialect):
        if value is not None:
            value = json.dumps(value)
        return value

    def process_result_value(self, value, dialect):
        if value is not None:
            value = json.loads(value)
        return value
```

The usage of `json` is only for the purposes of example. The `sqlalchemy.ext.mutable` extension can be used with any type whose target Python type may be mutable, including `PickleType`, *`postgresql.ARRAY`*, etc.

When using the `sqlalchemy.ext.mutable` extension, the value itself tracks all parents which reference it. Below, we illustrate a simple version of the `MutableDict` dictionary object, which applies the `Mutable` mixin to a plain Python dictionary:

```
from sqlalchemy.ext.mutable import Mutable
```

```
class MutableDict(Mutable, dict):
    @classmethod
    def coerce(cls, key, value):
        "Convert plain dictionaries to MutableDict."

        if not isinstance(value, MutableDict):
            if isinstance(value, dict):
                return MutableDict(value)

            # this call will raise ValueError
            return Mutable.coerce(key, value)
        else:
            return value

    def __setitem__(self, key, value):
        "Detect dictionary set events and emit change events."

        dict.__setitem__(self, key, value)
        self.changed()

    def __delitem__(self, key):
        "Detect dictionary del events and emit change events."

        dict.__delitem__(self, key)
        self.changed()
```

The above dictionary class takes the approach of subclassing the Python built-in `dict` to produce a dict subclass which routes all mutation events through `__setitem__`. There are variants on this approach, such as subclassing `UserDict.UserDict` or `collections.MutableMapping`; the part that's important to this example is that the `Mutable.changed()` method is called whenever an in-place change to the datastructure takes place.

We also redefine the `Mutable.coerce()` method which will be used to convert any values that are not instances of `MutableDict`, such as the plain dictionaries returned by the `json` module, into the appropriate type. Defining this method is optional; we could just as well created our `JSONEncodedDict` such that it always returns an instance of `MutableDict`, and additionally ensured that all calling code uses `MutableDict` explicitly. When `Mutable.coerce()` is not overridden, any values applied to a parent object which are not instances of the mutable type will raise a `ValueError`.

Our new `MutableDict` type offers a class method `as_mutable()` which we can use within column metadata to associate with types. This method grabs the given type object or class and associates a listener that will detect all future mappings of this type, applying event listening instrumentation to the mapped attribute. Such as, with classical table metadata:

```
from sqlalchemy import Table, Column, Integer

my_data = Table('my_data', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', MutableDict.as_mutable(JSONEncodedDict))
)
```

Above, `as_mutable()` returns an instance of `JSONEncodedDict` (if the type object was not an instance already), which will intercept any attributes which are mapped against this type. Below we establish a simple mapping against the `my_data` table:

```
from sqlalchemy import mapper

class MyDataClass(object):
    pass

# associates mutation listeners with MyDataClass.data
mapper(MyDataClass, my_data)
```

The MyDataClass.data member will now be notified of in place changes to its value.

There's no difference in usage when using declarative:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class MyDataClass(Base):
    __tablename__ = 'my_data'
    id = Column(Integer, primary_key=True)
    data = Column(MutableDict.as_mutable(JSONEncodedDict))
```

Any in-place changes to the MyDataClass.data member will flag the attribute as "dirty" on the parent object:

```
>>> from sqlalchemy.orm import Session

>>> sess = Session()
>>> m1 = MyDataClass(data={'value1':'foo'})
>>> sess.add(m1)
>>> sess.commit()

>>> m1.data['value1'] = 'bar'
>>> assert m1 in sess.dirty
True
```

The MutableDict can be associated with all future instances of JSONEncodedDict in one step, using associate_with(). This is similar to as_mutable() except it will intercept all occurrences of MutableDict in all mappings unconditionally, without the need to declare it individually:

```
MutableDict.associate_with(JSONEncodedDict)

class MyDataClass(Base):
    __tablename__ = 'my_data'
    id = Column(Integer, primary_key=True)
    data = Column(JSONEncodedDict)
```

Supporting Pickling

The key to the sqlalchemy.ext.mutable extension relies upon the placement of a weakref.WeakKeyDictionary upon the value object, which stores a mapping of parent mapped objects keyed to the attribute name under which they are associated with this value. WeakKeyDictionary objects are not picklable, due to the fact that they contain weakrefs and function callbacks. In our case, this is a good thing, since if this dictionary were picklable, it could lead to an excessively large pickle size for our value objects that are pickled by themselves outside of the context of the parent. The developer responsibility here is only to provide a __getstate__ method that excludes the _parents() collection from the pickle stream:

```
class MyMutableType(Mutable):
    def __getstate__(self):
        d = self.__dict__.copy()
        d.pop('_parents', None)
        return d
```

With our dictionary example, we need to return the contents of the dict itself (and also restore them on ___setstate___):

```
class MutableDict(Mutable, dict):
    # ....
```

```
    def __getstate__(self):
        return dict(self)

    def __setstate__(self, state):
        self.update(state)
```

In the case that our mutable value object is pickled as it is attached to one or more parent objects that are also part of the pickle, the `Mutable` mixin will re-establish the `Mutable._parents` collection on each value object as the owning parents themselves are unpickled.

Receiving Events

The `AttributeEvents.modified()` event handler may be used to receive an event when a mutable scalar emits a change event. This event handler is called when the `attributes.flag_modified()` function is called from within the mutable extension:

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

Base = declarative_base()

class MyDataClass(Base):
    __tablename__ = 'my_data'
    id = Column(Integer, primary_key=True)
    data = Column(MutableDict.as_mutable(JSONEncodedDict))

@event.listens_for(MyDataClass.data, "modified")
def modified_json(instance):
    print("json value modified:", instance.data)
```

Establishing Mutability on Composites

Composites are a special ORM feature which allow a single scalar attribute to be assigned an object value which represents information "composed" from one or more columns from the underlying mapped table. The usual example is that of a geometric "point", and is introduced in mapper_composite.

Changed in version 0.7: The internals of `orm.composite()` have been greatly simplified and in-place mutation detection is no longer enabled by default; instead, the user-defined value must detect changes on its own and propagate them to all owning parents. The `sqlalchemy.ext.mutable` extension provides the helper class `MutableComposite`, which is a slight variant on the `Mutable` class.

As is the case with `Mutable`, the user-defined composite class subclasses `MutableComposite` as a mixin, and detects and delivers change events to its parents via the `MutableComposite.changed()` method. In the case of a composite class, the detection is usually via the usage of Python descriptors (i.e. `@property`), or alternatively via the special Python method `__setattr__()`. Below we expand upon the `Point` class introduced in mapper_composite to subclass `MutableComposite` and to also route attribute set events via `__setattr__` to the `MutableComposite.changed()` method:

```
from sqlalchemy.ext.mutable import MutableComposite

class Point(MutableComposite):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __setattr__(self, key, value):
        "Intercept set events"

        # set the attribute
```

```
        object.__setattr__(self, key, value)

        # alert all parents to the change
        self.changed()

    def __composite_values__(self):
        return self.x, self.y

    def __eq__(self, other):
        return isinstance(other, Point) and \
            other.x == self.x and \
            other.y == self.y

    def __ne__(self, other):
        return not self.__eq__(other)
```

The `MutableComposite` class uses a Python metaclass to automatically establish listeners for any usage of `orm.composite()` that specifies our `Point` type. Below, when `Point` is mapped to the `Vertex` class, listeners are established which will route change events from `Point` objects to each of the `Vertex.start` and `Vertex.end` attributes:

```
from sqlalchemy.orm import composite, mapper
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),
    )

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)
})
```

Any in-place changes to the `Vertex.start` or `Vertex.end` members will flag the attribute as "dirty" on the parent object:

```
>>> from sqlalchemy.orm import Session

>>> sess = Session()
>>> v1 = Vertex(start=Point(3, 4), end=Point(12, 15))
>>> sess.add(v1)
>>> sess.commit()

>>> v1.end.x = 8
>>> assert v1 in sess.dirty
True
```

Coercing Mutable Composites

The `MutableBase.coerce()` method is also supported on composite types. In the case of `MutableComposite`, the `MutableBase.coerce()` method is only called for attribute set operations, not load operations. Overriding the `MutableBase.coerce()` method is essentially equivalent to using a `validates()` validation routine for all attributes which make use of the custom composite type:

```
class Point(MutableComposite):
    # other Point methods
    # ...

    def coerce(cls, key, value):
        if isinstance(value, tuple):
            value = Point(*value)
        elif not isinstance(value, Point):
            raise ValueError("tuple or Point expected")
        return value
```

New in version 0.7.10,0.8.0b2: Support for the `MutableBase.coerce()` method in conjunction with objects of type `MutableComposite`.

Supporting Pickling

As is the case with `Mutable`, the `MutableComposite` helper class uses a `weakref.WeakKeyDictionary` available via the `MutableBase._parents()` attribute which isn't picklable. If we need to pickle instances of `Point` or its owning class `Vertex`, we at least need to define a `__getstate__` that doesn't include the `_parents` dictionary. Below we define both a `__getstate__` and a `__setstate__` that package up the minimal form of our `Point` class:

```
class Point(MutableComposite):
    # ...

    def __getstate__(self):
        return self.x, self.y

    def __setstate__(self, state):
        self.x, self.y = state
```

As with `Mutable`, the `MutableComposite` augments the pickling process of the parent's object-relational state so that the `MutableBase._parents()` collection is restored to all `Point` objects.

API Reference

class sqlalchemy.ext.mutable.**MutableBase**

Common base class to `Mutable` and `MutableComposite`.

_parents

Dictionary of parent object->attribute name on the parent.

This attribute is a so-called "memoized" property. It initializes itself with a new `weakref.WeakKeyDictionary` the first time it is accessed, returning the same object upon subsequent access.

classmethod coerce(*key*, *value*)

Given a value, coerce it into the target type.

Can be overridden by custom subclasses to coerce incoming data into a particular type.

By default, raises `ValueError`.

This method is called in different scenarios depending on if the parent class is of type `Mutable` or of type `MutableComposite`. In the case of the former, it is called for both attribute-set operations as well as during ORM loading operations. For the latter, it is only called during attribute-set operations; the mechanics of the `composite()` construct handle coercion during load operations.

Parameters

- **key** – string name of the ORM-mapped attribute being set.

- **value** – the incoming value.

Returns the method should return the coerced value, or raise `ValueError` if the coercion cannot be completed.

class sqlalchemy.ext.mutable.Mutable

Mixin that defines transparent propagation of change events to a parent object.

See the example in mutable_scalars for usage information.

_get_listen_keys(*attribute*)

Given a descriptor attribute, return a `set()` of the attribute keys which indicate a change in the state of this attribute.

This is normally just `set([attribute.key])`, but can be overridden to provide for additional keys. E.g. a `MutableComposite` augments this set with the attribute keys associated with the columns that comprise the composite value.

This collection is consulted in the case of intercepting the `InstanceEvents.refresh()` and `InstanceEvents.refresh_flush()` events, which pass along a list of attribute names that have been refreshed; the list is compared against this set to determine if action needs to be taken.

New in version 1.0.5.

_listen_on_attribute(*attribute*, *coerce*, *parent_cls*)

Establish this type as a mutation listener for the given mapped descriptor.

_parents

Dictionary of parent object->attribute name on the parent.

This attribute is a so-called "memoized" property. It initializes itself with a new `weakref.WeakKeyDictionary` the first time it is accessed, returning the same object upon subsequent access.

classmethod as_mutable(*sqltype*)

Associate a SQL type with this mutable Python type.

This establishes listeners that will detect ORM mappings against the given type, adding mutation event trackers to those mappings.

The type is returned, unconditionally as an instance, so that `as_mutable()` can be used inline:

```
Table('mytable', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', MyMutableType.as_mutable(PickleType))
)
```

Note that the returned type is always an instance, even if a class is given, and that only columns which are declared specifically with that type instance receive additional instrumentation.

To associate a particular mutable type with all occurrences of a particular type, use the `Mutable.associate_with()` classmethod of the particular `Mutable` subclass to establish a global association.

> **Warning:** The listeners established by this method are *global* to all mappers, and are *not* garbage collected. Only use `as_mutable()` for types that are permanent to an application, not with ad-hoc types else this will cause unbounded growth in memory usage.

classmethod associate_with(*sqltype*)

Associate this wrapper with all future mapped columns of the given type.

This is a convenience method that calls `associate_with_attribute` automatically.

> **Warning:** The listeners established by this method are *global* to all mappers, and are *not* garbage collected. Only use `associate_with()` for types that are permanent to an application, not with ad-hoc types else this will cause unbounded growth in memory usage.

`classmethod associate_with_attribute(`*attribute*`)`
Establish this type as a mutation listener for the given mapped descriptor.

`changed()`
Subclasses should call this method whenever change events occur.

`coerce(`*key*, *value*`)`
Given a value, coerce it into the target type.

Can be overridden by custom subclasses to coerce incoming data into a particular type.

By default, raises `ValueError`.

This method is called in different scenarios depending on if the parent class is of type `Mutable` or of type `MutableComposite`. In the case of the former, it is called for both attribute-set operations as well as during ORM loading operations. For the latter, it is only called during attribute-set operations; the mechanics of the `composite()` construct handle coercion during load operations.

Parameters

- **`key`** – string name of the ORM-mapped attribute being set.
- **`value`** – the incoming value.

Returns the method should return the coerced value, or raise `ValueError` if the coercion cannot be completed.

`class sqlalchemy.ext.mutable.MutableComposite`
Mixin that defines transparent propagation of change events on a SQLAlchemy "composite" object to its owning parent or parents.

See the example in mutable_composites for usage information.

`changed()`
Subclasses should call this method whenever change events occur.

`class sqlalchemy.ext.mutable.MutableDict`
A dictionary type that implements `Mutable`.

The `MutableDict` object implements a dictionary that will emit change events to the underlying mapping when the contents of the dictionary are altered, including when values are added or removed.

Note that `MutableDict` does **not** apply mutable tracking to the *values themselves* inside the dictionary. Therefore it is not a sufficient solution for the use case of tracking deep changes to a *recursive* dictionary structure, such as a JSON structure. To support this use case, build a subclass of `MutableDict` that provides appropriate coersion to the values placed in the dictionary so that they too are "mutable", and emit events up to their parent structure.

New in version 0.8.

See also:

`MutableList`

`MutableSet`

`clear()`

`classmethod coerce(`*key*, *value*`)`
Convert plain dictionary to instance of this class.

`pop(`**arg*`)`

`popitem()`

`setdefault(key, value)`

`update(*a, **kw)`

`class sqlalchemy.ext.mutable.MutableList`

A list type that implements `Mutable`.

The `MutableList` object implements a list that will emit change events to the underlying mapping when the contents of the list are altered, including when values are added or removed.

Note that `MutableList` does **not** apply mutable tracking to the *values themselves* inside the list. Therefore it is not a sufficient solution for the use case of tracking deep changes to a *recursive* mutable structure, such as a JSON structure. To support this use case, build a subclass of `MutableList` that provides appropriate coersion to the values placed in the dictionary so that they too are "mutable", and emit events up to their parent structure.

New in version 1.1.

See also:

`MutableDict`

`MutableSet`

`append(x)`

`clear()`

`classmethod coerce(index, value)`

Convert plain list to instance of this class.

`extend(x)`

`insert(i, x)`

`pop(*arg)`

`remove(i)`

`reverse()`

`sort()`

`class sqlalchemy.ext.mutable.MutableSet`

A set type that implements `Mutable`.

The `MutableSet` object implements a set that will emit change events to the underlying mapping when the contents of the set are altered, including when values are added or removed.

Note that `MutableSet` does **not** apply mutable tracking to the *values themselves* inside the set. Therefore it is not a sufficient solution for the use case of tracking deep changes to a *recursive* mutable structure. To support this use case, build a subclass of `MutableSet` that provides appropriate coersion to the values placed in the dictionary so that they too are "mutable", and emit events up to their parent structure.

New in version 1.1.

See also:

`MutableDict`

`MutableList`

`add(elem)`

`clear()`

`classmethod coerce(index, value)`

Convert plain set to instance of this class.

`difference_update(*arg)`

discard(*elem*)

intersection_update(**arg*)

pop(**arg*)

remove(*elem*)

symmetric_difference_update(**arg*)

update(**arg*)

2.7.6 Ordering List

A custom list that manages index/position information for contained elements.

author Jason Kirtland

`orderinglist` is a helper for mutable ordered relationships. It will intercept list operations performed on a `relationship()`-managed collection and automatically synchronize changes in list position onto a target scalar attribute.

Example: A `slide` table, where each row refers to zero or more entries in a related `bullet` table. The bullets within a slide are displayed in order based on the value of the `position` column in the `bullet` table. As entries are reordered in memory, the value of the `position` attribute should be updated to reflect the new sort order:

```
Base = declarative_base()

class Slide(Base):
    __tablename__ = 'slide'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    bullets = relationship("Bullet", order_by="Bullet.position")

class Bullet(Base):
    __tablename__ = 'bullet'
    id = Column(Integer, primary_key=True)
    slide_id = Column(Integer, ForeignKey('slide.id'))
    position = Column(Integer)
    text = Column(String)
```

The standard relationship mapping will produce a list-like attribute on each `Slide` containing all related `Bullet` objects, but coping with changes in ordering is not handled automatically. When appending a `Bullet` into `Slide.bullets`, the `Bullet.position` attribute will remain unset until manually assigned. When the `Bullet` is inserted into the middle of the list, the following `Bullet` objects will also need to be renumbered.

The `OrderingList` object automates this task, managing the `position` attribute on all `Bullet` objects in the collection. It is constructed using the `ordering_list()` factory:

```
from sqlalchemy.ext.orderinglist import ordering_list

Base = declarative_base()

class Slide(Base):
    __tablename__ = 'slide'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    bullets = relationship("Bullet", order_by="Bullet.position",
```

```
                            collection_class=ordering_list('position'))

class Bullet(Base):
    __tablename__ = 'bullet'
    id = Column(Integer, primary_key=True)
    slide_id = Column(Integer, ForeignKey('slide.id'))
    position = Column(Integer)
    text = Column(String)
```

With the above mapping the `Bullet.position` attribute is managed:

```
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
```

The `OrderingList` construct only works with **changes** to a collection, and not the initial load from the database, and requires that the list be sorted when loaded. Therefore, be sure to specify `order_by` on the `relationship()` against the target ordering attribute, so that the ordering is correct when first loaded.

Warning: `OrderingList` only provides limited functionality when a primary key column or unique column is the target of the sort. Operations that are unsupported or are problematic include:

- two entries must trade values. This is not supported directly in the case of a primary key or unique constraint because it means at least one row would need to be temporarily removed first, or changed to a third, neutral value while the switch occurs.
- an entry must be deleted in order to make room for a new entry. SQLAlchemy's unit of work performs all INSERTs before DELETEs within a single flush. In the case of a primary key, it will trade an INSERT/DELETE of the same primary key for an UPDATE statement in order to lessen the impact of this limitation, however this does not take place for a UNIQUE column. A future feature will allow the "DELETE before INSERT" behavior to be possible, allevating this limitation, though this feature will require explicit configuration at the mapper level for sets of columns that are to be handled in this way.

`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`.

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. e.g.:

```
from sqlalchemy.ext.orderinglist import ordering_list

class Slide(Base):
    __tablename__ = 'slide'

    id = Column(Integer, primary_key=True)
```

```
    name = Column(String)

    bullets = relationship("Bullet", order_by="Bullet.position",
                           collection_class=ordering_list('position'))
```

Parameters

- **attr** – Name of the mapped attribute to use for storage and retrieval of ordering information
- **count_from** – 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.

Additional arguments are passed to the `OrderingList` constructor.

`sqlalchemy.ext.orderinglist.count_from_0(`*index*, *collection*`)`
Numbering function: consecutive integers starting at 0.

`sqlalchemy.ext.orderinglist.count_from_1(`*index*, *collection*`)`
Numbering function: consecutive integers starting at 1.

`sqlalchemy.ext.orderinglist.count_from_n_factory(`*start*`)`
Numbering function: consecutive integers starting at arbitrary start.

`class sqlalchemy.ext.orderinglist.OrderingList(`*ordering_attr=None*, *ordering_func=None*, *reorder_on_append=False*`)`
A custom list that manages position information for its children.

The `OrderingList` object is normally set up using the `ordering_list()` factory function, used in conjunction with the `relationship()` function.

`append(`*object*`)` → None – append object to end

`insert(`*index*, *entity*`)`
L.insert(index, object) – insert object before index

`pop(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`remove(`*value*`)` → None – remove first occurrence of value.
Raises ValueError if the value is not present.

`reorder()`
Synchronize ordering for the entire collection.

Sweeps through the list and ensures that each object has accurate ordering information set.

2.7.7 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 examples_sharding example included in the source distribution.

API Documentation

class sqlalchemy.ext.horizontal_shard.**ShardedSession**(*shard_chooser*, *id_chooser*, *query_chooser*, *shards=None*, *query_cls=<class 'sqlalchemy.ext.horizontal_shard.ShardedQuery'>*, ***kwargs*)

class sqlalchemy.ext.horizontal_shard.**ShardedQuery**(**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.

2.7.8 Hybrid Attributes

Define attributes on ORM-mapped classes that have "hybrid" behavior.

"hybrid" means the attribute has distinct behaviors defined at the class level and at the instance level.

The `hybrid` extension provides a special form of method decorator, is around 50 lines of code and has almost no dependencies on the rest of SQLAlchemy. It can, in theory, work with any descriptor-based expression system.

Consider a mapping `Interval`, representing integer `start` and `end` values. We can define higher level functions on mapped classes that produce SQL expressions at the class level, and Python expression evaluation at the instance level. Below, each function decorated with `hybrid_method` or `hybrid_property` may receive `self` as an instance of the class, or as the class itself:

```
from sqlalchemy import Column, Integer
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import Session, aliased
from sqlalchemy.ext.hybrid import hybrid_property, hybrid_method

Base = declarative_base()

class Interval(Base):
    __tablename__ = 'interval'

    id = Column(Integer, primary_key=True)
    start = Column(Integer, nullable=False)
    end = Column(Integer, nullable=False)

    def __init__(self, start, end):
        self.start = start
        self.end = end

    @hybrid_property
    def length(self):
        return self.end - self.start

    @hybrid_method
    def contains(self, point):
        return (self.start <= point) & (point <= self.end)

    @hybrid_method
    def intersects(self, other):
        return self.contains(other.start) | self.contains(other.end)
```

Above, the `length` property returns the difference between the `end` and `start` attributes. With an instance of `Interval`, this subtraction occurs in Python, using normal Python descriptor mechanics:

```
>>> i1 = Interval(5, 10)
>>> i1.length
5
```

When dealing with the `Interval` class itself, the `hybrid_property` descriptor evaluates the function body given the `Interval` class as the argument, which when evaluated with SQLAlchemy expression mechanics returns a new SQL expression:

```
>>> print Interval.length
interval."end" - interval.start

>>> print Session().query(Interval).filter(Interval.length > 10)
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end
FROM interval
WHERE interval."end" - interval.start > :param_1
```

ORM methods such as `filter_by()` generally use `getattr()` to locate attributes, so can also be used with hybrid attributes:

```
>>> print Session().query(Interval).filter_by(length=5)
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end
FROM interval
WHERE interval."end" - interval.start = :param_1
```

The `Interval` class example also illustrates two methods, `contains()` and `intersects()`, decorated with `hybrid_method`. This decorator applies the same idea to methods that `hybrid_property` applies to attributes. The methods return boolean values, and take advantage of the Python `|` and `&` bitwise operators to produce equivalent instance-level and SQL expression-level boolean behavior:

```
>>> i1.contains(6)
True
>>> i1.contains(15)
False
>>> i1.intersects(Interval(7, 18))
True
>>> i1.intersects(Interval(25, 29))
False

>>> print Session().query(Interval).filter(Interval.contains(15))
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end
FROM interval
WHERE interval.start <= :start_1 AND interval."end" > :end_1

>>> ia = aliased(Interval)
>>> print Session().query(Interval, ia).filter(Interval.intersects(ia))
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end, interval_1.id AS interval_1_id,
interval_1.start AS interval_1_start, interval_1."end" AS interval_1_end
FROM interval, interval AS interval_1
WHERE interval.start <= interval_1.start
    AND interval."end" > interval_1.start
    OR interval.start <= interval_1."end"
    AND interval."end" > interval_1."end"
```

Defining Expression Behavior Distinct from Attribute Behavior

Our usage of the `&` and `|` bitwise operators above was fortunate, considering our functions operated on two boolean values to return a new one. In many cases, the construction of an in-Python function and a SQLAlchemy SQL expression have enough differences that two separate Python expressions should be defined. The `hybrid` decorators define the `hybrid_property.expression()` modifier for this purpose. As an example we'll define the radius of the interval, which requires the usage of the absolute value function:

```
from sqlalchemy import func

class Interval(object):
    # ...

    @hybrid_property
    def radius(self):
        return abs(self.length) / 2

    @radius.expression
    def radius(cls):
        return func.abs(cls.length) / 2
```

Above the Python function `abs()` is used for instance-level operations, the SQL function `ABS()` is used via the `func` object for class-level expressions:

```
>>> i1.radius
2

>>> print Session().query(Interval).filter(Interval.radius > 5)
SELECT interval.id AS interval_id, interval.start AS interval_start,
    interval."end" AS interval_end
FROM interval
WHERE abs(interval."end" - interval.start) / :abs_1 > :param_1
```

Defining Setters

Hybrid properties can also define setter methods. If we wanted `length` above, when set, to modify the endpoint value:

```
class Interval(object):
    # ...

    @hybrid_property
    def length(self):
        return self.end - self.start

    @length.setter
    def length(self, value):
        self.end = self.start + value
```

The `length(self, value)` method is now called upon set:

```
>>> i1 = Interval(5, 10)
>>> i1.length
5
>>> i1.length = 12
>>> i1.end
17
```

Allowing Bulk ORM Update

A hybrid can define a custom "UPDATE" handler for when using the `Query.update()` method, allowing the hybrid to be used in the SET clause of the update.

Normally, when using a hybrid with `Query.update()`, the SQL expression is used as the column that's the target of the SET. If our `Interval` class had a hybrid `start_point` that linked to `Interval.start`, this could be substituted directly:

```
session.query(Interval).update({Interval.start_point: 10})
```

However, when using a composite hybrid like `Interval.length`, this hybrid represents more than one column. We can set up a handler that will accommodate a value passed to `Query.update()` which can affect this, using the `hybrid_propery.update_expression()` decorator. A handler that works similarly to our setter would be:

```
class Interval(object):
    # ...

    @hybrid_property
    def length(self):
        return self.end - self.start

    @length.setter
    def length(self, value):
        self.end = self.start + value

    @length.update_expression
    def length(cls, value):
        return [
            (cls.end, cls.start + value)
        ]
```

Above, if we use `Interval.length` in an UPDATE expression as:

```
session.query(Interval).update(
    {Interval.length: 25}, synchronize_session='fetch')
```

We'll get an UPDATE statement along the lines of:

```
UPDATE interval SET end=start + :value
```

In some cases, the default "evaluate" strategy can't perform the SET expression in Python; while the addition operator we're using above is supported, for more complex SET expressions it will usually be necessary to use either the "fetch" or False synchronization strategy as illustrated above.

New in version 1.2: added support for bulk updates to hybrid properties.

Working with Relationships

There's no essential difference when creating hybrids that work with related objects as opposed to column-based data. The need for distinct expressions tends to be greater. The two variants we'll illustrate are the "join-dependent" hybrid, and the "correlated subquery" hybrid.

Join-Dependent Relationship Hybrid

Consider the following declarative mapping which relates a `User` to a `SavingsAccount`:

```
from sqlalchemy import Column, Integer, ForeignKey, Numeric, String
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.hybrid import hybrid_property

Base = declarative_base()

class SavingsAccount(Base):
    __tablename__ = 'account'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'), nullable=False)
    balance = Column(Numeric(15, 5))

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(100), nullable=False)

    accounts = relationship("SavingsAccount", backref="owner")

    @hybrid_property
    def balance(self):
        if self.accounts:
            return self.accounts[0].balance
        else:
            return None

    @balance.setter
    def balance(self, value):
        if not self.accounts:
            account = Account(owner=self)
        else:
            account = self.accounts[0]
        account.balance = value

    @balance.expression
    def balance(cls):
        return SavingsAccount.balance
```

The above hybrid property `balance` works with the first `SavingsAccount` entry in the list of accounts for this user. The in-Python getter/setter methods can treat `accounts` as a Python list available on `self`.

However, at the expression level, it's expected that the `User` class will be used in an appropriate context such that an appropriate join to `SavingsAccount` will be present:

```
>>> print Session().query(User, User.balance).\
...     join(User.accounts).filter(User.balance > 5000)
SELECT "user".id AS user_id, "user".name AS user_name,
account.balance AS account_balance
FROM "user" JOIN account ON "user".id = account.user_id
WHERE account.balance > :balance_1
```

Note however, that while the instance level accessors need to worry about whether `self.accounts` is even present, this issue expresses itself differently at the SQL expression level, where we basically would use an outer join:

```
>>> from sqlalchemy import or_
>>> print (Session().query(User, User.balance).outerjoin(User.accounts).
...         filter(or_(User.balance < 5000, User.balance == None)))
SELECT "user".id AS user_id, "user".name AS user_name,
account.balance AS account_balance
```

```
FROM "user" LEFT OUTER JOIN account ON "user".id = account.user_id
WHERE account.balance <  :balance_1 OR account.balance IS NULL
```

Correlated Subquery Relationship Hybrid

We can, of course, forego being dependent on the enclosing query's usage of joins in favor of the correlated subquery, which can portably be packed into a single column expression. A correlated subquery is more portable, but often performs more poorly at the SQL level. Using the same technique illustrated at mapper_column_property_sql_expressions, we can adjust our `SavingsAccount` example to aggregate the balances for *all* accounts, and use a correlated subquery for the column expression:

```
from sqlalchemy import Column, Integer, ForeignKey, Numeric, String
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy import select, func

Base = declarative_base()

class SavingsAccount(Base):
    __tablename__ = 'account'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'), nullable=False)
    balance = Column(Numeric(15, 5))

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(100), nullable=False)

    accounts = relationship("SavingsAccount", backref="owner")

    @hybrid_property
    def balance(self):
        return sum(acc.balance for acc in self.accounts)

    @balance.expression
    def balance(cls):
        return select([func.sum(SavingsAccount.balance)]).\
                where(SavingsAccount.user_id==cls.id).\
                label('total_balance')
```

The above recipe will give us the `balance` column which renders a correlated SELECT:

```
>>> print s.query(User).filter(User.balance > 400)
SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE (SELECT sum(account.balance) AS sum_1
FROM account
WHERE account.user_id = "user".id) > :param_1
```

Building Custom Comparators

The hybrid property also includes a helper that allows construction of custom comparators. A comparator object allows one to customize the behavior of each SQLAlchemy expression operator individually. They are useful when creating custom types that have some highly idiosyncratic behavior on the SQL side.

Note: The `hybrid_property.comparator()` decorator introduced in this section **replaces** the use of

the `hybrid_property.expression()` decorator. They cannot be used together.

The example class below allows case-insensitive comparisons on the attribute named `word_insensitive`:

```
from sqlalchemy.ext.hybrid import Comparator, hybrid_property
from sqlalchemy import func, Column, Integer, String
from sqlalchemy.orm import Session
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class CaseInsensitiveComparator(Comparator):
    def __eq__(self, other):
        return func.lower(self.__clause_element__()) == func.lower(other)

class SearchWord(Base):
    __tablename__ = 'searchword'
    id = Column(Integer, primary_key=True)
    word = Column(String(255), nullable=False)

    @hybrid_property
    def word_insensitive(self):
        return self.word.lower()

    @word_insensitive.comparator
    def word_insensitive(cls):
        return CaseInsensitiveComparator(cls.word)
```

Above, SQL expressions against `word_insensitive` will apply the `LOWER()` SQL function to both sides:

```
>>> print Session().query(SearchWord).filter_by(word_insensitive="Trucks")
SELECT searchword.id AS searchword_id, searchword.word AS searchword_word
FROM searchword
WHERE lower(searchword.word) = lower(:lower_1)
```

The `CaseInsensitiveComparator` above implements part of the `ColumnOperators` interface. A "coercion" operation like lowercasing can be applied to all comparison operations (i.e. `eq`, `lt`, `gt`, etc.) using `Operators.operate()`:

```
class CaseInsensitiveComparator(Comparator):
    def operate(self, op, other):
        return op(func.lower(self.__clause_element__()), func.lower(other))
```

Reusing Hybrid Properties across Subclasses

A hybrid can be referred to from a superclass, to allow modifying methods like `hybrid_property.getter()`, `hybrid_property.setter()` to be used to redefine those methods on a subclass. This is similar to how the standard Python `@property` object works:

```
class FirstNameOnly(Base):
    # ...

    first_name = Column(String)

    @hybrid_property
    def name(self):
        return self.first_name

    @name.setter
    def name(self, value):
        self.first_name = value
```

```
class FirstNameLastName(FirstNameOnly):
    # ...

    last_name = Column(String)

    @FirstNameOnly.name.getter
    def name(self):
        return self.first_name + ' ' + self.last_name

    @name.setter
    def name(self, value):
        self.first_name, self.last_name = value.split(' ', 1)
```

Above, the `FirstNameLastName` class refers to the hybrid from `FirstNameOnly.name` to repurpose its getter and setter for the subclass.

When overriding `hybrid_property.expression()` and `hybrid_property.comparator()` alone as the first reference to the superclass, these names conflict with the same-named accessors on the class-level `QueryableAttribute` object returned at the class level. To override these methods when referring directly to the parent class descriptor, add the special qualifier `hybrid_property.overrides`, which will de-reference the instrumented attribute back to the hybrid object:

```
class FirstNameLastName(FirstNameOnly):
    # ...

    last_name = Column(String)

    @FirstNameOnly.overrides.expression
    def name(cls):
        return func.concat(cls.first_name, ' ', cls.last_name)
```

New in version 1.2: Added `hybrid_property.getter()` as well as the ability to redefine accessors per-subclass.

Hybrid Value Objects

Note in our previous example, if we were to compare the `word_insensitive` attribute of a `SearchWord` instance to a plain Python string, the plain Python string would not be coerced to lower case - the `CaseInsensitiveComparator` we built, being returned by `@word_insensitive.comparator`, only applies to the SQL side.

A more comprehensive form of the custom comparator is to construct a *Hybrid Value Object*. This technique applies the target value or expression to a value object which is then returned by the accessor in all cases. The value object allows control of all operations upon the value as well as how compared values are treated, both on the SQL expression side as well as the Python value side. Replacing the previous `CaseInsensitiveComparator` class with a new `CaseInsensitiveWord` class:

```
class CaseInsensitiveWord(Comparator):
    "Hybrid value representing a lower case representation of a word."

    def __init__(self, word):
        if isinstance(word, basestring):
            self.word = word.lower()
        elif isinstance(word, CaseInsensitiveWord):
            self.word = word.word
        else:
            self.word = func.lower(word)

    def operate(self, op, other):
        if not isinstance(other, CaseInsensitiveWord):
```

```
            other = CaseInsensitiveWord(other)
        return op(self.word, other.word)

    def __clause_element__(self):
        return self.word

    def __str__(self):
        return self.word

    key = 'word'
    "Label to apply to Query tuple results"
```

Above, the `CaseInsensitiveWord` object represents `self.word`, which may be a SQL function, or may be a Python native. By overriding `operate()` and `__clause_element__()` to work in terms of `self.word`, all comparison operations will work against the "converted" form of `word`, whether it be SQL side or Python side. Our `SearchWord` class can now deliver the `CaseInsensitiveWord` object unconditionally from a single hybrid call:

```
class SearchWord(Base):
    __tablename__ = 'searchword'
    id = Column(Integer, primary_key=True)
    word = Column(String(255), nullable=False)

    @hybrid_property
    def word_insensitive(self):
        return CaseInsensitiveWord(self.word)
```

The `word_insensitive` attribute now has case-insensitive comparison behavior universally, including SQL expression vs. Python expression (note the Python value is converted to lower case on the Python side here):

```
>>> print Session().query(SearchWord).filter_by(word_insensitive="Trucks")
SELECT searchword.id AS searchword_id, searchword.word AS searchword_word
FROM searchword
WHERE lower(searchword.word) = :lower_1
```

SQL expression versus SQL expression:

```
>>> sw1 = aliased(SearchWord)
>>> sw2 = aliased(SearchWord)
>>> print Session().query(
...                     sw1.word_insensitive,
...                     sw2.word_insensitive).\
...                         filter(
...                             sw1.word_insensitive > sw2.word_insensitive
...                         )
SELECT lower(searchword_1.word) AS lower_1,
lower(searchword_2.word) AS lower_2
FROM searchword AS searchword_1, searchword AS searchword_2
WHERE lower(searchword_1.word) > lower(searchword_2.word)
```

Python only expression:

```
>>> ws1 = SearchWord(word="SomeWord")
>>> ws1.word_insensitive == "sOmEwOrD"
True
>>> ws1.word_insensitive == "XOmEwOrX"
False
>>> print ws1.word_insensitive
someword
```

The Hybrid Value pattern is very useful for any kind of value that may have multiple representations,

such as timestamps, time deltas, units of measurement, currencies and encrypted passwords.

See also:

Hybrids and Value Agnostic Types - on the techspot.zzzeek.org blog

Value Agnostic Types, Part II - on the techspot.zzzeek.org blog

Building Transformers

A *transformer* is an object which can receive a `Query` object and return a new one. The `Query` object includes a method `with_transformation()` that returns a new `Query` transformed by the given function.

We can combine this with the `Comparator` class to produce one type of recipe which can both set up the FROM clause of a query as well as assign filtering criterion.

Consider a mapped class `Node`, which assembles using adjacency list into a hierarchical tree pattern:

```
from sqlalchemy import Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()

class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    parent = relationship("Node", remote_side=id)
```

Suppose we wanted to add an accessor `grandparent`. This would return the `parent` of `Node.parent`. When we have an instance of `Node`, this is simple:

```
from sqlalchemy.ext.hybrid import hybrid_property

class Node(Base):
    # ...

    @hybrid_property
    def grandparent(self):
        return self.parent.parent
```

For the expression, things are not so clear. We'd need to construct a `Query` where we `join()` twice along `Node.parent` to get to the `grandparent`. We can instead return a transforming callable that we'll combine with the `Comparator` class to receive any `Query` object, and return a new one that's joined to the `Node.parent` attribute and filtered based on the given criterion:

```
from sqlalchemy.ext.hybrid import Comparator

class GrandparentTransformer(Comparator):
    def operate(self, op, other):
        def transform(q):
            cls = self.__clause_element__()
            parent_alias = aliased(cls)
            return q.join(parent_alias, cls.parent).\
                        filter(op(parent_alias.parent, other))
        return transform

Base = declarative_base()

class Node(Base):
    __tablename__ = 'node'
    id =Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
```

```
    parent = relationship("Node", remote_side=id)

    @hybrid_property
    def grandparent(self):
        return self.parent.parent

    @grandparent.comparator
    def grandparent(cls):
        return GrandparentTransformer(cls)
```

The `GrandparentTransformer` overrides the core `Operators.operate()` method at the base of the `Comparator` hierarchy to return a query-transforming callable, which then runs the given comparison operation in a particular context. Such as, in the example above, the `operate` method is called, given the `Operators.eq` callable as well as the right side of the comparison `Node(id=5)`. A function `transform` is then returned which will transform a `Query` first to join to `Node.parent`, then to compare `parent_alias` using `Operators.eq` against the left and right sides, passing into `Query.filter`:

```
>>> from sqlalchemy.orm import Session
>>> session = Session()
{sql}>>> session.query(Node).\
...         with_transformation(Node.grandparent==Node(id=5)).\
...         all()
SELECT node.id AS node_id, node.parent_id AS node_parent_id
FROM node JOIN node AS node_1 ON node_1.id = node.parent_id
WHERE :param_1 = node_1.parent_id
{stop}
```

We can modify the pattern to be more verbose but flexible by separating the "join" step from the "filter" step. The tricky part here is ensuring that successive instances of `GrandparentTransformer` use the same `AliasedClass` object against `Node`. Below we use a simple memoizing approach that associates a `GrandparentTransformer` with each class:

```
class Node(Base):

    # ...

    @grandparent.comparator
    def grandparent(cls):
        # memoize a GrandparentTransformer
        # per class
        if '_gp' not in cls.__dict__:
            cls._gp = GrandparentTransformer(cls)
        return cls._gp

class GrandparentTransformer(Comparator):

    def __init__(self, cls):
        self.parent_alias = aliased(cls)

    @property
    def join(self):
        def go(q):
            return q.join(self.parent_alias, Node.parent)
        return go

    def operate(self, op, other):
        return op(self.parent_alias.parent, other)
```

```
{sql}>>> session.query(Node).\
...             with_transformation(Node.grandparent.join).\
...             filter(Node.grandparent==Node(id=5))
```

```
SELECT node.id AS node_id, node.parent_id AS node_parent_id
FROM node JOIN node AS node_1 ON node_1.id = node.parent_id
WHERE :param_1 = node_1.parent_id
{stop}
```

The "transformer" pattern is an experimental pattern that starts to make usage of some functional programming paradigms. While it's only recommended for advanced and/or patient developers, there's probably a whole lot of amazing things it can be used for.

API Reference

class sqlalchemy.ext.hybrid.**hybrid_method**(*func*, *expr=None*)

A decorator which allows definition of a Python object method with both instance-level and class-level behavior.

expression(*expr*)

Provide a modifying decorator that defines a SQL-expression producing method.

class sqlalchemy.ext.hybrid.**hybrid_property**(*fget*, *fset=None*, *fdel=None*, *expr=None*, *custom_comparator=None*, *update_expr=None*)

A decorator which allows definition of a Python descriptor with both instance-level and class-level behavior.

comparator(*comparator*)

Provide a modifying decorator that defines a custom comparator producing method.

The return value of the decorated method should be an instance of `Comparator`.

Note: The `hybrid_property.comparator()` decorator **replaces** the use of the `hybrid_property.expression()` decorator. They cannot be used together.

When a hybrid is invoked at the class level, the `Comparator` object given here is wrapped inside of a specialized `QueryableAttribute`, which is the same kind of object used by the ORM to represent other mapped attributes. The reason for this is so that other class-level attributes such as docstrings and a reference to the hybrid itself may be maintained within the structure that's returned, without any modifications to the original comparator object passed in.

Note: when referring to a hybrid property from an owning class (e.g. `SomeClass.some_hybrid`), an instance of `QueryableAttribute` is returned, representing the expression or comparator object as this hybrid object. However, that object itself has accessors called `expression` and `comparator`; so when attempting to override these decorators on a subclass, it may be necessary to qualify it using the `hybrid_property.overrides` modifier first. See that modifier for details.

---

deleter(*fdel*)

Provide a modifying decorator that defines a deletion method.

expression(*expr*)

Provide a modifying decorator that defines a SQL-expression producing method.

When a hybrid is invoked at the class level, the SQL expression given here is wrapped inside of a specialized `QueryableAttribute`, which is the same kind of object used by the ORM to represent other mapped attributes. The reason for this is so that other class-level attributes such as docstrings and a reference to the hybrid itself may be maintained within the structure that's returned, without any modifications to the original SQL expression passed in.

Note: when referring to a hybrid property from an owning class (e.g. `SomeClass.some_hybrid`), an instance of `QueryableAttribute` is returned, representing the expression or comparator object as well as this hybrid object. However, that object itself has accessors called `expression` and `comparator`; so when attempting to override these decorators on a subclass, it may be necessary to qualify it using the `hybrid_property.overrides` modifier first. See that modifier for details.

`getter`(*fget*)
Provide a modifying decorator that defines a getter method.

New in version 1.2.

`overrides`
Prefix for a method that is overriding an existing attribute.

The `hybrid_property.overrides` accessor just returns this hybrid object, which when called at the class level from a parent class, will de-reference the "instrumented attribute" normally returned at this level, and allow modifying decorators like `hybrid_property.expression()` and `hybrid_property.comparator()` to be used without conflicting with the same-named attributes normally present on the `QueryableAttribute`:

```
class SuperClass(object):
    # ...

    @hybrid_property
    def foobar(self):
        return self._foobar

class SubClass(SuperClass):
    # ...

    @SuperClass.foobar.overrides.expression
    def foobar(cls):
        return func.subfoobar(self._foobar)
```

New in version 1.2.

See also:

hybrid_reuse_subclass

`setter`(*fset*)
Provide a modifying decorator that defines a setter method.

`update_expression`(*meth*)
Provide a modifying decorator that defines an UPDATE tuple producing method.

The method accepts a single value, which is the value to be rendered into the SET clause of an UPDATE statement. The method should then process this value into individual column expressions that fit into the ultimate SET clause, and return them as a sequence of 2-tuples. Each tuple contains a column expression as the key and a value to be rendered.

E.g.:

```
class Person(Base):
    # ...

    first_name = Column(String)
    last_name = Column(String)

    @hybrid_property
    def fullname(self):
        return first_name + " " + last_name
```

```
@fullname.update_expression
def fullname(cls, value):
    fname, lname = value.split(" ", 1)
    return [
        (cls.first_name, fname),
        (cls.last_name, lname)
    ]
```

New in version 1.2.

class sqlalchemy.ext.hybrid.**Comparator**(*expression*)

A helper class that allows easy construction of custom `PropComparator` classes for usage with hybrids.

sqlalchemy.ext.hybrid.**HYBRID_METHOD** = symbol('HYBRID_METHOD')

sqlalchemy.ext.hybrid.**HYBRID_PROPERTY** = symbol('HYBRID_PROPERTY')

2.7.9 Indexable

Define attributes on ORM-mapped classes that have "index" attributes for columns with `Indexable` types.

"index" means the attribute is associated with an element of an `Indexable` column with the predefined index to access it. The `Indexable` types include types such as `ARRAY`, `JSON` and *HSTORE*.

The `indexable` extension provides `Column`-like interface for any element of an `Indexable` typed column. In simple cases, it can be treated as a `Column` - mapped attribute.

New in version 1.1.

Synopsis

Given `Person` as a model with a primary key and JSON data field. While this field may have any number of elements encoded within it, we would like to refer to the element called `name` individually as a dedicated attribute which behaves like a standalone column:

```
from sqlalchemy import Column, JSON, Integer
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.indexable import index_property

Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'

    id = Column(Integer, primary_key=True)
    data = Column(JSON)

    name = index_property('data', 'name')
```

Above, the `name` attribute now behaves like a mapped column. We can compose a new `Person` and set the value of `name`:

```
>>> person = Person(name='Alchemist')
```

The value is now accessible:

```
>>> person.name
'Alchemist'
```

Behind the scenes, the JSON field was initialized to a new blank dictionary and the field was set:

```
>>> person.data
{"name": "Alchemist'}
```

The field is mutable in place:

```
>>> person.name = 'Renamed'
>>> person.name
'Renamed'
>>> person.data
{'name': 'Renamed'}
```

When using `index_property`, the change that we make to the indexable structure is also automatically tracked as history; we no longer need to use `MutableDict` in order to track this change for the unit of work.

Deletions work normally as well:

```
>>> del person.name
>>> person.data
{}
```

Above, deletion of `person.name` deletes the value from the dictionary, but not the dictionary itself.

A missing key will produce `AttributeError`:

```
>>> person = Person()
>>> person.name
...
AttributeError: 'name'
```

Unless you set a default value:

```
>>> class Person(Base):
>>>     __tablename__ = 'person'
>>>
>>>     id = Column(Integer, primary_key=True)
>>>     data = Column(JSON)
>>>
>>>     name = index_property('data', 'name', default=None)  # See default

>>> person = Person()
>>> print(person.name)
None
```

The attributes are also accessible at the class level. Below, we illustrate `Person.name` used to generate an indexed SQL criteria:

```
>>> from sqlalchemy.orm import Session
>>> session = Session()
>>> query = session.query(Person).filter(Person.name == 'Alchemist')
```

The above query is equivalent to:

```
>>> query = session.query(Person).filter(Person.data['name'] == 'Alchemist')
```

Multiple `index_property` objects can be chained to produce multiple levels of indexing:

```
from sqlalchemy import Column, JSON, Integer
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.indexable import index_property
```

```
Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'

    id = Column(Integer, primary_key=True)
    data = Column(JSON)

    birthday = index_property('data', 'birthday')
    year = index_property('birthday', 'year')
    month = index_property('birthday', 'month')
    day = index_property('birthday', 'day')
```

Above, a query such as:

```
q = session.query(Person).filter(Person.year == '1980')
```

On a PostgreSQL backend, the above query will render as:

```
SELECT person.id, person.data
FROM person
WHERE person.data -> %(data_1)s -> %(param_1)s = %(param_2)s
```

Default Values

`index_property` includes special behaviors for when the indexed data structure does not exist, and a set operation is called:

- For an `index_property` that is given an integer index value, the default data structure will be a Python list of `None` values, at least as long as the index value; the value is then set at its place in the list. This means for an index value of zero, the list will be initialized to `[None]` before setting the given value, and for an index value of five, the list will be initialized to `[None, None, None, None, None]` before setting the fifth element to the given value. Note that an existing list is **not** extended in place to receive a value.
- for an `index_property` that is given any other kind of index value (e.g. strings usually), a Python dictionary is used as the default data structure.
- The default data structure can be set to any Python callable using the `index_property.datatype` parameter, overriding the previous rules.

Subclassing

`index_property` can be subclassed, in particular for the common use case of providing coercion of values or SQL expressions as they are accessed. Below is a common recipe for use with a PostgreSQL JSON type, where we want to also include automatic casting plus `astext()`:

```
class pg_json_property(index_property):
    def __init__(self, attr_name, index, cast_type):
        super(pg_json_property, self).__init__(attr_name, index)
        self.cast_type = cast_type

    def expr(self, model):
        expr = super(pg_json_property, self).expr(model)
        return expr.astext.cast(self.cast_type)
```

The above subclass can be used with the PostgreSQL-specific version of *postgresql.JSON*:

```
from sqlalchemy import Column, Integer
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.dialects.postgresql import JSON

Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'

    id = Column(Integer, primary_key=True)
    data = Column(JSON)

    age = pg_json_property('data', 'age', Integer)
```

The `age` attribute at the instance level works as before; however when rendering SQL, PostgreSQL's `->>` operator will be used for indexed access, instead of the usual index opearator of `->`:

```
>>> query = session.query(Person).filter(Person.age < 20)
```

The above query will render:

```
SELECT person.id, person.data
FROM person
WHERE CAST(person.data ->> %(data_1)s AS INTEGER) < %(param_1)s
```

API Reference

class sqlalchemy.ext.indexable.**index_property**(*attr_name*, *index*, *default=<object object>*, *datatype=None*, *mutable=True*, *onebased=True*)

A property generator. The generated property describes an object attribute that corresponds to an `Indexable` column.

New in version 1.1.

See also:

`sqlalchemy.ext.indexable`

2.7.10 Alternate Class Instrumentation

Extensible class instrumentation.

The `sqlalchemy.ext.instrumentation` package provides for alternate systems of class instrumentation within the ORM. Class instrumentation refers to how the ORM places attributes on the class which maintain data and track changes to that data, as well as event hooks installed on the class.

Note: The extension package is provided for the benefit of integration with other object management packages, which already perform their own instrumentation. It is not intended for general use.

For examples of how the instrumentation extension is used, see the example examples_instrumentation.

Changed in version 0.8: The `sqlalchemy.orm.instrumentation` was split out so that all functionality having to do with non-standard instrumentation was moved out to `sqlalchemy.ext.instrumentation`. When imported, the module installs itself within `sqlalchemy.orm.instrumentation` so that it takes effect, including recognition of `__sa_instrumentation_manager__` on mapped classes, as well `instrumentation_finders` being used to determine class instrumentation resolution.

API Reference

sqlalchemy.ext.instrumentation.**INSTRUMENTATION_MANAGER = '__sa_instrumentation_manager__'**
Attribute, elects custom instrumentation when present on a mapped class.

Allows a class to specify a slightly or wildly different technique for tracking changes made to mapped attributes and collections.

Only one instrumentation implementation is allowed in a given object inheritance hierarchy.

The value of this attribute must be a callable and will be passed a class object. The callable must return one of:

- An instance of an InstrumentationManager or subclass
- An object implementing all or some of InstrumentationManager (TODO)
- A dictionary of callables, implementing all or some of the above (TODO)
- An instance of a ClassManager or subclass

This attribute is consulted by SQLAlchemy instrumentation resolution, once the `sqlalchemy.ext.instrumentation` module has been imported. If custom finders are installed in the global instrumentation_finders list, they may or may not choose to honor this attribute.

class sqlalchemy.orm.instrumentation.**InstrumentationFactory**
Factory for new ClassManager instances.

class sqlalchemy.ext.instrumentation.**InstrumentationManager**(*class_*)
User-defined class instrumentation extension.

`InstrumentationManager` can be subclassed in order 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. For interception of class instrumentation events, see `InstrumentationEvents`.

The API for this class should be considered as semi-stable, and may change slightly with new releases.

Changed in version 0.8: `InstrumentationManager` was moved from `sqlalchemy.orm.instrumentation` to `sqlalchemy.ext.instrumentation`.

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*)

sqlalchemy.ext.instrumentation.instrumentation_finders = [<function find_native_user_instrumentation
An extensible sequence of callables which return instrumentation implementations

When a class is registered, each callable will be passed a class object. If None is returned, the next finder in the sequence is consulted. Otherwise the return must be an instrumentation factory that follows the same guidelines as sqlalchemy.ext.instrumentation.INSTRUMENTATION_MANAGER.

By default, the only finder is find_native_user_instrumentation_hook, which searches for INSTRUMENTATION_MANAGER. If all finders return None, standard ClassManager instrumentation is used.

class sqlalchemy.ext.instrumentation.ExtendedInstrumentationRegistry
Extends InstrumentationFactory with additional bookkeeping, to accommodate multiple types of class managers.

2.8 ORM 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. Descriptions and source code for all can be found here.

Additional SQLAlchemy examples, some user contributed, are available on the wiki at http://www.sqlalchemy.org/trac/wiki/UsageRecipes.

2.8.1 Mapping Recipes

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)
```

Associations

Examples illustrating the usage of the "association object" pattern, where an intermediary class mediates the relationship between two classes that are associated in a many-to-many pattern.

Directed 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()
```

Dynamic Relations as Dictionaries

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.

Generic Associations

Illustrates various methods of associating multiple types of parents with a particular child object.

The examples all use the declarative extension along with declarative mixins. Each one presents the identical use case at the end - two classes, `Customer` and `Supplier`, both subclassing the `HasAddresses` mixin, which ensures that the parent class is provided with an `addresses` collection which contains `Address` objects.

The **:viewsource:'.discriminator_on_association'** and **:viewsource:'.generic_fk'** scripts are modernized versions of recipes presented in the 2007 blog post Polymorphic Associations with SQLAlchemy.

Large Collections

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.

Materialized Paths

Illustrates the "materialized paths" pattern for hierarchical data using the SQLAlchemy ORM.

Nested Sets

Illustrates a rudimentary way to implement the "nested sets" pattern for hierarchical data using the SQLAlchemy ORM.

Performance

A performance profiling suite for a variety of SQLAlchemy use cases.

Each suite focuses on a specific use case with a particular performance profile and associated implications:

- bulk inserts
- individual inserts, with or without transactions
- fetching large numbers of rows
- running lots of short queries

All suites include a variety of use patterns illustrating both Core and ORM use, and are generally sorted in order of performance from worst to greatest, inversely based on amount of functionality provided by SQLAlchemy, greatest to least (these two things generally correspond perfectly).

A command line tool is presented at the package level which allows individual suites to be run:

```
$ python -m examples.performance --help
usage: python -m examples.performance [-h] [--test TEST] [--dburl DBURL]
                                      [--num NUM] [--profile] [--dump]
                                      [--runsnake] [--echo]

                                      {bulk_inserts,large_resultsets,single_inserts}

positional arguments:
  {bulk_inserts,large_resultsets,single_inserts}
                        suite to run

optional arguments:
  -h, --help            show this help message and exit
  --test TEST           run specific test name
  --dburl DBURL         database URL, default sqlite:///profile.db
  --num NUM             Number of iterations/items/etc for tests; default is 0
                        module-specific
  --profile             run profiling and dump call counts
  --dump                dump full call profile (implies --profile)
  --runsnake            invoke runsnakerun (implies --profile)
  --echo                Echo SQL output
```

An example run looks like:

```
$ python -m examples.performance bulk_inserts
```

Or with options:

```
$ python -m examples.performance bulk_inserts \
    --dburl mysql+mysqldb://scott:tiger@localhost/test \
    --profile --num 1000
```

See also:

faq_how_to_profile

File Listing

Running all tests with time

This is the default form of run:

```
$ python -m examples.performance single_inserts
Tests to run: test_orm_commit, test_bulk_save,
              test_bulk_insert_dictionaries, test_core,
              test_core_query_caching, test_dbapi_raw_w_connect,
              test_dbapi_raw_w_pool

test_orm_commit : Individual INSERT/COMMIT pairs via the
    ORM (10000 iterations); total time 13.690218 sec
test_bulk_save : Individual INSERT/COMMIT pairs using
    the "bulk" API  (10000 iterations); total time 11.290371 sec
test_bulk_insert_dictionaries : Individual INSERT/COMMIT pairs using
    the "bulk" API with dictionaries (10000 iterations);
    total time 10.814626 sec
test_core : Individual INSERT/COMMIT pairs using Core.
    (10000 iterations); total time 9.665620 sec
test_core_query_caching : Individual INSERT/COMMIT pairs using Core
    with query caching (10000 iterations); total time 9.209010 sec
test_dbapi_raw_w_connect : Individual INSERT/COMMIT pairs w/ DBAPI +
    connection each time (10000 iterations); total time 9.551103 sec
```

```
test_dbapi_raw_w_pool : Individual INSERT/COMMIT pairs w/ DBAPI +
    connection pool (10000 iterations); total time 8.001813 sec
```

Dumping Profiles for Individual Tests

A Python profile output can be dumped for all tests, or more commonly individual tests:

```
$ python -m examples.performance single_inserts --test test_core --num 1000 --dump
Tests to run: test_core
test_core : Individual INSERT/COMMIT pairs using Core. (1000 iterations); total fn calls 186109
         186109 function calls (186102 primitive calls) in 1.089 seconds

   Ordered by: internal time, call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.634    0.001    0.634    0.001 {method 'commit' of 'sqlite3.Connection' objects}
     1000    0.154    0.000    0.154    0.000 {method 'execute' of 'sqlite3.Cursor' objects}
     1000    0.021    0.000    0.074    0.000 /Users/classic/dev/sqlalchemy/lib/sqlalchemy/sql/
↪compiler.py:1950(_get_colparams)
     1000    0.015    0.000    0.034    0.000 /Users/classic/dev/sqlalchemy/lib/sqlalchemy/
↪engine/default.py:503(_init_compiled)
        1    0.012    0.012    1.091    1.091 examples/performance/single_inserts.py:79(test_
↪core)

    ...
```

Using RunSnake

This option requires the RunSnake command line tool be installed:

```
$ python -m examples.performance single_inserts --test test_core --num 1000 --runsnake
```

A graphical RunSnake output will be displayed.

Writing your Own Suites

The profiler suite system is extensible, and can be applied to your own set of tests. This is a valuable technique to use in deciding upon the proper approach for some performance-critical set of routines. For example, if we wanted to profile the difference between several kinds of loading, we can create a file `test_loads.py`, with the following content:

```
from examples.performance import Profiler
from sqlalchemy import Integer, Column, create_engine, ForeignKey
from sqlalchemy.orm import relationship, joinedload, subqueryload, Session
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
engine = None
session = None


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

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


# Init with name of file, default number of items
Profiler.init("test_loads", 1000)


@Profiler.setup_once
def setup_once(dburl, echo, num):
    "setup once.  create an engine, insert fixture data"
    global engine
    engine = create_engine(dburl, echo=echo)
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)
    sess = Session(engine)
    sess.add_all([
        Parent(children=[Child() for j in range(100)])
        for i in range(num)
    ])
    sess.commit()


@Profiler.setup
def setup(dburl, echo, num):
    "setup per test.  create a new Session."
    global session
    session = Session(engine)
    # pre-connect so this part isn't profiled (if we choose)
    session.connection()


@Profiler.profile
def test_lazyload(n):
    "load everything, no eager loading."

    for parent in session.query(Parent):
        parent.children


@Profiler.profile
def test_joinedload(n):
    "load everything, joined eager loading."

    for parent in session.query(Parent).options(joinedload("children")):
        parent.children


@Profiler.profile
def test_subqueryload(n):
    "load everything, subquery eager loading."

    for parent in session.query(Parent).options(subqueryload("children")):
        parent.children

if __name__ == '__main__':
    Profiler.main()
```

We can run our new script directly:

```
$ python test_loads.py  --dburl postgresql+psycopg2://scott:tiger@localhost/test
Running setup once...
Tests to run: test_lazyload, test_joinedload, test_subqueryload
test_lazyload : load everything, no eager loading. (1000 iterations); total time 11.971159 sec
test_joinedload : load everything, joined eager loading. (1000 iterations); total time 2.
↪754592 sec
test_subqueryload : load everything, subquery eager loading. (1000 iterations); total time 2.
↪977696 sec
```

As well as see RunSnake output for an individual test:

```
$ python test_loads.py  --num 100 --runsnake --test test_joinedload
```

Relationship Join Conditions

Examples of various `orm.relationship()` configurations, which make use of the `primaryjoin` argument to compose special types of join conditions.

XML Persistence

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.

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
```

Versioning Objects

Versioning with a History Table

Illustrates an extension which creates version tables for entities and stores records for each change. 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:

```
cd examples/versioning
nosetests -v
```

A fragment of example usage, using declarative:

```
from history_meta import Versioned, versioned_session

Base = declarative_base()

class SomeClass(Versioned, 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

Session = sessionmaker(bind=engine)
versioned_session(Session)

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')]
```

The `Versioned` mixin is designed to work with declarative. To use the extension with classical 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_
```

Versioning using Temporal Rows

Illustrates an extension which versions data by storing new rows for each change; that is, what would normally be an UPDATE becomes an INSERT.

Vertical Attribute Mapping

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()
```

2.8.2 Inheritance Mapping Recipes

Basic Inheritance Mappings

Working examples of single-table, joined-table, and concrete-table inheritance as described in inheritance_toplevel.

2.8.3 Special APIs

Attribute Instrumentation

Examples illustrating modifications to SQLAlchemy's attribute management system.

Horizontal 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.

The construction of generic sharding routines is an ambitious approach to the issue of organizing instances among multiple databases. For a more plain-spoken alternative, the "distinct entity" approach is a simple method of assigning objects to different tables (and potentially database nodes) in an explicit way - described on the wiki at EntityName.

2.8.4 Extending the ORM

Dogpile Caching

Illustrates how to embed dogpile.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.

Changed in version 0.8: The example was modernized to use dogpile.cache, replacing Beaker as the caching library in use.

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 dogpile.cache, 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"))

# specify that each Person's "addresses" collection comes from
# cache too
q = q.options(RelationshipCache(Person.addresses, "default"))

# query
print q.all()
```

To run, both SQLAlchemy and dogpile.cache 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 Python modules so that relative imports work:

```
python -m examples.dogpile_caching.helloworld

python -m examples.dogpile_caching.relationship_caching

python -m examples.dogpile_caching.advanced

python -m examples.dogpile_caching.local_session_caching
```

PostGIS Integration

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()
```

CHAPTER

THREE

SQLALCHEMY CORE

The breadth of SQLAlchemy's SQL rendering engine, DBAPI integration, transaction integration, and schema description services are documented here. In contrast to the ORM's domain-centric mode of usage, the SQL Expression Language provides a schema-centric usage paradigm.

3.1 SQL Expression Language Tutorial

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 ormtutorial_toplevel, 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.

3.1.1 Version Check

A quick check to verify that we are on at least **version 1.2** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__  # doctest: +SKIP
1.2.0
```

3.1.2 Connecting

For this tutorial we will use an in-memory-only SQLite database. This is an easy way to test things without needing to have an actual database defined anywhere. To connect we use `create_engine()`:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:', echo=True)
```

The `echo` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard `logging` module. With it enabled, we'll see all the generated SQL produced. If you are working through this tutorial and want less output generated, set it to `False`. This tutorial will format the SQL behind a popup window so it doesn't get in our way; just click the "SQL" links to see what's being generated.

The return value of `create_engine()` is an instance of `Engine`, and it represents the core interface to the database, adapted through a dialect that handles the details of the database and DBAPI in use. In this case the SQLite dialect will interpret instructions to the Python built-in `sqlite3` module.

Lazy Connecting

The `Engine`, when first returned by `create_engine()`, has not actually tried to connect to the database yet; that happens only the first time it is asked to perform a task against the database.

The first time a method like `Engine.execute()` or `Engine.connect()` is called, the `Engine` establishes a real DBAPI connection to the database, which is then used to emit the SQL.

See also:

database_urls - includes examples of `create_engine()` connecting to several kinds of databases with links to more information.

3.1.3 Define and Create Tables

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

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

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:

```
{sql}>>> metadata.create_all(engine)
SE...
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))
)
```

We include this more verbose `Table` construct separately to highlight the difference between a minimal construct geared primarily towards in-Python usage only, versus one that will be used to emit CREATE TABLE statements on a particular set of backends with more stringent requirements.

3.1.4 Insert Expressions

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

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

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

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

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

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

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

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

3.1.5 Executing

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

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

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

```
>>> result = conn.execute(ins)
{opensql}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 using `ResultProxy.inserted_primary_key`:

```
>>> 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 `ResultProxy.inserted_primary_key` returns a list so that it supports composite primary keys). Methods here range from using `cursor.lastrowid`, to selecting from a database-specific function, to using `INSERT..RETURNING` syntax; this all occurs transparently.

3.1.6 Executing Multiple Statements

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

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

Above, because we specified all three columns in 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'},
... ])
{opensql}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
{stop}<sqlalchemy.engine.result.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.

The "executemany" style of invocation is available for each of the `insert()`, `update()` and `delete()` constructs.

3.1.7 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)
{opensql}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')
```

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:

```
{sql}>>> result = conn.execute(s)
SELECT users.id, users.name, users.fullname
FROM users
()

{stop}>>> 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:

```
{sql}>>> 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
()
{stop}name: jack ; fullname: Jack Jones
name: wendy ; fullname: Wendy Williams
```

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])
{sql}>>> result = conn.execute(s)
SELECT users.name, users.fullname
FROM users
()
{stop}>>> for row in result:
...     print(row)
(u'jack', u'Jack Jones')
(u'wendy', u'Wendy Williams')
```

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:

```
{sql}>>> for row in conn.execute(select([users, addresses])):
...     print(row)
SELECT users.id, users.name, users.fullname, addresses.id, addresses.user_id, addresses.email_
↪address
FROM users, addresses
()
{stop}(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')
```

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. We do that using `Select.where()`:

```
>>> s = select([users, addresses]).where(users.c.id == addresses.c.user_id)
{sql}>>> for row in conn.execute(s):
...     print(row)
SELECT users.id, users.name, users.fullname, addresses.id,
   addresses.user_id, addresses.email_address
FROM users, addresses
WHERE users.id = addresses.user_id
()
{stop}(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.elements.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 `ColumnElement`.

3.1.8 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 the resulting `ColumnElement`; 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://'))) # doctest: +SKIP
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 `Operators.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`.

When using `Operators.op()`, the return type of the expression may be important, especialy when the operator is used in an expression that will be sent as a result column. For this case, be sure to make the type explicit, if not what's normally expected, using `type_coerce()`:

```
from sqlalchemy import type_coerce
expr = type_coerce(somecolumn.op('-%>')('foo'), MySpecialType())
stmt = select([expr])
```

For boolean operators, use the `Operators.bool_op()` method, which will ensure that the return type of the expression is handled as boolean:

```
somecolumn.bool_op('-->')('some value')
```

New in version 1.2.0b3: Added the `Operators.bool_op()` method.

Operator Customization

While `Operators.op()` is handy to get at a custom operator in a hurry, the Core supports fundamental customization and extension of the operator system at the type level. The behavior of existing operators can be modified on a per-type basis, and new operations can be defined which become available for all column expressions that are part of that particular type. See the section types_operators for a description.

3.1.9 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@yahoo.com'
...         ),
...         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@yahoo.com')
...     ) \
...     & ~(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')]).\
...        where(
...           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')
...                )
...           )
...        )
>>> 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.

A shortcut to using `and_()` is to chain together multiple `where()` clauses. The above can also be written as:

```
>>> s = select([(users.c.fullname +
...                ", " + addresses.c.email_address).
...                label('title')]).\
...        where(users.c.id == addresses.c.user_id).\
...        where(users.c.name.between('m', 'z')).\
...        where(
...               or_(
...                  addresses.c.email_address.like('%@aol.com'),
...                  addresses.c.email_address.like('%@msn.com')
...               )
...        )
>>> 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',)]
```

The way that we can build up a `select()` construct through successive method calls is called method chaining.

3.1.10 Using Textual SQL

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, for those cases when the SQL is already known and there isn't a strong need for the statement to support dynamic features. The `text()` construct is used to compose a textual statement that is passed to the database mostly unchanged. Below, we create a `text()` object and execute it:

```
>>> 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)")
{sql}>>> 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')
{stop}[(u'Wendy Williams, wendy@aol.com',)]
```

Above, we can see that bound parameters are specified in `text()` using the named colon format; this format is consistent regardless of database backend. To send values in for the parameters, we passed them into the `execute()` method as additional arguments.

Specifying Bound Parameter Behaviors

The `text()` construct supports pre-established bound values using the `TextClause.bindparams()` method:

```
stmt = text("SELECT * FROM users WHERE users.name BETWEEN :x AND :y")
stmt = stmt.bindparams(x="m", y="z")
```

The parameters can also be explicitly typed:

```
stmt = stmt.bindparams(bindparam("x", String), bindparam("y", String))
result = conn.execute(stmt, {"x": "m", "y": "z"})
```

Typing for bound parameters is necessary when the type requires Python-side or special SQL-side processing provided by the datatype.

See also:

`TextClause.bindparams()` - full method description

Specifying Result-Column Behaviors

We may also specify information about the result columns using the `TextClause.columns()` method; this method can be used to specify the return types, based on name:

```
stmt = stmt.columns(id=Integer, name=String)
```

or it can be passed full column expressions positionally, either typed or untyped. In this case it's a good idea to list out the columns explicitly within our textual SQL, since the correlation of our column expressions to the SQL will be done positionally:

```
stmt = text("SELECT id, name FROM users")
stmt = stmt.columns(users.c.id, users.c.name)
```

When we call the `TextClause.columns()` method, we get back a `TextAsFrom` object that supports the full suite of `TextAsFrom.c` and other "selectable" operations:

```
j = stmt.join(addresses, stmt.c.id == addresses.c.user_id)

new_stmt = select([stmt.c.id, addresses.c.id]).\
    select_from(j).where(stmt.c.name == 'x')
```

The positional form of `TextClause.columns()` is particularly useful when relating textual SQL to existing Core or ORM models, because we can use column expressions directly without worrying about name conflicts or other issues with the result column names in the textual SQL:

```
>>> stmt = text("SELECT users.id, addresses.id, users.id, "
...     "users.name, addresses.email_address AS email "
...     "FROM users JOIN addresses ON users.id=addresses.user_id "
...     "WHERE users.id = 1").columns(
...         users.c.id,
...         addresses.c.id,
...         addresses.c.user_id,
...         users.c.name,
...         addresses.c.email_address
...     )
{sql}>>> result = conn.execute(stmt)
SELECT users.id, addresses.id, users.id, users.name,
    addresses.email_address AS email
FROM users JOIN addresses ON users.id=addresses.user_id WHERE users.id = 1
()
{stop}
```

Above, there's three columns in the result that are named "id", but since we've associated these with column expressions positionally, the names aren't an issue when the result-columns are fetched using the actual column object as a key. Fetching the `email_address` column would be:

```
>>> row = result.fetchone()
>>> row[addresses.c.email_address]
'jack@yahoo.com'
```

If on the other hand we used a string column key, the usual rules of name- based matching still apply, and we'd get an ambiguous column error for the `id` value:

```
>>> row["id"]
Traceback (most recent call last):
...
InvalidRequestError: Ambiguous column name 'id' in result set column descriptions
```

It's important to note that while accessing columns from a result set using `Column` objects may seem unusual, it is in fact the only system used by the ORM, which occurs transparently beneath the facade of the `Query` object; in this way, the `TextClause.columns()` method is typically very applicable to textual

statements to be used in an ORM context. The example at orm_tutorial_literal_sql illustrates a simple usage.

New in version 1.1: The `TextClause.columns()` method now accepts column expressions which will be matched positionally to a plain text SQL result set, eliminating the need for column names to match or even be unique in the SQL statement when matching table metadata or ORM models to textual SQL.

See also:

`TextClause.columns()` - full method description

orm_tutorial_literal_sql - integrating ORM-level queries with `text()`

Using text() fragments inside bigger statements

`text()` can also be used to produce fragments of SQL that can be freely within a `select()` object, which accepts `text()` objects as an argument for most of its builder functions. Below, we combine the usage of `text()` within a `select()` object. The `select()` construct provides the "geometry" of the statement, and the `text()` construct provides the textual content within this form. We can build a statement without the need to refer to any pre-established `Table` metadata:

```
>>> s = select([
...         text("users.fullname || ', ' || addresses.email_address AS title")
...     ]).\
...         where(
...             and_(
...                 text("users.id = addresses.user_id"),
...                 text("users.name BETWEEN 'm' AND 'z'"),
...                 text(
...                     "(addresses.email_address LIKE :x "
...                     "OR addresses.email_address LIKE :y)")
...             )
...         ).select_from(text('users, addresses'))
{sql}>>> 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_address LIKE ? OR addresses.email_address LIKE ?)
('%@aol.com', '%@msn.com')
{stop}[(u'Wendy Williams, wendy@aol.com',)]
```

Changed in version 1.0.0: The `select()` construct emits warnings when string SQL fragments are coerced to `text()`, and `text()` should be used explicitly. See migration_2992 for background.

Using More Specific Text with `table()`, `literal_column()`, and `column()`

We can move our level of structure back in the other direction too, by using `column()`, `literal_column()`, and `table()` for some of the key elements of our statement. Using these constructs, we can get some more expression capabilities than if we used `text()` directly, as they provide to the Core more information about how the strings they store are to be used, but still without the need to get into full `Table` based metadata. Below, we also specify the `String` datatype for two of the key `literal_column()` objects, so that the string-specific concatenation operator becomes available. We also use `literal_column()` in order to use table-qualified expressions, e.g. `users.fullname`, that will be rendered as is; using `column()` implies an individual column name that may be quoted:

```
>>> from sqlalchemy import select, and_, text, String
>>> from sqlalchemy.sql import table, literal_column
>>> s = select([
...     literal_column("users.fullname", String) +
...     ', ' +
...     literal_column("addresses.email_address").label("title")
```

```
... ]).\
...     where(
...         and_(
...             literal_column("users.id") == literal_column("addresses.user_id"),
...             text("users.name BETWEEN 'm' AND 'z'"),
...             text(
...                 "(addresses.email_address LIKE :x OR "
...                 "addresses.email_address LIKE :y)")
...         )
...     ).select_from(table('users')).select_from(table('addresses'))

{sql}>>> conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()
SELECT users.fullname || ? || addresses.email_address AS anon_1
FROM users, addresses
WHERE users.id = addresses.user_id
AND users.name BETWEEN 'm' AND 'z'
AND (addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
(', ', '%@aol.com', '%@msn.com')
{stop}[(u'Wendy Williams, wendy@aol.com',)]
```

Ordering or Grouping by a Label

One place where we sometimes want to use a string as a shortcut is when our statement has some labeled column element that we want to refer to in a place such as the "ORDER BY" or "GROUP BY" clause; other candidates include fields within an "OVER" or "DISTINCT" clause. If we have such a label in our `select()` construct, we can refer to it directly by passing the string straight into `select.order_by()` or `select.group_by()`, among others. This will refer to the named label and also prevent the expression from being rendered twice:

```
>>> from sqlalchemy import func
>>> stmt = select([
...          addresses.c.user_id,
...          func.count(addresses.c.id).label('num_addresses')]).\
...          order_by("num_addresses")

{sql}>>> conn.execute(stmt).fetchall()
SELECT addresses.user_id, count(addresses.id) AS num_addresses
FROM addresses ORDER BY num_addresses
()
{stop}[(2, 4)]
```

We can use modifiers like `asc()` or `desc()` by passing the string name:

```
>>> from sqlalchemy import func, desc
>>> stmt = select([
...          addresses.c.user_id,
...          func.count(addresses.c.id).label('num_addresses')]).\
...          order_by(desc("num_addresses"))

{sql}>>> conn.execute(stmt).fetchall()
SELECT addresses.user_id, count(addresses.id) AS num_addresses
FROM addresses ORDER BY num_addresses DESC
()
{stop}[(2, 4)]
```

Note that the string feature here is very much tailored to when we have already used the `label()` method to create a specifically-named label. In other cases, we always want to refer to the `ColumnElement` object directly so that the expression system can make the most effective choices for rendering. Below, we illustrate how using the `ColumnElement` eliminates ambiguity when we want to order by a column name that appears more than once:

```
>>> u1a, u1b = users.alias(), users.alias()
>>> stmt = select([u1a, u1b]).\
...             where(u1a.c.name > u1b.c.name).\
...             order_by(u1a.c.name)  # using "name" here would be ambiguous

{sql}>>> conn.execute(stmt).fetchall()
SELECT users_1.id, users_1.name, users_1.fullname, users_2.id,
users_2.name, users_2.fullname
FROM users AS users_1, users AS users_2
WHERE users_1.name > users_2.name ORDER BY users_1.name
()
{stop}[(2, u'wendy', u'Wendy Williams', 1, u'jack', u'Jack Jones')]
```

3.1.11 Using Aliases

The alias in SQL corresponds to a "renamed" version of a table or SELECT statement, which occurs anytime you say "SELECT .. FROM sometable AS someothername". The `AS` creates a new name for the table. Aliases are a key construct as they allow any table or subquery to be referenced by a unique name. In the case of a table, this allows the same table to be named in the FROM clause multiple times. In the case of a SELECT statement, it provides a parent name for the columns represented by the statement, allowing them to be referenced relative to this name.

In SQLAlchemy, any `Table`, `select()` construct, or other selectable can be turned into an alias using the `FromClause.alias()` method, which produces a `Alias` construct. As an example, suppose we know that our user `jack` has two particular email addresses. How can we locate jack based on the combination of those two addresses? To accomplish this, we'd use a join to the `addresses` table, once for each address. We create two `Alias` constructs against `addresses`, and then use them both within a `select()` construct:

```
>>> a1 = addresses.alias()
>>> a2 = addresses.alias()
>>> s = select([users]).\
...         where(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'
...         ))
{sql}>>> 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.email_address = ?
    AND addresses_2.email_address = ?
('jack@msn.com', 'jack@yahoo.com')
{stop}[(1, u'jack', u'Jack Jones')]
```

Note that the `Alias` construct generated the names `addresses_1` and `addresses_2` in the final SQL result. The generation of these names is determined by the position of the construct within the statement. If we created a query using only the second `a2` alias, the name would come out as `addresses_1`. The generation of the names is also *deterministic*, meaning the same SQLAlchemy statement construct will produce the identical SQL string each time it is rendered for a particular dialect.

Since on the outside, we refer to the alias using the `Alias` construct itself, we don't need to be concerned about the generated name. However, for the purposes of debugging, it can be specified by passing a string name to the `FromClause.alias()` method:

```
>>> a1 = addresses.alias('a1')
```

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]).where(users.c.id == a1.c.id)
{sql}>>> 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.email_address = ?
        AND addresses_2.email_address = ?) AS anon_1
WHERE users.id = anon_1.id
('jack@msn.com', 'jack@yahoo.com')
{stop}[(u'jack',)]
```

3.1.12 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 use of the `select_from()` method:

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

The `outerjoin()` method creates `LEFT OUTER JOIN` constructs, and is used in the same way as `join()`:

```
>>> s = select([users.c.fullname]).select_from(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 ;).

See also:

`expression.join()`

`expression.outerjoin()`

`Join`

3.1.13 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 `bindparam()` construct is used to produce a bound parameter with a given name. While SQLAlchemy always refers to bound parameters by name on the API side, the database dialect converts to the appropriate named or positional style at execution time, as here where it converts to positional for SQLite:

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

Another important aspect of `bindparam()` is that it 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("'%'")))
{sql}>>> conn.execute(s, username='wendy').fetchall()
SELECT users.id, users.name, users.fullname
FROM users
WHERE users.name LIKE ? || '%'
('wendy',)
{stop}[(2, u'wendy', u'Wendy Williams')]
```

bindparam() constructs 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]).\
...     where(
...        or_(
...          users.c.name.like(
...                 bindparam('name', type_=String) + text("'%'")),
...          addresses.c.email_address.like(
...                 bindparam('name', type_=String) + text("'@%'"))
...        )
...     ).\
...     select_from(users.outerjoin(addresses)).\
...     order_by(addresses.c.id)
{sql}>>> conn.execute(s, name='jack').fetchall()
SELECT users.id, users.name, users.fullname, addresses.id,
    addresses.user_id, addresses.email_address
FROM users LEFT OUTER JOIN addresses ON users.id = addresses.user_id
WHERE users.name LIKE ? || '%' OR addresses.email_address LIKE ? || '@%'
ORDER BY addresses.id
('jack', 'jack')
{stop}[(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com'), (1, u'jack', u'Jack Jones', 2, 1,␣
↪u'jack@msn.com')]
```

See also:

bindparam()

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(:concat_1, :concat_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:

```
>>> conn.execute(
...     select([
```

```
...             func.max(addresses.c.email_address, type_=String).
...                 label('maxemail')
...           ])
...     ).scalar()
{opensql}SELECT max(addresses.email_address) AS maxemail
FROM addresses
()
{stop}u'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')]).\
...         select_from(
...              func.calculate(
...                     bindparam('x'),
...                     bindparam('y')
...                 )
...              )
>>> calc = calculate.alias()
>>> print(select([users]).where(users.c.id > calc.c.z))
SELECT users.id, users.name, users.fullname
FROM users, (SELECT q, z, r
FROM calculate(:x, :y)) AS anon_1
WHERE users.id > anon_1.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:

```
>>> calc1 = calculate.alias('c1').unique_params(x=17, y=45)
>>> calc2 = calculate.alias('c2').unique_params(x=5, y=12)
>>> s = select([users]).\
...         where(users.c.id.between(calc1.c.z, calc2.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 # doctest: +SKIP
{u'x_2': 5, u'y_2': 12, u'y_1': 45, u'x_1': 17}
```

See also:

`func`

Window Functions

Any `FunctionElement`, including functions generated by `func`, can be turned into a "window function", that is an OVER clause, using the `FunctionElement.over()` method:

```
>>> s = select([
...         users.c.id,
...         func.row_number().over(order_by=users.c.name)
...     ])
>>> print(s)
```

```
SELECT users.id, row_number() OVER (ORDER BY users.name) AS anon_1
FROM users
```

FunctionElement.over() also supports range specification using either the expression.over.rows or expression.over.range parameters:

```
>>> s = select([
...         users.c.id,
...         func.row_number().over(
...                 order_by=users.c.name,
...                 rows=(-2, None))
...     ])
>>> print(s)
SELECT users.id, row_number() OVER
(ORDER BY users.name ROWS BETWEEN :param_1 PRECEDING AND UNBOUNDED FOLLOWING) AS anon_1
FROM users
```

expression.over.rows and expression.over.range each accept a two-tuple which contains a combination of negative and positive integers for ranges, zero to indicate "CURRENT ROW" and None to indicate "UNBOUNDED". See the examples at over() for more detail.

New in version 1.1: support for "rows" and "range" specification for window functions

See also:

over()

FunctionElement.over()

Unions and Other Set Operations

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

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

{sql}>>> 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.email_address
FROM addresses
WHERE addresses.email_address LIKE ? ORDER BY addresses.email_address
('foo@bar.com', '%@yahoo.com')
{stop}[(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().
...             where(addresses.c.email_address.like('%@%.com')),
...     addresses.select().
...             where(addresses.c.email_address.like('%@msn.com'))
```

```
... )

{sql}>>> 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, addresses.email_address
FROM addresses
WHERE addresses.email_address LIKE ?
('%@%.com', '%@msn.com')
{stop}[(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().
...             where(addresses.c.email_address.like('%@yahoo.com')),
...         addresses.select().
...             where(addresses.c.email_address.like('%@msn.com'))
...     ).alias().select(),   # apply subquery here
...     addresses.select(addresses.c.email_address.like('%@msn.com'))
... )
{sql}>>> 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')
{stop}[(1, 1, u'jack@yahoo.com')]
```

See also:

`union()`

`union_all()`

`intersect()`

`intersect_all()`

`except_()`

`except_all()`

Scalar Selects

A scalar select is a SELECT that returns exactly one row and one column. It can then be used as a column expression. A scalar select is often a correlated subquery, which relies upon the enclosing SELECT statement in order to acquire at least one of its FROM clauses.

The `select()` construct can be modified to act as a column expression by calling either the `as_scalar()` or `label()` method:

```
>>> stmt = select([func.count(addresses.c.id)]).\
...             where(users.c.id == addresses.c.user_id).\
...             as_scalar()
```

The above construct is now a `ScalarSelect` object, and is no longer part of the `FromClause` hierarchy; it instead is within the `ColumnElement` family of expression constructs. We can place this construct the same as any other column within another `select()`:

```
>>> conn.execute(select([users.c.name, stmt])).fetchall()
{opensql}SELECT users.name, (SELECT count(addresses.id) AS count_1
FROM addresses
WHERE users.id = addresses.user_id) AS anon_1
FROM users
()
{stop}[(u'jack', 2), (u'wendy', 2)]
```

To apply a non-anonymous column name to our scalar select, we create it using `SelectBase.label()` instead:

```
>>> stmt = select([func.count(addresses.c.id)]).\
...             where(users.c.id == addresses.c.user_id).\
...             label("address_count")
>>> conn.execute(select([users.c.name, stmt])).fetchall()
{opensql}SELECT users.name, (SELECT count(addresses.id) AS count_1
FROM addresses
WHERE users.id = addresses.user_id) AS address_count
FROM users
()
{stop}[(u'jack', 2), (u'wendy', 2)]
```

See also:

`Select.as_scalar()`

`Select.label()`

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 correlates embedded FROM objects to that of an enclosing query, if present, and if the inner SELECT statement would still have at least one FROM clause of its own. For example:

```
>>> stmt = select([addresses.c.user_id]).\
...             where(addresses.c.user_id == users.c.id).\
...             where(addresses.c.email_address == 'jack@yahoo.com')
>>> enclosing_stmt = select([users.c.name]).where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
{opensql}SELECT users.name
FROM users
WHERE users.id = (SELECT addresses.user_id
    FROM addresses
    WHERE addresses.user_id = users.id
```

```
    AND addresses.email_address = ?)
('jack@yahoo.com',)
{stop}[(u'jack',)]
```

Auto-correlation will usually do what's expected, however it can also be controlled. For example, if we wanted a statement to correlate only to the `addresses` table but not the `users` table, even if both were present in the enclosing SELECT, we use the `correlate()` method to specify those FROM clauses that may be correlated:

```
>>> stmt = select([users.c.id]).\
...             where(users.c.id == addresses.c.user_id).\
...             where(users.c.name == 'jack').\
...             correlate(addresses)
>>> enclosing_stmt = select(
...         [users.c.name, addresses.c.email_address]).\
...     select_from(users.join(addresses)).\
...     where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
{opensql}SELECT users.name, addresses.email_address
 FROM users JOIN addresses ON users.id = addresses.user_id
 WHERE users.id = (SELECT users.id
 FROM users
 WHERE users.id = addresses.user_id AND users.name = ?)
 ('jack',)
 {stop}[(u'jack', u'jack@yahoo.com'), (u'jack', u'jack@msn.com')]
```

To entirely disable a statement from correlating, we can pass `None` as the argument:

```
>>> stmt = select([users.c.id]).\
...             where(users.c.name == 'wendy').\
...             correlate(None)
>>> enclosing_stmt = select([users.c.name]).\
...     where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
{opensql}SELECT users.name
 FROM users
 WHERE users.id = (SELECT users.id
  FROM users
  WHERE users.name = ?)
('wendy',)
{stop}[(u'wendy',)]
```

We can also control correlation via exclusion, using the `Select.correlate_except()` method. Such as, we can write our SELECT for the `users` table by telling it to correlate all FROM clauses except for `users`:

```
>>> stmt = select([users.c.id]).\
...             where(users.c.id == addresses.c.user_id).\
...             where(users.c.name == 'jack').\
...             correlate_except(users)
>>> enclosing_stmt = select(
...         [users.c.name, addresses.c.email_address]).\
...     select_from(users.join(addresses)).\
...     where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
{opensql}SELECT users.name, addresses.email_address
 FROM users JOIN addresses ON users.id = addresses.user_id
 WHERE users.id = (SELECT users.id
 FROM users
 WHERE users.id = addresses.user_id AND users.name = ?)
 ('jack',)
 {stop}[(u'jack', u'jack@yahoo.com'), (u'jack', u'jack@msn.com')]
```

LATERAL correlation

LATERAL correlation is a special sub-category of SQL correlation which allows a selectable unit to refer to another selectable unit within a single FROM clause. This is an extremely special use case which, while part of the SQL standard, is only known to be supported by recent versions of PostgreSQL.

Normally, if a SELECT statement refers to `table1 JOIN (some SELECT) AS subquery` in its FROM clause, the subquery on the right side may not refer to the "table1" expression from the left side; correlation may only refer to a table that is part of another SELECT that entirely encloses this SELECT. The LATERAL keyword allows us to turn this behavior around, allowing an expression such as:

```
SELECT people.people_id, people.age, people.name
FROM people JOIN LATERAL (SELECT books.book_id AS book_id
FROM books WHERE books.owner_id = people.people_id)
AS book_subq ON true
```

Where above, the right side of the JOIN contains a subquery that refers not just to the "books" table but also the "people" table, correlating to the left side of the JOIN. SQLAlchemy Core supports a statement like the above using the `Select.lateral()` method as follows:

```
>>> from sqlalchemy import table, column, select, true
>>> people = table('people', column('people_id'), column('age'), column('name'))
>>> books = table('books', column('book_id'), column('owner_id'))
>>> subq = select([books.c.book_id]).\
...       where(books.c.owner_id == people.c.people_id).lateral("book_subq")
>>> print(select([people]).select_from(people.join(subq, true())))
SELECT people.people_id, people.age, people.name
FROM people JOIN LATERAL (SELECT books.book_id AS book_id
FROM books WHERE books.owner_id = people.people_id)
AS book_subq ON true
```

Above, we can see that the `Select.lateral()` method acts a lot like the `Select.alias()` method, including that we can specify an optional name. However the construct is the `Lateral` construct instead of an `Alias` which provides for the LATERAL keyword as well as special instructions to allow correlation from inside the FROM clause of the enclosing statement.

The `Select.lateral()` method interacts normally with the `Select.correlate()` and `Select.correlate_except()` methods, except that the correlation rules also apply to any other tables present in the enclosing statement's FROM clause. Correlation is "automatic" to these tables by default, is explicit if the table is specified to `Select.correlate()`, and is explicit to all tables except those specified to `Select.correlate_except()`.

New in version 1.1: Support for the LATERAL keyword and lateral correlation.

See also:

`Lateral`

`Select.lateral()`

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

Ordering is done by passing column expressions to the `order_by()` method:

```
>>> stmt = select([users.c.name]).order_by(users.c.name)
>>> conn.execute(stmt).fetchall()
{opensql}SELECT users.name
FROM users ORDER BY users.name
()
{stop}[(u'jack',), (u'wendy',)]
```

Ascending or descending can be controlled using the `asc()` and `desc()` modifiers:

```
>>> stmt = select([users.c.name]).order_by(users.c.name.desc())
>>> conn.execute(stmt).fetchall()
{opensql}SELECT users.name
FROM users ORDER BY users.name DESC
()
{stop}[(u'wendy',), (u'jack',)]
```

Grouping refers to the GROUP BY clause, and is usually used in conjunction with aggregate functions to establish groups of rows to be aggregated. This is provided via the `group_by()` method:

```
>>> stmt = select([users.c.name, func.count(addresses.c.id)]).\
...             select_from(users.join(addresses)).\
...             group_by(users.c.name)
>>> conn.execute(stmt).fetchall()
{opensql}SELECT users.name, count(addresses.id) AS count_1
FROM users JOIN addresses
    ON users.id = addresses.user_id
GROUP BY users.name
()
{stop}[(u'jack', 2), (u'wendy', 2)]
```

HAVING can be used to filter results on an aggregate value, after GROUP BY has been applied. It's available here via the `having()` method:

```
>>> stmt = select([users.c.name, func.count(addresses.c.id)]).\
...             select_from(users.join(addresses)).\
...             group_by(users.c.name).\
...             having(func.length(users.c.name) > 4)
>>> conn.execute(stmt).fetchall()
{opensql}SELECT users.name, count(addresses.id) AS count_1
FROM users JOIN addresses
    ON users.id = addresses.user_id
GROUP BY users.name
HAVING length(users.name) > ?
(4,)
{stop}[(u'wendy', 2)]
```

A common system of dealing with duplicates in composed SELECT statements is the DISTINCT modifier. A simple DISTINCT clause can be added using the `Select.distinct()` method:

```
>>> stmt = select([users.c.name]).\
...             where(addresses.c.email_address.
...                     contains(users.c.name)).\
...             distinct()
>>> conn.execute(stmt).fetchall()
{opensql}SELECT DISTINCT users.name
FROM users, addresses
WHERE (addresses.email_address LIKE '%' || users.name || '%')
()
{stop}[(u'jack',), (u'wendy',)]
```

Most database backends support a system of limiting how many rows are returned, and the majority also feature a means of starting to return rows after a given "offset". While common backends like PostgreSQL, MySQL and SQLite support LIMIT and OFFSET keywords, other backends need to refer to more esoteric features such as "window functions" and row ids to achieve the same effect. The `limit()` and `offset()` methods provide an easy abstraction into the current backend's methodology:

```
>>> stmt = select([users.c.name, addresses.c.email_address]).\
...             select_from(users.join(addresses)).\
...             limit(1).offset(1)
>>> conn.execute(stmt).fetchall()
{opensql}SELECT users.name, addresses.email_address
```

```
FROM users JOIN addresses ON users.id = addresses.user_id
 LIMIT ? OFFSET ?
(1, 1)
{stop}[(u'jack', u'jack@msn.com')]
```

3.1.14 Inserts, Updates and Deletes

We've seen `insert()` demonstrated earlier in this tutorial. Where `insert()` produces INSERT, the `update()` method produces UPDATE. Both of these constructs feature a method called `values()` which specifies the VALUES or SET clause of the statement.

The `values()` method accommodates any column expression as a value:

```
>>> stmt = users.update().\
...             values(fullname="Fullname: " + users.c.name)
>>> conn.execute(stmt)
{opensql}UPDATE users SET fullname=(? || users.name)
('Fullname: ',)
COMMIT
{stop}<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

When using `insert()` or `update()` in an "execute many" context, we may also want to specify named bound parameters which we can refer to in the argument list. The two constructs will automatically generate bound placeholders for any column names passed in the dictionaries sent to `execute()` at execution time. However, if we wish to use explicitly targeted named parameters with composed expressions, we need to use the `bindparam()` construct. When using `bindparam()` with `insert()` or `update()`, the names of the table's columns themselves are reserved for the "automatic" generation of bind names. We can combine the usage of implicitly available bind names and explicitly named parameters as in the example below:

```
>>> stmt = users.insert().\
...          values(name=bindparam('_name') + " .. name")
>>> conn.execute(stmt, [
...        {'id':4, '_name':'name1'},
...        {'id':5, '_name':'name2'},
...        {'id':6, '_name':'name3'},
...     ])
{opensql}INSERT INTO users (id, name) VALUES (?, (? || ?))
((4, 'name1', ' .. name'), (5, 'name2', ' .. name'), (6, 'name3', ' .. name'))
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

An UPDATE statement is emitted using the `update()` construct. This works much like an INSERT, except there is an additional WHERE clause that can be specified:

```
>>> stmt = users.update().\
...             where(users.c.name == 'jack').\
...             values(name='ed')

>>> conn.execute(stmt)
{opensql}UPDATE users SET name=? WHERE users.name = ?
('ed', 'jack')
COMMIT
{stop}<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

When using `update()` in an "executemany" context, we may wish to also use explicitly named bound parameters in the WHERE clause. Again, `bindparam()` is the construct used to achieve this:

```
>>> stmt = users.update().\
...             where(users.c.name == bindparam('oldname')).\
```

```
...             values(name=bindparam('newname'))
>>> conn.execute(stmt, [
...     {'oldname':'jack', 'newname':'ed'},
...     {'oldname':'wendy', 'newname':'mary'},
...     {'oldname':'jim', 'newname':'jake'},
...     ])
{opensql}UPDATE users SET name=? WHERE users.name = ?
(('ed', 'jack'), ('mary', 'wendy'), ('jake', 'jim'))
COMMIT
{stop}<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

Correlated Updates

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

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

Multiple Table Updates

New in version 0.7.4.

The PostgreSQL, Microsoft SQL Server, and MySQL backends all support UPDATE statements that refer to multiple tables. For PG and MSSQL, this is the "UPDATE FROM" syntax, which updates one table at a time, but can reference additional tables in an additional "FROM" clause that can then be referenced in the WHERE clause directly. On MySQL, multiple tables can be embedded into a single UPDATE statement separated by a comma. The SQLAlchemy `update()` construct supports both of these modes implicitly, by specifying multiple tables in the WHERE clause:

```
stmt = users.update().\
        values(name='ed wood').\
        where(users.c.id == addresses.c.id).\
        where(addresses.c.email_address.startswith('ed%'))
conn.execute(stmt)
```

The resulting SQL from the above statement would render as:

```
UPDATE users SET name=:name FROM addresses
WHERE users.id = addresses.id AND
addresses.email_address LIKE :email_address_1 || '%'
```

When using MySQL, columns from each table can be assigned to in the SET clause directly, using the dictionary form passed to `Update.values()`:

```
stmt = users.update().\
        values({
            users.c.name:'ed wood',
            addresses.c.email_address:'ed.wood@foo.com'
        }).\
```

```
        where(users.c.id == addresses.c.id).\
        where(addresses.c.email_address.startswith('ed%'))
```

The tables are referenced explicitly in the SET clause:

```
UPDATE users, addresses SET addresses.email_address=%s,
        users.name=%s WHERE users.id = addresses.id
        AND addresses.email_address LIKE concat(%s, '%')
```

When the construct is used on a non-supporting database, the compiler will raise `NotImplementedError`. For convenience, when a statement is printed as a string without specification of a dialect, the "string SQL" compiler will be invoked which provides a non-working SQL representation of the construct.

Parameter-Ordered Updates

The default behavior of the `update()` construct when rendering the SET clauses is to render them using the column ordering given in the originating `Table` object. This is an important behavior, since it means that the rendering of a particular UPDATE statement with particular columns will be rendered the same each time, which has an impact on query caching systems that rely on the form of the statement, either client side or server side. Since the parameters themselves are passed to the `Update.values()` method as Python dictionary keys, there is no other fixed ordering available.

However in some cases, the order of parameters rendered in the SET clause of an UPDATE statement can be significant. The main example of this is when using MySQL and providing updates to column values based on that of other column values. The end result of the following statement:

```
UPDATE some_table SET x = y + 10, y = 20
```

Will have a different result than:

```
UPDATE some_table SET y = 20, x = y + 10
```

This because on MySQL, the individual SET clauses are fully evaluated on a per-value basis, as opposed to on a per-row basis, and as each SET clause is evaluated, the values embedded in the row are changing.

To suit this specific use case, the `preserve_parameter_order` flag may be used. When using this flag, we supply a **Python list of 2-tuples** as the argument to the `Update.values()` method:

```
stmt = some_table.update(preserve_parameter_order=True).\
    values([(some_table.c.y, 20), (some_table.c.x, some_table.c.y + 10)])
```

The list of 2-tuples is essentially the same structure as a Python dictionary except it is ordered. Using the above form, we are assured that the "y" column's SET clause will render first, then the "x" column's SET clause.

New in version 1.0.10: Added support for explicit ordering of UPDATE parameters using the `preserve_parameter_order` flag.

Deletes

Finally, a delete. This is accomplished easily enough using the `delete()` construct:

```
>>> conn.execute(addresses.delete())
{opensql}DELETE FROM addresses
()
COMMIT
{stop}<sqlalchemy.engine.result.ResultProxy object at 0x...>

>>> conn.execute(users.delete().where(users.c.name > 'm'))
{opensql}DELETE FROM users WHERE users.name > ?
```

```
('m',)
COMMIT
{stop}<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

Multiple Table Deletes

New in version 1.2.

The PostgreSQL, Microsoft SQL Server, and MySQL backends all support DELETE statements that refer to multiple tables within the WHERE criteria. For PG and MySQL, this is the "DELETE USING" syntax, and for SQL Server, it's a "DELETE FROM" that refers to more than one table. The SQLAlchemy `delete()` construct supports both of these modes implicitly, by specifying multiple tables in the WHERE clause:

```
stmt = users.delete().\
        where(users.c.id == addresses.c.id).\
        where(addresses.c.email_address.startswith('ed%'))
conn.execute(stmt)
```

On a Postgresql backend, the resulting SQL from the above statement would render as:

```
DELETE FROM users USING addresses
WHERE users.id = addresses.id
AND (addresses.email_address LIKE %(email_address_1)s || '%%')
```

When the construct is used on a non-supporting database, the compiler will raise `NotImplementedError`. For convenience, when a statement is printed as a string without specification of a dialect, the "string SQL" compiler will be invoked which provides a non-working SQL representation of the construct.

Matched Row Counts

Both of `update()` and `delete()` are associated with *matched row counts*. This is a number indicating the number of rows that were matched by the WHERE clause. Note that by "matched", this includes rows where no UPDATE actually took place. The value is available as `rowcount`:

```
>>> result = conn.execute(users.delete())
{opensql}DELETE FROM users
()
COMMIT
{stop}>>> result.rowcount
1
```

3.1.15 Further Reference

Expression Language Reference: expression_api_toplevel

Database Metadata Reference: metadata_toplevel

Engine Reference: *Engine Configuration*

Connection Reference: connections_toplevel

Types Reference: types_toplevel

3.2 SQL Statements and Expressions API

This section presents the API reference for the SQL Expression Language. For a full introduction to its usage, see sqlexpression_toplevel.

3.2.1 Column Elements and Expressions

The expression API consists of a series of classes that each represent a specific lexical element within a SQL string. Composed together into a larger structure, they form a statement construct that may be *compiled* into a string representation that can be passed to a database. The classes are organized into a hierarchy that begins at the basemost ClauseElement class. Key subclasses include ColumnElement, which represents the role of any column-based expression in a SQL statement, such as in the columns clause, WHERE clause, and ORDER BY clause, and FromClause, which represents the role of a token that is placed in the FROM clause of a SELECT statement.

`sqlalchemy.sql.expression.all_(`*expr*`)`
Produce an ALL expression.

This may apply to an array type for some dialects (e.g. postgresql), or to a subquery for others (e.g. mysql). e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == all_(mytable.c.somearray)

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == all_(select([table.c.value]))
```

New in version 1.1.

See also:

`expression.any_()`

`sqlalchemy.sql.expression.and_(`**clauses*`)`
Produce a conjunction of expressions joined by `AND`.

E.g.:

```
from sqlalchemy import and_

stmt = select([users_table]).where(
                and_(
                    users_table.c.name == 'wendy',
                    users_table.c.enrolled == True
                )
            )
```

The `and_()` conjunction is also available using the Python `&` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') &
                (users_table.c.enrolled == True)
            )
```

The `and_()` operation is also implicit in some cases; the `Select.where()` method for example can be invoked multiple times against a statement, which will have the effect of each clause being combined using `and_()`:

```
stmt = select([users_table]).                        where(users_table.c.name == 'wendy').
↪                        where(users_table.c.enrolled == True)
```

See also:

`or_()`

sqlalchemy.sql.expression.**any_**(*expr*)
Produce an ANY expression.

This may apply to an array type for some dialects (e.g. postgresql), or to a subquery for others (e.g. mysql). e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == any_(mytable.c.somearray)

# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == any_(select([table.c.value]))
```

New in version 1.1.

See also:

`expression.all_()`

sqlalchemy.sql.expression.**asc**(*column*)
Produce an ascending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import asc
stmt = select([users_table]).order_by(asc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name ASC
```

The `asc()` function is a standalone version of the `ColumnElement.asc()` method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.asc())
```

Parameters `column` – A `ColumnElement` (e.g. scalar SQL expression) with which to apply the `asc()` operation.

See also:

`desc()`

`nullsfirst()`

`nullslast()`

`Select.order_by()`

sqlalchemy.sql.expression.**between**(*expr*, *lower_bound*, *upper_bound*, *symmetric=False*)
Produce a `BETWEEN` predicate clause.

E.g.:

```
from sqlalchemy import between
stmt = select([users_table]).where(between(users_table.c.id, 5, 7))
```

Would produce SQL resembling:

```
SELECT id, name FROM user WHERE id BETWEEN :id_1 AND :id_2
```

The `between()` function is a standalone version of the `ColumnElement.between()` method available on all SQL expressions, as in:

```
stmt = select([users_table]).where(users_table.c.id.between(5, 7))
```

All arguments passed to `between()`, including the left side column expression, are coerced from Python scalar values if a the value is not a `ColumnElement` subclass. For example, three fixed values can be compared as in:

```
print(between(5, 3, 7))
```

Which would produce:

```
:param_1 BETWEEN :param_2 AND :param_3
```

Parameters

- **expr** – a column expression, typically a `ColumnElement` instance or alternatively a Python scalar expression to be coerced into a column expression, serving as the left side of the `BETWEEN` expression.
- **lower_bound** – a column or Python scalar expression serving as the lower bound of the right side of the `BETWEEN` expression.
- **upper_bound** – a column or Python scalar expression serving as the upper bound of the right side of the `BETWEEN` expression.
- **symmetric** – if True, will render " BETWEEN SYMMETRIC ". Note that not all databases support this syntax.

  New in version 0.9.5.

See also:

`ColumnElement.between()`

sqlalchemy.sql.expression.**bindparam**(*key*, *value=symbol('NO_ARG')*, *type_=None*, *unique=False*, *required=symbol('NO_ARG')*, *quote=None*, *callable_=None*, *expanding=False*, *isoutparam=False*, *_compared_to_operator=None*, *_compared_to_type=None*)

Produce a "bound expression".

The return value is an instance of `BindParameter`; this is a `ColumnElement` subclass which represents a so-called "placeholder" value in a SQL expression, the value of which is supplied at the point at which the statement in executed against a database connection.

In SQLAlchemy, the `bindparam()` construct has the ability to carry along the actual value that will be ultimately used at expression time. In this way, it serves not just as a "placeholder" for eventual population, but also as a means of representing so-called "unsafe" values which should not be rendered directly in a SQL statement, but rather should be passed along to the DBAPI as values which need to be correctly escaped and potentially handled for type-safety.

When using `bindparam()` explicitly, the use case is typically one of traditional deferment of parameters; the `bindparam()` construct accepts a name which can then be referred to at execution time:

```
from sqlalchemy import bindparam

stmt = select([users_table]).\
            where(users_table.c.name == bindparam('username'))
```

The above statement, when rendered, will produce SQL similar to:

```
SELECT id, name FROM user WHERE name = :username
```

In order to populate the value of `:username` above, the value would typically be applied at execution time to a method like `Connection.execute()`:

```
result = connection.execute(stmt, username='wendy')
```

Explicit use of `bindparam()` is also common when producing UPDATE or DELETE statements that are to be invoked multiple times, where the WHERE criterion of the statement is to change on each invocation, such as:

```
stmt = (users_table.update().
        where(user_table.c.name == bindparam('username')).
        values(fullname=bindparam('fullname'))
        )

connection.execute(
    stmt, [{"username": "wendy", "fullname": "Wendy Smith"},
           {"username": "jack", "fullname": "Jack Jones"},
           ]
)
```

SQLAlchemy's Core expression system makes wide use of `bindparam()` in an implicit sense. It is typical that Python literal values passed to virtually all SQL expression functions are coerced into fixed `bindparam()` constructs. For example, given a comparison operation such as:

```
expr = users_table.c.name == 'Wendy'
```

The above expression will produce a `BinaryExpression` construct, where the left side is the `Column` object representing the `name` column, and the right side is a `BindParameter` representing the literal value:

```
print(repr(expr.right))
BindParameter('%(4327771088 name)s', 'Wendy', type_=String())
```

The expression above will render SQL such as:

```
user.name = :name_1
```

Where the `:name_1` parameter name is an anonymous name. The actual string `Wendy` is not in the rendered string, but is carried along where it is later used within statement execution. If we invoke a statement like the following:

```
stmt = select([users_table]).where(users_table.c.name == 'Wendy')
result = connection.execute(stmt)
```

We would see SQL logging output as:

```
SELECT "user".id, "user".name
FROM "user"
WHERE "user".name = %(name_1)s
{'name_1': 'Wendy'}
```

Above, we see that `Wendy` is passed as a parameter to the database, while the placeholder `:name_1` is rendered in the appropriate form for the target database, in this case the PostgreSQL database.

Similarly, `bindparam()` is invoked automatically when working with CRUD statements as far as the "VALUES" portion is concerned. The `insert()` construct produces an `INSERT` expression which will, at statement execution time, generate bound placeholders based on the arguments passed, as in:

```
stmt = users_table.insert()
result = connection.execute(stmt, name='Wendy')
```

The above will produce SQL output as:

```
INSERT INTO "user" (name) VALUES (%(name)s)
{'name': 'Wendy'}
```

The `Insert` construct, at compilation/execution time, rendered a single `bindparam()` mirroring the column name `name` as a result of the single `name` parameter we passed to the `Connection.execute()` method.

Parameters

- **key** – the key (e.g. the name) 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 `BindParameter` objects exist with the same key, or if its length is too long and truncation is required.
- **value** – Initial value for this bind param. Will be used at statement execution time as the value for this parameter passed to the DBAPI, if no other value is indicated to the statement execution method for this particular parameter name. Defaults to `None`.
- **callable_** – A callable function that takes the place of "value". The function will be called at statement execution time to determine the ultimate value. Used for scenarios where the actual bind value cannot be determined at the point at which the clause construct is created, but embedded bind values are still desirable.
- **type_** – A `TypeEngine` class or instance representing an optional datatype for this `bindparam()`. If not passed, a type may be determined automatically for the bind, based on the given value; for example, trivial Python types such as `str`, `int`, `bool` may result in the `String`, `Integer` or `Boolean` types being automatically selected.

  The type of a `bindparam()` is significant especially in that the type will apply pre-processing to the value before it is passed to the database. For example, a `bindparam()` which refers to a datetime value, and is specified as holding the `DateTime` type, may apply conversion needed to the value (such as stringification on SQLite) before passing the value to the database.
- **unique** – if True, the key name of this `BindParameter` will be modified if another `BindParameter` of the same name already has been located within the containing expression. This flag is used generally by the internals when producing so-called "anonymous" bound expressions, it isn't generally applicable to explicitly-named `bindparam()` constructs.
- **required** – If `True`, a value is required at execution time. If not passed, it defaults to `True` if neither `bindparam.value` or `bindparam.callable` were passed. If either of these parameters are present, then `bindparam.required` defaults to `False`.

  Changed in version 0.8: If the `required` flag is not specified, it will be set automatically to `True` or `False` depending on whether or not the `value` or `callable` parameters were specified.
- **quote** – True if this parameter name requires quoting and is not currently known as a SQLAlchemy reserved word; this currently only applies to the Oracle backend, where bound names must sometimes be quoted.
- **isoutparam** – if True, the parameter should be treated like a stored procedure "OUT" parameter. This applies to backends such as Oracle which support OUT parameters.
- **expanding** – if True, this parameter will be treated as an "expanding" parameter at execution time; the parameter value is expected to be a sequence, rather

than a scalar value, and the string SQL statement will be transformed on a per-execution basis to accomodate the sequence with a variable number of parameter slots passed to the DBAPI. This is to allow statement caching to be used in conjunction with an IN clause.

Note: The "expanding" feature does not support "executemany"- style parameter sets, nor does it support empty IN expressions.

Note: The "expanding" feature should be considered as **experimental** within the 1.2 series.

New in version 1.2.

See also:

coretutorial_bind_param

Insert Expressions

`outparam()`

`sqlalchemy.sql.expression.case`(*whens*, *value=None*, *else_=None*)

Produce a `CASE` expression.

The `CASE` construct in SQL is a conditional object that acts somewhat analogously to an "if/then" construct in other languages. It returns an instance of `Case`.

`case()` in its usual form is passed a list of "when" constructs, that is, a list of conditions and results as tuples:

```
from sqlalchemy import case

stmt = select([users_table]).\
            where(
                case(
                    [
                        (users_table.c.name == 'wendy', 'W'),
                        (users_table.c.name == 'jack', 'J')
                    ],
                    else_='E'
                )
            )
```

The above statement will produce SQL resembling:

```
SELECT id, name FROM user
WHERE CASE
    WHEN (name = :name_1) THEN :param_1
    WHEN (name = :name_2) THEN :param_2
    ELSE :param_3
END
```

When simple equality expressions of several values against a single parent column are needed, `case()` also has a "shorthand" format used via the `case.value` parameter, which is passed a column expression to be compared. In this form, the `case.whens` parameter is passed as a dictionary containing expressions to be compared against keyed to result expressions. The statement below is equivalent to the preceding statement:

```
stmt = select([users_table]).\
            where(
                case(
```

```
                {"wendy": "W", "jack": "J"},
                value=users_table.c.name,
                else_='E'
            )
        )
```

The values which are accepted as result values in `case.whens` as well as with `case.else_` are coerced from Python literals into `bindparam()` constructs. SQL expressions, e.g. `ColumnElement` constructs, are accepted as well. To coerce a literal string expression into a constant expression rendered inline, use the `literal_column()` construct, as in:

```
from sqlalchemy import case, literal_column

case(
    [
        (
            orderline.c.qty > 100,
            literal_column("'greaterthan100'")
        ),
        (
            orderline.c.qty > 10,
            literal_column("'greaterthan10'")
        )
    ],
    else_=literal_column("'lessthan10'")
)
```

The above will render the given constants without using bound parameters for the result values (but still for the comparison values), as in:

```
CASE
    WHEN (orderline.qty > :qty_1) THEN 'greaterthan100'
    WHEN (orderline.qty > :qty_2) THEN 'greaterthan10'
    ELSE 'lessthan10'
END
```

Parameters

- **whens** – The criteria to be compared against, `case.whens` accepts two different forms, based on whether or not `case.value` is used.

  In the first form, it accepts a list of 2-tuples; each 2-tuple consists of `(<sql expression>, <value>)`, where the SQL expression is a boolean expression and "value" is a resulting value, e.g.:

  ```
  case([
      (users_table.c.name == 'wendy', 'W'),
      (users_table.c.name == 'jack', 'J')
  ])
  ```

  In the second form, it accepts a Python dictionary of comparison values mapped to a resulting value; this form requires `case.value` to be present, and values will be compared using the `==` operator, e.g.:

  ```
  case(
      {"wendy": "W", "jack": "J"},
      value=users_table.c.name
  )
  ```

- **value** – An optional SQL expression which will be used as a fixed "comparison point" for candidate values within a dictionary passed to `case.whens`.

- **else_** – An optional SQL expression which will be the evaluated result of the CASE construct if all expressions within case.whens evaluate to false. When omitted, most databases will produce a result of NULL if none of the "when" expressions evaluate to true.

sqlalchemy.sql.expression.cast(*expression*, *type_*)

Produce a CAST expression.

cast() returns an instance of Cast.

E.g.:

```
from sqlalchemy import cast, Numeric

stmt = select([
            cast(product_table.c.unit_price, Numeric(10, 4))
        ])
```

The above statement will produce SQL resembling:

```
SELECT CAST(unit_price AS NUMERIC(10, 4)) FROM product
```

The cast() function performs two distinct functions when used. The first is that it renders the CAST expression within the resulting SQL string. The second is that it associates the given type (e.g. TypeEngine class or instance) with the column expression on the Python side, which means the expression will take on the expression operator behavior associated with that type, as well as the bound-value handling and result-row-handling behavior of the type.

Changed in version 0.9.0: cast() now applies the given type to the expression such that it takes effect on the bound-value, e.g. the Python-to-database direction, in addition to the result handling, e.g. database-to-Python, direction.

An alternative to cast() is the type_coerce() function. This function performs the second task of associating an expression with a specific type, but does not render the CAST expression in SQL.

Parameters

- **expression** – A SQL expression, such as a ColumnElement expression or a Python string which will be coerced into a bound literal value.
- **type_** – A TypeEngine class or instance indicating the type to which the CAST should apply.

See also:

type_coerce() - Python-side type coercion without emitting CAST.

sqlalchemy.sql.expression.column(*text*, *type_=None*, *is_literal=False*, *_selectable=None*)

Produce a ColumnClause object.

The ColumnClause is a lightweight analogue to the Column class. The column() function can be invoked with just a name alone, as in:

```
from sqlalchemy import column

id, name = column("id"), column("name")
stmt = select([id, name]).select_from("user")
```

The above statement would produce SQL like:

```
SELECT id, name FROM user
```

Once constructed, column() may be used like any other SQL expression element such as within select() constructs:

```
from sqlalchemy.sql import column

id, name = column("id"), column("name")
stmt = select([id, name]).select_from("user")
```

The text handled by `column()` is assumed to be handled like the name of a database column; if the string contains mixed case, special characters, or matches a known reserved word on the target backend, the column expression will render using the quoting behavior determined by the backend. To produce a textual SQL expression that is rendered exactly without any quoting, use `literal_column()` instead, or pass `True` as the value of `column.is_literal`. Additionally, full SQL statements are best handled using the `text()` construct.

`column()` can be used in a table-like fashion by combining it with the `table()` function (which is the lightweight analogue to `Table`) to produce a working table construct with minimal boilerplate:

```
from sqlalchemy import table, column, select

user = table("user",
        column("id"),
        column("name"),
        column("description"),
)

stmt = select([user.c.description]).where(user.c.name == 'wendy')
```

A `column()` / `table()` construct like that illustrated above can be created in an ad-hoc fashion and is not associated with any `schema.MetaData`, DDL, or events, unlike its `Table` counterpart.

Changed in version 1.0.0: `expression.column()` can now be imported from the plain `sqlalchemy` namespace like any other SQL element.

Parameters

- **text** – the text of the element.
- **type** – `types.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 essentially invokes `column()` while passing `is_literal=True`.

See also:

`Column`

`literal_column()`

`table()`

`text()`

sqlexpression_literal_column

`sqlalchemy.sql.expression.collate`(*expression*, *collation*)

Return the clause `expression COLLATE collation`.

e.g.:

```
collate(mycolumn, 'utf8_bin')
```

produces:

```
mycolumn COLLATE utf8_bin
```

The collation expression is also quoted if it is a case sensitive identifier, e.g. contains uppercase characters.

Changed in version 1.2: quoting is automatically applied to COLLATE expressions if they are case sensitive.

sqlalchemy.sql.expression.**desc**(*column*)

Produce a descending ORDER BY clause element.

e.g.:

```
from sqlalchemy import desc

stmt = select([users_table]).order_by(desc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name DESC
```

The desc() function is a standalone version of the ColumnElement.desc() method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.desc())
```

Parameters **column** – A ColumnElement (e.g. scalar SQL expression) with which to apply the desc() operation.

See also:

asc()

nullsfirst()

nullslast()

Select.order_by()

sqlalchemy.sql.expression.**distinct**(*expr*)

Produce an column-expression-level unary DISTINCT clause.

This applies the DISTINCT keyword to an individual column expression, and is typically contained within an aggregate function, as in:

```
from sqlalchemy import distinct, func
stmt = select([func.count(distinct(users_table.c.name))])
```

The above would produce an expression resembling:

```
SELECT COUNT(DISTINCT name) FROM user
```

The distinct() function is also available as a column-level method, e.g. ColumnElement.distinct(), as in:

```
stmt = select([func.count(users_table.c.name.distinct())])
```

The distinct() operator is different from the Select.distinct() method of Select, which produces a SELECT statement with DISTINCT applied to the result set as a whole, e.g. a SELECT DISTINCT expression. See that method for further information.

See also:

ColumnElement.distinct()

Select.distinct()

func

sqlalchemy.sql.expression.**extract**(*field*, *expr*, ***kwargs*)
Return a `Extract` construct.

This is typically available as `extract()` as well as `func.extract` from the `func` namespace.

sqlalchemy.sql.expression.**false**()
Return a `False_` construct.

E.g.:

```
>>> from sqlalchemy import false
>>> print select([t.c.x]).where(false())
SELECT x FROM t WHERE false
```

A backend which does not support true/false constants will render as an expression against 1 or 0:

```
>>> print select([t.c.x]).where(false())
SELECT x FROM t WHERE 0 = 1
```

The `true()` and `false()` constants also feature "short circuit" operation within an `and_()` or `or_()` conjunction:

```
>>> print select([t.c.x]).where(or_(t.c.x > 5, true()))
SELECT x FROM t WHERE true

>>> print select([t.c.x]).where(and_(t.c.x > 5, false()))
SELECT x FROM t WHERE false
```

Changed in version 0.9: `true()` and `false()` feature better integrated behavior within conjunctions and on dialects that don't support true/false constants.

See also:

`true()`

sqlalchemy.sql.expression.**func** = <sqlalchemy.sql.functions._FunctionGenerator object>
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)
```

The element is a column-oriented SQL element like any other, and is used in that way:

```
>>> print(select([func.count(table.c.id)]))
SELECT count(sometable.id) FROM sometable
```

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)
```

The object returned by a `func` call is usually an instance of `Function`. This object meets the "column" interface, including comparison and labeling functions. The object can also be passed the `execute()` method of a `Connection` or `Engine`, where it will be wrapped inside of a SELECT statement first:

```
print(connection.execute(func.current_timestamp()).scalar())
```

In a few exception cases, the `func` accessor will redirect a name to a built-in expression such as `cast()` or `extract()`, as these names have well-known meaning but are not exactly the same as "functions" from a SQLAlchemy perspective.

New in version 0.8: `func` can return non-function expression constructs for common quasi-functional names like `cast()` and `extract()`.

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.

Note: The `func` construct has only limited support for calling standalone "stored procedures", especially those with special parameterization concerns.

See the section stored_procedures for details on how to use the DBAPI-level `callproc()` method for fully traditional stored procedures.

`sqlalchemy.sql.expression.funcfilter`(*func*, **criterion*)

Produce a `FunctionFilter` object against a function.

Used against aggregate and window functions, for database backends that support the "FILTER" clause.

E.g.:

```
from sqlalchemy import funcfilter
funcfilter(func.count(1), MyClass.name == 'some name')
```

Would produce "COUNT(1) FILTER (WHERE myclass.name = 'some name')".

This function is also available from the `func` construct itself via the `FunctionElement.filter()` method.

New in version 1.0.0.

See also:

`FunctionElement.filter()`

`sqlalchemy.sql.expression.label`(*name*, *element*, *type_=None*)

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 `ColumnElement.label()` method on `ColumnElement`.

Parameters

- **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 `ColumnElement` subclass, such as a `Column` object. Use this function to force the generation of a literal clause, which will be created as a `BindParameter` 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*)
Produce a `ColumnClause` object that has the `column.is_literal` flag set to True.

`literal_column()` is similar to `column()`, except that it is more often used as a "standalone" column expression that renders exactly as stated; while `column()` stores a string name that will be assumed to be part of a table and may be quoted as such, `literal_column()` can be that, or any other arbitrary column-oriented expression.

Parameters

- **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.

See also:

`column()`

`text()`

sqlexpression_literal_column

sqlalchemy.sql.expression.**not_**(*clause*)
Return a negation of the given clause, i.e. `NOT(clause)`.

The `~` operator is also overloaded on all `ColumnElement` subclasses to produce the same result.

sqlalchemy.sql.expression.**null**()
Return a constant `Null` construct.

sqlalchemy.sql.expression.**nullsfirst**(*column*)
Produce the `NULLS FIRST` modifier for an `ORDER BY` expression.

`nullsfirst()` is intended to modify the expression produced by `asc()` or `desc()`, and indicates how NULL values should be handled when they are encountered during ordering:

```
from sqlalchemy import desc, nullsfirst

stmt = select([users_table]).                    order_by(nullsfirst(desc(users_table.
↪c.name)))
```

The SQL expression from the above would resemble:

```
SELECT id, name FROM user ORDER BY name DESC NULLS FIRST
```

Like `asc()` and `desc()`, `nullsfirst()` is typically invoked from the column expression itself using `ColumnElement.nullsfirst()`, rather than as its standalone function version, as in:

```
stmt = (select([users_table]).
        order_by(users_table.c.name.desc().nullsfirst())
        )
```

See also:

`asc()`

`desc()`

`nullslast()`

`Select.order_by()`

sqlalchemy.sql.expression.**nullslast**(*column*)

Produce the `NULLS LAST` modifier for an `ORDER BY` expression.

`nullslast()` is intended to modify the expression produced by `asc()` or `desc()`, and indicates how NULL values should be handled when they are encountered during ordering:

```
from sqlalchemy import desc, nullslast

stmt = select([users_table]).                                    order_by(nullslast(desc(users_table.
↪c.name)))
```

The SQL expression from the above would resemble:

```
SELECT id, name FROM user ORDER BY name DESC NULLS LAST
```

Like `asc()` and `desc()`, `nullslast()` is typically invoked from the column expression itself using `ColumnElement.nullslast()`, rather than as its standalone function version, as in:

```
stmt = select([users_table]).                                    order_by(users_table.c.name.desc().
↪nullslast())
```

See also:

`asc()`

`desc()`

`nullsfirst()`

`Select.order_by()`

sqlalchemy.sql.expression.**or_**(**clauses*)

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The `or_()` conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
```

```
                (users_table.c.name == 'jack')
            )
```

See also:

`and_()`

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.**over**(*element*, *partition_by=None*, *order_by=None*, *range_=None*, *rows=None*)

Produce an `Over` object against a function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

`over()` is usually called using the `FunctionElement.over()` method, e.g.:

```
func.row_number().over(order_by=mytable.c.some_column)
```

Would produce:

```
ROW_NUMBER() OVER(ORDER BY some_column)
```

Ranges are also possible using the `expression.over.range_` and `expression.over.rows` parameters. These mutually-exclusive parameters each accept a 2-tuple, which contains a combination of integers and None:

```
func.row_number().over(order_by=my_table.c.some_column, range_=(None, 0))
```

The above would produce:

```
ROW_NUMBER() OVER(ORDER BY some_column RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

A value of None indicates "unbounded", a value of zero indicates "current row", and negative / positive integers indicate "preceding" and "following":

- RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING:

  ```
  func.row_number().over(order_by='x', range_=(-5, 10))
  ```

- ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW:

  ```
  func.row_number().over(order_by='x', rows=(None, 0))
  ```

- RANGE BETWEEN 2 PRECEDING AND UNBOUNDED FOLLOWING:

  ```
  func.row_number().over(order_by='x', range_=(-2, None))
  ```

- RANGE BETWEEN 1 FOLLOWING AND 3 FOLLOWING:

  ```
  func.row_number().over(order_by='x', range_=(1, 3))
  ```

New in version 1.1: support for RANGE / ROWS within a window

Parameters

- **element** – a `FunctionElement`, `WithinGroup`, or other compatible construct.

- **partition_by** – a column element or string, or a list of such, that will be used as the PARTITION BY clause of the OVER construct.
- **order_by** – a column element or string, or a list of such, that will be used as the ORDER BY clause of the OVER construct.
- **range_** – optional range clause for the window. This is a tuple value which can contain integer values or None, and will render a RANGE BETWEEN PRECEDING / FOLLOWING clause

  New in version 1.1.
- **rows** – optional rows clause for the window. This is a tuple value which can contain integer values or None, and will render a ROWS BETWEEN PRECEDING / FOLLOWING clause.

  New in version 1.1.

This function is also available from the `func` construct itself via the `FunctionElement.over()` method.

See also:

`expression.func`

`expression.within_group()`

`sqlalchemy.sql.expression.text(text, bind=None, bindparams=None, typemap=None, autocommit=None)`

Construct a new `TextClause` clause, representing a textual SQL string directly.

E.g.:

```
from sqlalchemy import text

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. The construct can also be provided with a `.c` collection of column elements, allowing it to be embedded in other SQL expression constructs as a subquery.

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)
```

For SQL statements where a colon is required verbatim, as within an inline string, use a backslash to escape:

```
t = text("SELECT * FROM users WHERE name='\:username'")
```

The `TextClause` construct includes methods which can provide information about the bound parameters as well as the column values which would be returned from the textual statement, assuming it's an executable SELECT type of statement. The `TextClause.bindparams()` method is used to provide bound parameter detail, and `TextClause.columns()` method allows specification of return columns including names and types:

```
t = text("SELECT * FROM users WHERE id=:user_id").\
        bindparams(user_id=7).\
        columns(id=Integer, name=String)

for id, name in connection.execute(t):
    print(id, name)
```

The `text()` construct is used in cases when a literal string SQL fragment is specified as part of a larger query, such as for the WHERE clause of a SELECT statement:

```
s = select([users.c.id, users.c.name]).where(text("id=:user_id"))
result = connection.execute(s, user_id=12)
```

`text()` is also used for the construction of a full, standalone statement using plain text. 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 `Connection.execution_options.autocommit` option:

```
t = text("EXEC my_procedural_thing()").\
        execution_options(autocommit=True)
```

Note that SQLAlchemy's usual "autocommit" behavior applies to `text()` constructs implicitly - 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** – Deprecated. A list of `bindparam()` instances used to provide information about parameters embedded in the statement. This argument now invokes the `TextClause.bindparams()` method on the construct before returning it. E.g.:

  ```
  stmt = text("SELECT * FROM table WHERE id=:id",
            bindparams=[bindparam('id', value=5, type_=Integer)])
  ```

  Is equivalent to:

  ```
  stmt = text("SELECT * FROM table WHERE id=:id").\
            bindparams(bindparam('id', value=5, type_=Integer))
  ```

  Deprecated since version 0.9.0: the `TextClause.bindparams()` method supersedes the `bindparams` argument to `text()`.

- **typemap** – Deprecated. 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 parameter now invokes the `TextClause.columns()` method, which returns a `TextAsFrom` construct that gains a `.c` collection and can be embedded in other expressions. E.g.:

  ```
  stmt = text("SELECT * FROM table",
                typemap={'id': Integer, 'name': String},
            )
  ```

  Is equivalent to:

  ```
  stmt = text("SELECT * FROM table").columns(id=Integer,
                                             name=String)
  ```

  Or alternatively:

```
from sqlalchemy.sql import column
stmt = text("SELECT * FROM table").columns(
                        column('id', Integer),
                        column('name', String)
                    )
```

Deprecated since version 0.9.0: the `TextClause.columns()` method supersedes the `typemap` argument to `text()`.

See also:

sqlexpression_text - in the Core tutorial

orm_tutorial_literal_sql - in the ORM tutorial

`sqlalchemy.sql.expression.true()`
Return a constant `True_` construct.

E.g.:

```
>>> from sqlalchemy import true
>>> print select([t.c.x]).where(true())
SELECT x FROM t WHERE true
```

A backend which does not support true/false constants will render as an expression against 1 or 0:

```
>>> print select([t.c.x]).where(true())
SELECT x FROM t WHERE 1 = 1
```

The `true()` and `false()` constants also feature "short circuit" operation within an `and_()` or `or_()` conjunction:

```
>>> print select([t.c.x]).where(or_(t.c.x > 5, true()))
SELECT x FROM t WHERE true

>>> print select([t.c.x]).where(and_(t.c.x > 5, false()))
SELECT x FROM t WHERE false
```

Changed in version 0.9: `true()` and `false()` feature better integrated behavior within conjunctions and on dialects that don't support true/false constants.

See also:

`false()`

`sqlalchemy.sql.expression.tuple_(*clauses, **kw)`
Return a `Tuple`.

Main usage is to produce a composite IN construct:

```
from sqlalchemy import tuple_

tuple_(table.c.col1, table.c.col2).in_(
    [(1, 2), (5, 12), (10, 19)]
)
```

Warning: The composite IN construct is not supported by all backends, and is currently known to work on PostgreSQL and MySQL, but not SQLite. Unsupported backends will raise a subclass of `DBAPIError` when such an expression is invoked.

`sqlalchemy.sql.expression.type_coerce(expression, type_)`
Associate a SQL expression with a particular type, without rendering `CAST`.

E.g.:

```
from sqlalchemy import type_coerce

stmt = select([
    type_coerce(log_table.date_string, StringDateTime())
])
```

The above construct will produce a `TypeCoerce` object, which renders SQL that labels the expression, but otherwise does not modify its value on the SQL side:

```
SELECT date_string AS anon_1 FROM log
```

When result rows are fetched, the `StringDateTime` type will be applied to result rows on behalf of the `date_string` column. The rationale for the "anon_1" label is so that the type-coerced column remains separate in the list of result columns vs. other type-coerced or direct values of the target column. In order to provide a named label for the expression, use `ColumnElement.label()`:

```
stmt = select([
    type_coerce(
        log_table.date_string, StringDateTime()).label('date')
])
```

A type that features bound-value handling will also have that behavior take effect when literal values or `bindparam()` constructs are passed to `type_coerce()` as targets. For example, if a type implements the `TypeEngine.bind_expression()` method or `TypeEngine.bind_processor()` method or equivalent, these functions will take effect at statement compilation/execution time when a literal value is passed, as in:

```
# bound-value handling of MyStringType will be applied to the
# literal value "some string"
stmt = select([type_coerce("some string", MyStringType)])
```

`type_coerce()` is similar to the `cast()` function, except that it does not render the `CAST` expression in the resulting statement.

Parameters

- **expression** – A SQL expression, such as a `ColumnElement` expression or a Python string which will be coerced into a bound literal value.
- **type_** – A `TypeEngine` class or instance indicating the type to which the expression is coerced.

See also:

`cast()`

`sqlalchemy.sql.expression.within_group(element, *order_by)`

Produce a `WithinGroup` object against a function.

Used against so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including `percentile_cont`, `rank`, `dense_rank`, etc.

`within_group()` is usually called using the `FunctionElement.within_group()` method, e.g.:

```
from sqlalchemy import within_group
stmt = select([
    department.c.id,
    func.percentile_cont(0.5).within_group(
        department.c.salary.desc()
    )
])
```

The above statement would produce SQL similar to `SELECT department.id, percentile_cont(0.5) WITHIN GROUP (ORDER BY department.salary DESC)`.

Parameters

- **element** – a `FunctionElement` construct, typically generated by `func`.
- ***order_by** – one or more column elements that will be used as the ORDER BY clause of the WITHIN GROUP construct.

New in version 1.1.

See also:

`expression.func`

`expression.over()`

class sqlalchemy.sql.expression.**BinaryExpression**(*left*, *right*, *operator*, *type_=None*, *negate=None*, *modifiers=None*)

Represent an expression that is `LEFT <operator> RIGHT`.

A `BinaryExpression` is generated automatically whenever two column expressions are used in a Python binary expression:

```
>>> from sqlalchemy.sql import column
>>> column('a') + column('b')
<sqlalchemy.sql.expression.BinaryExpression object at 0x101029dd0>
>>> print column('a') + column('b')
a + b
```

compare(*other*, ***kw*)

Compare this `BinaryExpression` against the given `BinaryExpression`.

class sqlalchemy.sql.expression.**BindParameter**(*key*, *value=symbol('NO_ARG')*, *type_=None*, *unique=False*, *required=symbol('NO_ARG')*, *quote=None*, *callable_=None*, *expanding=False*, *isoutparam=False*, *_compared_to_operator=None*, *_compared_to_type=None*)

Represent a "bound expression".

`BindParameter` is invoked explicitly using the `bindparam()` function, as in:

```
from sqlalchemy import bindparam

stmt = select([users_table]).\
            where(users_table.c.name == bindparam('username'))
```

Detailed discussion of how `BindParameter` is used is at `bindparam()`.

See also:

`bindparam()`

compare(*other*, ***kw*)

Compare this `BindParameter` to the given clause.

effective_value

Return the value of this bound parameter, taking into account if the `callable` parameter was set.

The `callable` value will be evaluated and returned if present, else `value`.

class sqlalchemy.sql.expression.**Case**(*whens*, *value=None*, *else_=None*)

Represent a `CASE` expression.

`Case` is produced using the `case()` factory function, as in:

```
from sqlalchemy import case

stmt = select([users_table]).                    where(
                case(
                    [
                        (users_table.c.name == 'wendy', 'W'),
                        (users_table.c.name == 'jack', 'J')
                    ],
                    else_='E'
                )
            )
```

Details on `Case` usage is at `case()`.

See also:

`case()`

class sqlalchemy.sql.expression.**Cast**(*expression*, *type_*)

Represent a `CAST` expression.

`Cast` is produced using the `cast()` factory function, as in:

```
from sqlalchemy import cast, Numeric

stmt = select([
            cast(product_table.c.unit_price, Numeric(10, 4))
        ])
```

Details on `Cast` usage is at `cast()`.

See also:

`cast()`

class sqlalchemy.sql.expression.**ClauseElement**

Base class for elements of a programmatically constructed SQL expression.

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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()` elements 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}
```

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()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

`class sqlalchemy.sql.expression.ClauseList(*clauses, **kwargs)`

Describe a list of clauses, separated by an operator.

By default, is comma-separated, such as a column listing.

`compare(other, **kw)`

Compare this `ClauseList` to the given `ClauseList`, including a comparison of all the clause items.

`class sqlalchemy.sql.expression.ColumnClause(text, type_=None, is_literal=False, _selectable=None)`

Represents a column expression from any textual string.

The `ColumnClause`, a lightweight analogue to the `Column` class, is typically invoked using the `column()` function, as in:

```
from sqlalchemy import column

id, name = column("id"), column("name")
stmt = select([id, name]).select_from("user")
```

The above statement would produce SQL like:

```
SELECT id, name FROM user
```

`ColumnClause` is the immediate superclass of the schema-specific `Column` object. While the `Column` class has all the same capabilities as `ColumnClause`, the `ColumnClause` class is usable by itself in those cases where behavioral requirements are limited to simple SQL expression generation. The object has none of the associations with schema-level metadata or with execution-time behavior that `Column` does, so in that sense is a "lightweight" version of `Column`.

Full details on `ColumnClause` usage is at `column()`.

See also:

`column()`

`Column`

`class sqlalchemy.sql.expression.ColumnCollection(*columns)`

An ordered dictionary that stores a list of ColumnElement instances.

Overrides the `__eq__()` method to produce SQL clauses between sets of correlated columns.

`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

Represent a column-oriented SQL expression suitable for usage in the "columns" clause, WHERE clause etc. of a statement.

While the most familiar kind of ColumnElement is the Column object, ColumnElement serves as the basis for any unit that may be present in a SQL expression, including the expressions themselves, SQL functions, bound parameters, literal expressions, keywords such as NULL, etc. ColumnElement is the ultimate base class for all such elements.

A wide variety of SQLAlchemy Core functions work at the SQL expression level, and are intended to accept instances of ColumnElement as arguments. These functions will typically document that they accept a "SQL expression" as an argument. What this means in terms of SQLAlchemy usually refers to an input which is either already in the form of a ColumnElement object, or a value which can be **coerced** into one. The coercion rules followed by most, but not all, SQLAlchemy Core functions with regards to SQL expressions are as follows:

- a literal Python value, such as a string, integer or floating point value, boolean, datetime, Decimal object, or virtually any other Python object, will be coerced into a "literal bound value". This generally means that a bindparam() will be produced featuring the given value embedded into the construct; the resulting BindParameter object is an instance of ColumnElement. The Python value will ultimately be sent to the DBAPI at execution time as a parameterized argument to the execute() or executemany() methods, after SQLAlchemy type-specific converters (e.g. those provided by any associated TypeEngine objects) are applied to the value.
- any special object value, typically ORM-level constructs, which feature a method called __clause_element__(). The Core expression system looks for this method when an object of otherwise unknown type is passed to a function that is looking to coerce the argument into a ColumnElement expression. The __clause_element__() method, if present, should return a ColumnElement instance. The primary use of __clause_element__() within SQLAlchemy is that of class-bound attributes on ORM-mapped classes; a User class which contains a mapped attribute named .name will have a method User.name.__clause_element__() which when invoked returns the Column called name associated with the mapped table.
- The Python None value is typically interpreted as NULL, which in SQLAlchemy Core produces an instance of null().

A ColumnElement provides the ability to generate new ColumnElement objects using Python expressions. This means that Python operators such as ==, != and < are overloaded to mimic SQL operations, and allow the instantiation of further ColumnElement instances which are composed from other, more fundamental ColumnElement objects. For example, two ColumnClause objects can be added together with the addition operator + to produce a BinaryExpression. Both ColumnClause and BinaryExpression are subclasses of ColumnElement:

```
>>> from sqlalchemy.sql import column
>>> column('a') + column('b')
<sqlalchemy.sql.expression.BinaryExpression object at 0x101029dd0>
>>> print column('a') + column('b')
a + b
```

See also:

Column

expression.column()

all_()

Produce a all_() clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == mytable.c.somearray.all_()
```

```
# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().all_()
```

See also:

`all_()` - standalone version

`any_()` - ANY operator

New in version 1.1.

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.

any_()
Produce a `any_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == mytable.c.somearray.any_()

# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().any_()
```

See also:

`any_()` - standalone version

`all_()` - ALL operator

New in version 1.1.

asc()
Produce a `asc()` clause against the parent object.

base_columns

between(*cleft*, *cright*, *symmetric=False*)
Produce a `between()` clause against the parent object, given the lower and upper range.

bind = None

bool_op(*opstring*, *precedence=0*)
Return a custom boolean operator.

This method is shorthand for calling `Operators.op()` and passing the `Operators.op.is_comparison` flag with True.

New in version 1.2.0b3.

See also:

`Operators.op()`

cast(*type_*)
Produce a type cast, i.e. `CAST(<expression> AS <type>)`.

This is a shortcut to the `cast()` function.

New in version 1.0.7.

`collate`(*collation*)
Produce a `collate()` clause against the parent object, given the collation string.

See also:

`collate()`

`comparator`

`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 corresponding 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.

`compile`(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`concat`(*other*)
Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

`contains`(*other*, ***kwargs*)
Implement the 'contains' operator.

Produces a LIKE expression that tests against a match for the middle of a string value:

```
column LIKE '%' || <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.contains("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.contains.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.contains.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.contains.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.contains("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '/'
  ```

  With the value of :param as `"foo/%bar"`.

  New in version 1.2.

  Changed in version 1.2.0: The `ColumnOperators.contains.autoescape` parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the `ColumnOperators.contains.escape` parameter.
- **escape** – a character which when given will render with the `ESCAPE` keyword to establish that character as the escape character. This character can then be placed preceding occurrences of `%` and `_` to allow them to act as themselves and not wildcard characters.

  An expression such as:

```
somecolumn.contains("foo/%bar", escape="^")
```

Will render as:

```
somecolumn LIKE '%' || :param || '%' ESCAPE '^'
```

The parameter may also be combined with `ColumnOperators.contains.autoescape`:

```
somecolumn.contains("foo%bar^bat", escape="^", autoescape=True)
```

Where above, the given literal parameter will be converted to `"foo^%bar^^bat"` before being passed to the database.

See also:

`ColumnOperators.startswith()`

`ColumnOperators.endswith()`

`ColumnOperators.like()`

desc()
Produce a `desc()` clause against the parent object.

description = None

distinct()
Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)
Implement the 'endswith' operator.

Produces a LIKE expression that tests against a match for the end of a string value:

```
column LIKE '%' || <other>
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.endswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.endswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.endswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the `ColumnOperators.endswith.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

```
somecolumn.endswith("foo%bar", autoescape=True)
```

Will render as:

```
somecolumn LIKE '%' || :param ESCAPE '/'
```

With the value of :param as "foo/%bar".

New in version 1.2.

Changed in version 1.2.0: The ColumnOperators.endswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.endswith.escape parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

```
somecolumn.endswith("foo/%bar", escape="^")
```

  Will render as:

```
somecolumn LIKE '%' || :param ESCAPE '^'
```

  The parameter may also be combined with ColumnOperators.endswith.autoescape:

```
somecolumn.endswith("foo%bar^bat", escape="^", autoescape=True)
```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.contains()

ColumnOperators.like()

expression
Return a column expression.

Part of the inspection interface; returns self.

foreign_keys = []

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).

ilike(*other*, *escape=None*)
Implement the ilike operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

```
lower(a) LIKE lower(other)
```

Or on backends that support the ILIKE operator:

```
a ILIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)
Implement the in operator.

In a column context, produces the clause `a IN other`. "other" may be a tuple/list of column expressions, or a `select()` construct.

In the case that `other` is an empty sequence, the compiler produces an "empty in" expression. This defaults to the expression "1 != 1" to produce false in all cases. The `create_engine.empty_in_strategy` may be used to alter this behavior.

Changed in version 1.2: The `ColumnOperators.in_()` and `ColumnOperators.notin_()` operators now produce a "static" expression for an empty IN sequence by default.

is_(*other*)
Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of `None`, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

is_clause_element = True

is_distinct_from(*other*)
Implement the IS DISTINCT FROM operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

New in version 1.1.

is_selectable = False

isnot(*other*)
Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of `None`, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

isnot_distinct_from(*other*)
Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

New in version 1.1.

key = None
the 'key' that in some circumstances refers to this object in a Python namespace.

This typically refers to the "key" of the column as present in the .c collection of a selectable, e.g. sometable.c["somekey"] would return a Column with a .key of "somekey".

label(*name*)
Produce a column label, i.e. <columnname> AS <name>.

This is a shortcut to the label() function.

if 'name' is None, an anonymous label name will be generated.

like(*other*, *escape=None*)
Implement the like operator.

In a column context, produces the expression:

```
a LIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)
Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- PostgreSQL - renders x @@ to_tsquery(y)
- MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
- Oracle - renders CONTAINS(x, y)
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
implement the NOT ILIKE operator.

This is equivalent to using negation with ColumnOperators.ilike(), i.e. ~x.ilike(y).

New in version 0.8.

See also:

`ColumnOperators.ilike()`

notin_(*other*)
implement the `NOT IN` operator.

This is equivalent to using negation with `ColumnOperators.in_()`, i.e. `~x.in_(y)`.

In the case that `other` is an empty sequence, the compiler produces an "empty not in" expression. This defaults to the expression "1 = 1" to produce true in all cases. The `create_engine.empty_in_strategy` may be used to alter this behavior.

Changed in version 1.2: The `ColumnOperators.in_()` and `ColumnOperators.notin_()` operators now produce a "static" expression for an empty IN sequence by default.

See also:

`ColumnOperators.in_()`

notlike(*other*, *escape=None*)
implement the `NOT LIKE` operator.

This is equivalent to using negation with `ColumnOperators.like()`, i.e. `~x.like(y)`.

New in version 0.8.

See also:

`ColumnOperators.like()`

nullsfirst()
Produce a `nullsfirst()` clause against the parent object.

nullslast()
Produce a `nullslast()` clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.

- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the Operators.op.is_comparison flag.

- **return_type** – a TypeEngine class or object that will force the return type of an expression produced by this operator to be of that type. By default, operators that specify Operators.op.is_comparison will resolve to Boolean, and those that do not will be of the same type as the left-hand operand.

  New in version 1.2.0b3: - added the Operators.op.return_type argument.

See also:

types_operators

relationship_custom_operator

operate(*op*, **other*, ***kwargs*)

params(**optionaldict*, ***kwargs*)

Return a copy with bindparam() elements replaced.

Returns a copy of this ClauseElement with bindparam() elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key = False

proxy_set

reverse_operate(*op*, *other*, ***kwargs*)

self_group(*against=None*)

shares_lineage(*othercolumn*)

Return True if the given ColumnElement has a common ancestor to this ColumnElement.

startswith(*other*, ***kwargs*)

Implement the startswith operator.

Produces a LIKE expression that tests against a match for the start of a string value:

```
column LIKE <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.startswith("foobar"))
```

Since the operator uses LIKE, wildcard characters "%" and "_" that are present inside the <other> expression will behave like wildcards as well. For literal string values, the ColumnOperators.startswith.autoescape flag may be set to True to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the ColumnOperators.startswith.escape parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.startswith.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.startswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.startswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.startswith.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.startswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.startswith.autoescape:

  ```
  somecolumn.startswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.endswith()

ColumnOperators.contains()

ColumnOperators.like()

supports_execution = False

timetuple = None

type

unique_params(**optionaldict*, ***kwargs*)

Return a copy with bindparam() elements replaced.

Same functionality as params(), except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.operators.ColumnOperators

Defines boolean, comparison, and other operators for ColumnElement expressions.

By default, all methods call down to operate() or reverse_operate(), passing in the appropriate operator function from the Python builtin operator module or a SQLAlchemy-specific operator function from sqlalchemy.expression.operators. For example the __eq__ function:

```
def __eq__(self, other):
    return self.operate(operators.eq, other)
```

Where operators.eq is essentially:

```
def eq(a, b):
    return a == b
```

The core column expression unit ColumnElement overrides Operators.operate() and others to return further ColumnElement constructs, so that the == operation above is replaced by a clause construct.

See also:

types_operators

TypeEngine.comparator_factory

ColumnOperators

PropComparator

__add__(*other*)

Implement the + operator.

In a column context, produces the clause a + b if the parent object has non-string affinity. If the parent object has a string affinity, produces the concatenation operator, a || b - see ColumnOperators.concat().

__and__(*other*)

Implement the & operator.

When used with SQL expressions, results in an AND operation, equivalent to and_(), that is:

```
a & b
```

is equivalent to:

```
from sqlalchemy import and_
and_(a, b)
```

Care should be taken when using & regarding operator precedence; the & operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) & (b == 4)
```

__delattr__

Implement delattr(self, name).

__dir__() → list

default dir() implementation

__div__(*other*)

Implement the / operator.

In a column context, produces the clause a / b.

__eq__(*other*)
Implement the == operator.

In a column context, produces the clause `a = b`. If the target is `None`, produces `a IS NULL`.

__format__()
default object formatter

__ge__(*other*)
Implement the >= operator.

In a column context, produces the clause `a >= b`.

__getattribute__
Return getattr(self, name).

__getitem__(*index*)
Implement the [] operator.

This can be used by some database-specific types such as PostgreSQL ARRAY and HSTORE.

__gt__(*other*)
Implement the > operator.

In a column context, produces the clause `a > b`.

__hash__
Return hash(self).

__init__
Initialize self. See help(type(self)) for accurate signature.

__init_subclass__()
This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__invert__()
Implement the ~ operator.

When used with SQL expressions, results in a NOT operation, equivalent to `not_()`, that is:

```
~a
```

is equivalent to:

```
from sqlalchemy import not_
not_(a)
```

__le__(*other*)
Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lshift__(*other*)
implement the << operator.

Not used by SQLAlchemy core, this is provided for custom operator systems which want to use << as an extension point.

__lt__(*other*)
Implement the < operator.

In a column context, produces the clause `a < b`.

__mod__(*other*)
Implement the % operator.

In a column context, produces the clause `a % b`.

__mul__(*other*)
Implement the * operator.

In a column context, produces the clause `a * b`.

__ne__(*other*)
Implement the != operator.

In a column context, produces the clause `a != b`. If the target is `None`, produces `a IS NOT NULL`.

__neg__()
Implement the - operator.

In a column context, produces the clause `-a`.

__new__()
Create and return a new object. See help(type) for accurate signature.

__or__(*other*)
Implement the | operator.

When used with SQL expressions, results in an OR operation, equivalent to `or_()`, that is:

```
a | b
```

is equivalent to:

```
from sqlalchemy import or_
or_(a, b)
```

Care should be taken when using | regarding operator precedence; the | operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) | (b == 4)
```

__radd__(*other*)
Implement the + operator in reverse.

See `ColumnOperators.__add__()`.

__rdiv__(*other*)
Implement the / operator in reverse.

See `ColumnOperators.__div__()`.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__rmod__(*other*)
Implement the % operator in reverse.

See `ColumnOperators.__mod__()`.

__rmul__(*other*)
Implement the * operator in reverse.

See `ColumnOperators.__mul__()`.

__rshift__(*other*)
implement the >> operator.

Not used by SQLAlchemy core, this is provided for custom operator systems which want to use >> as an extension point.

__rsub__(*other*)
Implement the - operator in reverse.

See `ColumnOperators.__sub__()`.

__rtruediv__(*other*)
Implement the // operator in reverse.

See `ColumnOperators.__truediv__()`.

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__sub__(*other*)
Implement the - operator.

In a column context, produces the clause `a - b`.

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.___subclasscheck___(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__truediv__(*other*)
Implement the // operator.

In a column context, produces the clause `a / b`.

all_()
Produce a `all_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == mytable.c.somearray.all_()

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().all_()
```

See also:

`all_()` - standalone version

`any_()` - ANY operator

New in version 1.1.

any_()
Produce a `any_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == mytable.c.somearray.any_()
```

```
# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().any_()
```

See also:

`any_()` - standalone version

`all_()` - ALL operator

New in version 1.1.

`asc()`
Produce a `asc()` clause against the parent object.

`between`(*cleft*, *cright*, *symmetric=False*)
Produce a `between()` clause against the parent object, given the lower and upper range.

`bool_op`(*opstring*, *precedence=0*)
Return a custom boolean operator.

This method is shorthand for calling `Operators.op()` and passing the `Operators.op.is_comparison` flag with True.

New in version 1.2.0b3.

See also:

`Operators.op()`

`collate`(*collation*)
Produce a `collate()` clause against the parent object, given the collation string.

See also:

`collate()`

`concat`(*other*)
Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

`contains`(*other*, ***kwargs*)
Implement the 'contains' operator.

Produces a LIKE expression that tests against a match for the middle of a string value:

```
column LIKE '%' || <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.contains("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.contains.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.contains.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **`other`** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.contains.autoescape` flag is set to True.

- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.contains("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.contains.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.contains.escape parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.contains("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.contains.autoescape:

  ```
  somecolumn.contains("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.endswith()

ColumnOperators.like()

desc()
Produce a desc() clause against the parent object.

distinct()
Produce a distinct() clause against the parent object.

endswith(*other*, ***kwargs*)
Implement the 'endswith' operator.

Produces a LIKE expression that tests against a match for the end of a string value:

```
column LIKE '%' || <other>
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.endswith("foobar"))
```

Since the operator uses LIKE, wildcard characters "%" and "_" that are present inside the <other> expression will behave like wildcards as well. For literal string values, the ColumnOperators.endswith.autoescape flag may be set to True to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the ColumnOperators.endswith.escape parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.endswith.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.endswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.endswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.endswith.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.endswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.endswith.autoescape:

  ```
  somecolumn.endswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.contains()

ColumnOperators.like()

ilike(*other*, *escape=None*)
Implement the **ilike** operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

```
lower(a) LIKE lower(other)
```

Or on backends that support the ILIKE operator:

```
a ILIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the **ESCAPE** keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)
Implement the **in** operator.

In a column context, produces the clause **a IN other**. "other" may be a tuple/list of column expressions, or a **select()** construct.

In the case that **other** is an empty sequence, the compiler produces an "empty in" expression. This defaults to the expression "1 != 1" to produce false in all cases. The **create_engine.empty_in_strategy** may be used to alter this behavior.

Changed in version 1.2: The **ColumnOperators.in_()** and **ColumnOperators.notin_()** operators now produce a "static" expression for an empty IN sequence by default.

is_(*other*)
Implement the **IS** operator.

Normally, **IS** is generated automatically when comparing to a value of **None**, which resolves to **NULL**. However, explicit usage of **IS** may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

is_distinct_from(*other*)
Implement the **IS DISTINCT FROM** operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

New in version 1.1.

isnot(*other*)
Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

isnot_distinct_from(*other*)
Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

New in version 1.1.

like(*other*, *escape=None*)
Implement the like operator.

In a column context, produces the expression:

```
a LIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)
Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- PostgreSQL - renders x @@ to_tsquery(y)
- MySQL - renders MATCH (x) AGAINST (y IN BOOLEAN MODE)
- Oracle - renders CONTAINS(x, y)
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
implement the NOT ILIKE operator.

This is equivalent to using negation with ColumnOperators.ilike(), i.e. ~x.ilike(y).

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)
implement the NOT IN operator.

This is equivalent to using negation with ColumnOperators.in_(), i.e. ~x.in_(y).

In the case that other is an empty sequence, the compiler produces an "empty not in" expression. This defaults to the expression "1 = 1" to produce true in all cases. The create_engine.empty_in_strategy may be used to alter this behavior.

Changed in version 1.2: The ColumnOperators.in_() and ColumnOperators.notin_() operators now produce a "static" expression for an empty IN sequence by default.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)
implement the NOT LIKE operator.

This is equivalent to using negation with ColumnOperators.like(), i.e. ~x.like(y).

New in version 0.8.

See also:

ColumnOperators.like()

nullsfirst()
Produce a nullsfirst() clause against the parent object.

nullslast()
Produce a nullslast() clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of 0 is lower than all operators except for the comma (,) and AS operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >,

etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

New in version 0.9.2: - added the `Operators.op.is_comparison` flag.

- **return_type** – a `TypeEngine` class or object that will force the return type of an expression produced by this operator to be of that type. By default, operators that specify `Operators.op.is_comparison` will resolve to `Boolean`, and those that do not will be of the same type as the left-hand operand.

  New in version 1.2.0b3: - added the `Operators.op.return_type` argument.

See also:

types_operators

relationship_custom_operator

`operate`(*op*, **other*, ***kwargs*)

Operate on an argument.

This is the lowest level of operation, raises `NotImplementedError` by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding `ColumnOperators` to apply `func.lower()` to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as `ColumnOperators.contains()`.

`reverse_operate`(*op*, *other*, ***kwargs*)

Reverse operate on an argument.

Usage is the same as `operate()`.

`startswith`(*other*, ***kwargs*)

Implement the `startswith` operator.

Produces a LIKE expression that tests against a match for the start of a string value:

```
column LIKE <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.startswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.startswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.startswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the ColumnOperators.startswith.autoescape flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of "%", "_" and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.startswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '/'
  ```

  With the value of :param as "foo/%bar".

  New in version 1.2.

  Changed in version 1.2.0: The ColumnOperators.startswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.startswith.escape parameter.
- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

  ```
  somecolumn.startswith("foo/%bar", escape="^")
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '^'
  ```

  The parameter may also be combined with ColumnOperators.startswith.autoescape:

  ```
  somecolumn.startswith("foo%bar^bat", escape="^", autoescape=True)
  ```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.endswith()

ColumnOperators.contains()

ColumnOperators.like()

timetuple = None
Hack, allows datetime objects to be compared on the LHS.

class sqlalchemy.sql.base.DialectKWArgs
Establish the ability for a class to have dialect-specific arguments with defaults and constructor validation.

The DialectKWArgs interacts with the DefaultDialect.construct_arguments present on a dialect.

See also:

DefaultDialect.construct_arguments

classmethod argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The DialectKWArgs.argument_for() method is a per-argument way adding extra arguments to the DefaultDialect.construct_arguments dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a NoSuchModuleError is raised. The dialect must also include an existing DefaultDialect.construct_arguments collection, indicating that it participates in the keyword-argument validation and default system, else ArgumentError is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

dialect_kwargs

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>_<kwarg> format. Only arguments that were actually passed are included; unlike the DialectKWArgs.dialect_options collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>_<kwarg> where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The DialectKWArgs.dialect_kwargs collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect_name> and <argument_name>. For example, the postgresql_where argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

kwargs
A synonym for DialectKWArgs.dialect_kwargs.

class sqlalchemy.sql.expression.**Extract**(*field*, *expr*, ***kwargs*)
Represent a SQL EXTRACT clause, extract(field FROM expr).

class sqlalchemy.sql.elements.**False_**
Represent the false keyword, or equivalent, in a SQL statement.

False_ is accessed as a constant via the false() function.

class sqlalchemy.sql.expression.**FunctionFilter**(*func*, **criterion*)
Represent a function FILTER clause.

This is a special operator against aggregate and window functions, which controls which rows are passed to it. It's supported only by certain database backends.

Invocation of FunctionFilter is via FunctionElement.filter():

```
func.count(1).filter(True)
```

New in version 1.0.0.

See also:

FunctionElement.filter()

filter(**criterion*)
Produce an additional FILTER against the function.

This method adds additional criteria to the initial criteria set up by FunctionElement.filter().

Multiple criteria are joined together at SQL render time via AND.

over(*partition_by=None*, *order_by=None*)
Produce an OVER clause against this filtered function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

The expression:

```
func.rank().filter(MyClass.y > 5).over(order_by='x')
```

is shorthand for:

```
from sqlalchemy import over, funcfilter
over(funcfilter(func.rank(), MyClass.y > 5), order_by='x')
```

See over() for a full description.

class sqlalchemy.sql.expression.**Label**(*name*, *element*, *type_=None*)
Represents a column label (AS).

Represent a label, as typically applied to any column-level element using the AS sql keyword.

class sqlalchemy.sql.elements.**Null**
Represent the NULL keyword in a SQL statement.

Null is accessed as a constant via the null() function.

class sqlalchemy.sql.expression.**Over**(*element*, *partition_by=None*, *order_by=None*, *range_=None*, *rows=None*)
Represent an OVER clause.

This is a special operator against a so-called "window" function, as well as any aggregate function, which produces results relative to the result set itself. It's supported only by certain database backends.

func
the element referred to by this `Over` clause.

Deprecated since version 1.1: the `func` element has been renamed to `.element`. The two attributes are synonymous though `.func` is read-only.

class sqlalchemy.sql.expression.**TextClause**(*text*, *bind=None*)
Represent a literal SQL text fragment.

E.g.:

```
from sqlalchemy import text

t = text("SELECT * FROM users")
result = connection.execute(t)
```

The `Text` construct is produced using the `text()` function; see that function for full documentation.

See also:

`text()`

bindparams(**binds*, ***names_to_values*)
Establish the values and/or types of bound parameters within this `TextClause` construct.

Given a text construct such as:

```
from sqlalchemy import text
stmt = text("SELECT id, name FROM user WHERE name=:name "
            "AND timestamp=:timestamp")
```

the `TextClause.bindparams()` method can be used to establish the initial value of `:name` and `:timestamp`, using simple keyword arguments:

```
stmt = stmt.bindparams(name='jack',
            timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5))
```

Where above, new `BindParameter` objects will be generated with the names `name` and `timestamp`, and values of `jack` and `datetime.datetime(2012, 10, 8, 15, 12, 5)`, respectively. The types will be inferred from the values given, in this case `String` and `DateTime`.

When specific typing behavior is needed, the positional `*binds` argument can be used in which to specify `bindparam()` constructs directly. These constructs must include at least the `key` argument, then an optional value and type:

```
from sqlalchemy import bindparam
stmt = stmt.bindparams(
                bindparam('name', value='jack', type_=String),
                bindparam('timestamp', type_=DateTime)
            )
```

Above, we specified the type of `DateTime` for the `timestamp` bind, and the type of `String` for the `name` bind. In the case of `name` we also set the default value of `"jack"`.

Additional bound parameters can be supplied at statement execution time, e.g.:

```
result = connection.execute(stmt,
            timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5))
```

The `TextClause.bindparams()` method can be called repeatedly, where it will re-use existing `BindParameter` objects to add new information. For example, we can call `TextClause.bindparams()` first with typing information, and a second time with value information, and it will be combined:

```
stmt = text("SELECT id, name FROM user WHERE name=:name "
            "AND timestamp=:timestamp")
stmt = stmt.bindparams(
    bindparam('name', type_=String),
    bindparam('timestamp', type_=DateTime)
)
stmt = stmt.bindparams(
    name='jack',
    timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5)
)
```

New in version 0.9.0: The `TextClause.bindparams()` method supersedes the argument `bindparams` passed to `text()`.

columns(*selectable*, **cols*, ***types*)

Turn this `TextClause` object into a `TextAsFrom` object that can be embedded into another statement.

This function essentially bridges the gap between an entirely textual SELECT statement and the SQL expression language concept of a "selectable":

```
from sqlalchemy.sql import column, text

stmt = text("SELECT id, name FROM some_table")
stmt = stmt.columns(column('id'), column('name')).alias('st')

stmt = select([mytable]).                    select_from(
            mytable.join(stmt, mytable.c.name == stmt.c.name)
        ).where(stmt.c.id > 5)
```

Above, we pass a series of `column()` elements to the `TextClause.columns()` method positionally. These `column()` elements now become first class elements upon the `TextAsFrom.c` column collection, just like any other selectable.

The column expressions we pass to `TextClause.columns()` may also be typed; when we do so, these `TypeEngine` objects become the effective return type of the column, so that SQLAlchemy's result-set-processing systems may be used on the return values. This is often needed for types such as date or boolean types, as well as for unicode processing on some dialect configurations:

```
stmt = text("SELECT id, name, timestamp FROM some_table")
stmt = stmt.columns(
            column('id', Integer),
            column('name', Unicode),
            column('timestamp', DateTime)
        )

for id, name, timestamp in connection.execute(stmt):
    print(id, name, timestamp)
```

As a shortcut to the above syntax, keyword arguments referring to types alone may be used, if only type conversion is needed:

```
stmt = text("SELECT id, name, timestamp FROM some_table")
stmt = stmt.columns(
            id=Integer,
            name=Unicode,
            timestamp=DateTime
        )

for id, name, timestamp in connection.execute(stmt):
    print(id, name, timestamp)
```

The positional form of `TextClause.columns()` also provides the unique feature of **positional column targeting**, which is particularly useful when using the ORM with complex textual queries. If we specify the columns from our model to `TextClause.columns()`, the result set will match to those columns positionally, meaning the name or origin of the column in the textual SQL doesn't matter:

```
stmt = text("SELECT users.id, addresses.id, users.id, "
     "users.name, addresses.email_address AS email "
     "FROM users JOIN addresses ON users.id=addresses.user_id "
     "WHERE users.id = 1").columns(
        User.id,
        Address.id,
        Address.user_id,
        User.name,
        Address.email_address
     )

query = session.query(User).from_statement(stmt).options(
    contains_eager(User.addresses))
```

New in version 1.1: the `TextClause.columns()` method now offers positional column targeting in the result set when the column expressions are passed purely positionally.

The `TextClause.columns()` method provides a direct route to calling `FromClause.alias()` as well as `SelectBase.cte()` against a textual SELECT statement:

```
stmt = stmt.columns(id=Integer, name=String).cte('st')

stmt = select([sometable]).where(sometable.c.id == stmt.c.id)
```

New in version 0.9.0: `text()` can now be converted into a fully featured "selectable" construct using the `TextClause.columns()` method. This method supersedes the `typemap` argument to `text()`.

class sqlalchemy.sql.expression.**Tuple**(**clauses*, ***kw*)
Represent a SQL tuple.

class sqlalchemy.sql.expression.**WithinGroup**(*element*, **order_by*)
Represent a WITHIN GROUP (ORDER BY) clause.

This is a special operator against so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including `percentile_cont()`, `rank()`, `dense_rank()`, etc.

It's supported only by certain database backends, such as PostgreSQL, Oracle and MS SQL Server.

The `WithinGroup` construct extracts its type from the method `FunctionElement.within_group_type()`. If this returns `None`, the function's `.type` is used.

over(*partition_by=None*, *order_by=None*)
Produce an OVER clause against this `WithinGroup` construct.

This function has the same signature as that of `FunctionElement.over()`.

class sqlalchemy.sql.elements.**True_**
Represent the `true` keyword, or equivalent, in a SQL statement.

`True_` is accessed as a constant via the `true()` function.

class sqlalchemy.sql.expression.**TypeCoerce**(*expression*, *type_*)
Represent a Python-side type-coercion wrapper.

`TypeCoerce` supplies the `expression.type_coerce()` function; see that function for usage details.

Changed in version 1.1: The `type_coerce()` function now produces a persistent `TypeCoerce` wrapper object rather than translating the given object in place.

See also:

expression.type_coerce()

class sqlalchemy.sql.operators.custom_op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*, *natural_self_precedent=False*, *eager_grouping=False*)

Represent a 'custom' operator.

custom_op is normally instantiated when the Operators.op() or Operators.bool_op() methods are used to create a custom operator callable. The class can also be used directly when programmatically constructing expressions. E.g. to represent the "factorial" operation:

```
from sqlalchemy.sql import UnaryExpression
from sqlalchemy.sql import operators
from sqlalchemy import Numeric

unary = UnaryExpression(table.c.somecolumn,
        modifier=operators.custom_op("!"),
        type_=Numeric)
```

See also:

Operators.op()

Operators.bool_op()

class sqlalchemy.sql.operators.Operators

Base of comparison and logical operators.

Implements base methods operate() and reverse_operate(), as well as __and__(), __or__(), __invert__().

Usually is used via its most common subclass ColumnOperators.

__and__(*other*)

Implement the & operator.

When used with SQL expressions, results in an AND operation, equivalent to and_(), that is:

```
a & b
```

is equivalent to:

```
from sqlalchemy import and_
and_(a, b)
```

Care should be taken when using & regarding operator precedence; the & operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) & (b == 4)
```

__invert__()

Implement the ~ operator.

When used with SQL expressions, results in a NOT operation, equivalent to not_(), that is:

```
~a
```

is equivalent to:

```
from sqlalchemy import not_
not_(a)
```

__or__(*other*)
Implement the | operator.

When used with SQL expressions, results in an OR operation, equivalent to or_(), that is:

```
a | b
```

is equivalent to:

```
from sqlalchemy import or_
or_(a, b)
```

Care should be taken when using | regarding operator precedence; the | operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) | (b == 4)
```

bool_op(*opstring*, *precedence=0*)
Return a custom boolean operator.

This method is shorthand for calling Operators.op() and passing the Operators.op.is_comparison flag with True.

New in version 1.2.0b3.

See also:

Operators.op()

op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of 0 is lower than all operators except for the comma (,) and AS operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the Operators.op.is_comparison flag.

- **return_type** – a TypeEngine class or object that will force the return type of an expression produced by this operator to be of that type. By default, operators that specify Operators.op.is_comparison will resolve to Boolean, and those that do not will be of the same type as the left-hand operand.

  New in version 1.2.0b3: - added the Operators.op.return_type argument.

See also:

types_operators

relationship_custom_operator

operate(*op*, **other*, ***kwargs*)
Operate on an argument.

This is the lowest level of operation, raises NotImplementedError by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding ColumnOperators to apply func.lower() to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as ColumnOperators.contains().

reverse_operate(*op*, *other*, ***kwargs*)
Reverse operate on an argument.

Usage is the same as operate().

class sqlalchemy.sql.elements.quoted_name
Represent a SQL identifier combined with quoting preferences.

quoted_name is a Python unicode/str subclass which represents a particular identifier name along with a quote flag. This quote flag, when set to True or False, overrides automatic quoting behavior for this identifier in order to either unconditionally quote or to not quote the name. If left at its default of None, quoting behavior is applied to the identifier on a per-backend basis based on an examination of the token itself.

A quoted_name object with quote=True is also prevented from being modified in the case of a so-called "name normalize" option. Certain database backends, such as Oracle, Firebird, and DB2 "normalize" case-insensitive names as uppercase. The SQLAlchemy dialects for these backends convert from SQLAlchemy's lower-case-means-insensitive convention to the upper-case-means-insensitive conventions of those backends. The quote=True flag here will prevent this conversion from occurring to support an identifier that's quoted as all lower case against such a backend.

The quoted_name object is normally created automatically when specifying the name for key schema constructs such as Table, Column, and others. The class can also be passed explicitly as the name to any function that receives a name which can be quoted. Such as to use the Engine.has_table() method with an unconditionally quoted name:

```
from sqlalchemy import create_engine
from sqlalchemy.sql import quoted_name

engine = create_engine("oracle+cx_oracle://some_dsn")
engine.has_table(quoted_name("some_table", True))
```

The above logic will run the "has table" logic against the Oracle backend, passing the name exactly as `"some_table"` without converting to upper case.

New in version 0.9.0.

Changed in version 1.2: The `quoted_name` construct is now importable from `sqlalchemy.sql`, in addition to the previous location of `sqlalchemy.sql.elements`.

`class sqlalchemy.sql.expression.UnaryExpression(`*element*, *operator=None*, *modifier=None*, *type_=None*, *negate=None*, *wraps_column_expression=False*`)`

Define a 'unary' expression.

A unary expression has a single column expression and an operator. The operator can be placed on the left (where it is called the 'operator') or right (where it is called the 'modifier') of the column expression.

`UnaryExpression` is the basis for several unary operators including those used by `desc()`, `asc()`, `distinct()`, `nullsfirst()` and `nullslast()`.

`compare(`*other*, ***kw*`)`

Compare this `UnaryExpression` against the given `ClauseElement`.

3.2.2 Selectables, Tables, FROM objects

The term "selectable" refers to any object that rows can be selected from; in SQLAlchemy, these objects descend from `FromClause` and their distinguishing feature is their `FromClause.c` attribute, which is a namespace of all the columns contained within the FROM clause (these elements are themselves `ColumnElement` subclasses).

`sqlalchemy.sql.expression.alias(`*selectable*, *name=None*, *flat=False*`)`

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.

When an `Alias` is created from a `Table` object, 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 `name` parameter is optional, and provides the name to use in the rendered SQL. If blank, an "anonymous" name will be deterministically generated at compile time. Deterministic means the name is guaranteed to be unique against other constructs used in the same statement, and will also be the same name for each successive compilation of the same statement object.

Parameters

- **`selectable`** – any `FromClause` subclass, such as a table, select statement, etc.
- **`name`** – string name to be assigned as the alias. If `None`, a name will be deterministically generated at compile time.
- **`flat`** – Will be passed through to if the given selectable is an instance of `Join` - see `Join.alias()` for details.

  New in version 0.9.0.

`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*)
Construct a new `Exists` against an existing `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.**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*, *full=False*)
Produce a `Join` object, given two `FromClause` expressions.

E.g.:

```
j = join(user_table, address_table,
         user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Similar functionality is available given any `FromClause` object (e.g. such as a `Table`) using the `FromClause.join()` method.

Parameters

- **left** – The left side of the join.
- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.

- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of JOIN.

  New in version 1.1.

See also:

FromClause.join() - method form, based on a given left side

Join - the type of object produced

sqlalchemy.sql.expression.**lateral**(*selectable*, *name=None*)

Return a Lateral object.

Lateral is an Alias subclass that represents a subquery with the LATERAL keyword applied to it.

The special behavior of a LATERAL subquery is that it appears in the FROM clause of an enclosing SELECT, but may correlate to other FROM clauses of that SELECT. It is a special case of subquery only supported by a small number of backends, currently more recent PostgreSQL versions.

New in version 1.1.

See also:

lateral_selects - overview of usage.

sqlalchemy.sql.expression.**outerjoin**(*left*, *right*, *onclause=None*, *full=False*)

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.

Parameters

- **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 FromClause.join() or FromClause.outerjoin() methods on the resulting Join object.

sqlalchemy.sql.expression.**select**(*columns=None*, *whereclause=None*, *from_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, ***kwargs*)

Construct a new Select.

Similar functionality is also available via the FromClause.select() method on any FromClause.

All arguments which accept ClauseElement arguments also accept string arguments, which will be converted as appropriate into either text() or literal_column() constructs.

See also:

coretutorial_selecting - Core Tutorial description of select().

Parameters

- **columns** – A list of ColumnElement or FromClause objects which will form the columns clause of the resulting statement. For those objects that are instances of FromClause (typically Table or Alias objects), the FromClause.c collection is extracted to form a collection of ColumnElement objects.

This parameter will also accept `Text` constructs as given, as well as ORM-mapped classes.

Note: The `select.columns` parameter is not available in the method form of `select()`, e.g. `FromClause.select()`.

See also:

`Select.column()`

`Select.with_only_columns()`

- **`whereclause`** – A `ClauseElement` expression which will be used to form the `WHERE` clause. It is typically preferable to add WHERE criterion to an existing `Select` using method chaining with `Select.where()`.

  See also:

  `Select.where()`

- **`from_obj`** – A list of `ClauseElement` objects which will be added to the `FROM` clause of the resulting statement. This is equivalent to calling `Select.select_from()` using method chaining on an existing `Select` object.

  See also:

  `Select.select_from()` - full description of explicit FROM clause specification.

- **`autocommit`** – Deprecated. Use `.execution_options(autocommit=<True|False>)` to set the autocommit option.

  See also:

  `Executable.execution_options()`

- **`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.

- **`correlate=True`** – indicates that this `Select` object should have its contained `FromClause` elements "correlated" to an enclosing `Select` object. It is typically preferable to specify correlations on an existing `Select` construct using `Select.correlate()`.

  See also:

  `Select.correlate()` - full description of correlation.

- **`distinct=False`** – when `True`, applies a `DISTINCT` qualifier to the columns clause of the resulting statement.

  The boolean argument may also be a column expression or list of column expressions - this is a special calling form which is understood by the PostgreSQL dialect to render the `DISTINCT ON (<columns>)` syntax.

  `distinct` is also available on an existing `Select` object via the `distinct()` method.

  See also:

  `Select.distinct()`

- **`for_update=False`** –

  when `True`, applies `FOR UPDATE` to the end of the resulting statement.

  Deprecated since version 0.9.0: - use `Select.with_for_update()` to specify the structure of the `FOR UPDATE` clause.

for_update accepts various string values interpreted by specific backends, including:

- "read" - on MySQL, translates to LOCK IN SHARE MODE; on PostgreSQL, translates to FOR SHARE.
- "nowait" - on PostgreSQL and Oracle, translates to FOR UPDATE NOWAIT.
- "read_nowait" - on PostgreSQL, translates to FOR SHARE NOWAIT.

See also:

Select.with_for_update() - improved API for specifying the FOR UPDATE clause.

- **group_by** – a list of ClauseElement objects which will comprise the GROUP BY clause of the resulting select. This parameter is typically specified more naturally using the Select.group_by() method on an existing Select.

  See also:

  Select.group_by()

- **having** – a ClauseElement that will comprise the HAVING clause of the resulting select when GROUP BY is used. This parameter is typically specified more naturally using the Select.having() method on an existing Select.

  See also:

  Select.having()

- **limit=None** – a numerical value which usually renders as a LIMIT expression in the resulting select. Backends that don't support LIMIT will attempt to provide similar functionality. This parameter is typically specified more naturally using the Select.limit() method on an existing Select.

  See also:

  Select.limit()

- **offset=None** – a numeric value which usually renders as an OFFSET expression in the resulting select. Backends that don't support OFFSET will attempt to provide similar functionality. This parameter is typically specified more naturally using the Select.offset() method on an existing Select.

  See also:

  Select.offset()

- **order_by** – a scalar or list of ClauseElement objects which will comprise the ORDER BY clause of the resulting select. This parameter is typically specified more naturally using the Select.order_by() method on an existing Select.

  See also:

  Select.order_by()

- **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.

  This parameter can also be specified on an existing Select object using the Select.apply_labels() method.

  See also:

  Select.apply_labels()

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*)
Produce a new TableClause.

The object returned is an instance of TableClause, which represents the "syntactical" portion of the schema-level Table object. It may be used to construct lightweight table constructs.

Changed in version 1.0.0: expression.table() can now be imported from the plain sqlalchemy namespace like any other SQL element.

Parameters

- **name** – Name of the table.
- **columns** – A collection of expression.column() constructs.

sqlalchemy.sql.expression.**tablesample**(*selectable*, *sampling*, *name=None*, *seed=None*)
Return a TableSample object.

TableSample is an Alias subclass that represents a table with the TABLESAMPLE clause applied to it. tablesample() is also available from the FromClause class via the FromClause.tablesample() method.

The TABLESAMPLE clause allows selecting a randomly selected approximate percentage of rows from a table. It supports multiple sampling methods, most commonly BERNOULLI and SYSTEM.

e.g.:

```
from sqlalchemy import func

selectable = people.tablesample(
            func.bernoulli(1),
            name='alias',
            seed=func.random())
stmt = select([selectable.c.people_id])
```

Assuming people with a column people_id, the above statement would render as:

```
SELECT alias.people_id FROM
people AS alias TABLESAMPLE bernoulli(:bernoulli_1)
REPEATABLE (random())
```

New in version 1.1.

Parameters

- **sampling** – a float percentage between 0 and 100 or functions.Function.
- **name** – optional alias name
- **seed** – any real-valued SQL expression. When specified, the REPEATABLE sub-clause is also rendered.

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()`.

`class sqlalchemy.sql.expression.Alias(selectable, name=None)`
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 `FromClause.alias()` method available on all `FromClause` subclasses.

`alias(name=None, flat=False)`
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

`c`
An alias for the `columns` attribute.

`columns`
A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

`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(default, 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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

count(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

join(*right*, *onclause=None*, *isouter=False*, *full=False*)
Return a Join from this FromClause to another FromClause.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any FromClause object such as a Table object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies FromClause.join.isouter.

  New in version 1.1.

See also:

join() - standalone function

Join - the type of object produced

lateral(*name=None*)
Return a LATERAL alias of this FromClause.

The return value is the Lateral construct also provided by the top-level lateral() function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

outerjoin(*right*, *onclause=None*, *full=False*)
Return a Join from this FromClause to another FromClause, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

params(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

select(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

unique_params(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

`class sqlalchemy.sql.expression.CompoundSelect(`*keyword*, **selects*, ***kwargs*`)`

Forms the basis of `UNION`, `UNION ALL`, and other SELECT-based set operations.

See also:

`union()`

`union_all()`

`intersect()`

`intersect_all()`

`except()`

`except_all()`

`alias(`*name=None*, *flat=False*`)`
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

`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.

This is an **in-place** mutation method; the `group_by()` method is preferred, as it provides standard method chaining.

`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.

This is an **in-place** mutation method; the `order_by()` method is preferred, as it provides standard method chaining.

`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 True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

`c`
An alias for the `columns` attribute.

columns
A named-based collection of ColumnElement objects maintained by this FromClause.

The columns, or c collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`cte`(*name=None*, *recursive=False*)
Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **`name`** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **`recursive`** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
```

```
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

description
a brief description of this FromClause.

Used primarily for error message formatting.

execute(**multiparams*, ***params*)
Compile and execute this `Executable`.

execution_options(***kw*)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

for_update
Provide legacy dialect support for the `for_update` attribute.

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

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.

join(*right*, *onclause=None*, *isouter=False*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`label`(*name*)
return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

`as_scalar()`.

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`limit`(*limit*)
return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a `LIMIT` expression in the resulting select. Backends that don't support `LIMIT` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.limit()` can now accept arbitrary SQL expressions as well as integer values.

Parameters `limit` – an integer LIMIT parameter, or a SQL expression that provides an integer result.

`offset`(*offset*)
return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an `OFFSET` expression in the resulting select. Backends that don't support `OFFSET` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.offset()` can now accept arbitrary SQL expressions as well as integer values.

Parameters `offset` – an integer OFFSET parameter, or a SQL expression that provides an integer result.

`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.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any FromClause object such as a Table object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

FromClause.join()

Join

params(**optionaldict*, ***kwargs*)
Return a copy with bindparam() elements replaced.

Returns a copy of this ClauseElement with bindparam() elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this FromClause.

scalar(**multiparams*, ***params*)
Compile and execute this Executable, returning the result's scalar representation.

select(*whereclause=None*, ***params*)
return a SELECT of this FromClause.

See also:

select() - general purpose method which allows for arbitrary column lists.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this FromClause.

The return value is the TableSample construct also provided by the top-level tablesample() function.

New in version 1.1.

See also:

tablesample() - usage guidelines and parameters

unique_params(**optionaldict*, ***kwargs*)
Return a copy with bindparam() elements replaced.

Same functionality as params(), except adds *unique=True* to affected bind parameters so that multiple statements can be used.

with_for_update(*nowait=False*, *read=False*, *of=None*, *skip_locked=False*, *key_share=False*)

Specify a `FOR UPDATE` clause for this `GenerativeSelect`.

E.g.:

```
stmt = select([table]).with_for_update(nowait=True)
```

On a database like PostgreSQL or Oracle, the above would render a statement like:

```
SELECT table.a, table.b FROM table FOR UPDATE NOWAIT
```

on other backends, the `nowait` option is ignored and instead would produce:

```
SELECT table.a, table.b FROM table FOR UPDATE
```

When called with no arguments, the statement will render with the suffix `FOR UPDATE`. Additional arguments can then be provided which allow for common database-specific variants.

Parameters

- **nowait** – boolean; will render `FOR UPDATE NOWAIT` on Oracle and PostgreSQL dialects.
- **read** – boolean; will render `LOCK IN SHARE MODE` on MySQL, `FOR SHARE` on PostgreSQL. On PostgreSQL, when combined with `nowait`, will render `FOR SHARE NOWAIT`.
- **of** – SQL expression or list of SQL expression elements (typically `Column` objects or a compatible expression) which will render into a `FOR UPDATE OF` clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend.
- **skip_locked** – boolean, will render `FOR UPDATE SKIP LOCKED` on Oracle and PostgreSQL dialects or `FOR SHARE SKIP LOCKED` if `read=True` is also specified.

  New in version 1.1.0.
- **key_share** – boolean, will render `FOR NO KEY UPDATE`, or if combined with `read=True` will render `FOR KEY SHARE`, on the PostgreSQL dialect.

  New in version 1.1.0.

class sqlalchemy.sql.expression.CTE(*selectable*, *name=None*, *recursive=False*, *_cte_alias=None*, *_restates=frozenset()*, *_suffixes=None*)

Represent a Common Table Expression.

The `CTE` object is obtained using the `SelectBase.cte()` method from any selectable. See that method for complete examples.

New in version 0.7.6.

c

An alias for the `columns` attribute.

columns

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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`(*default*, *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 from 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*.
- **`compile_kwargs`** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`foreign_keys`
Return the collection of ForeignKey objects which this FromClause references.

`join`(*right*, *onclause=None*, *isouter=False*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **`right`** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **`onclause`** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **`isouter`** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **`full`** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **`right`** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **`onclause`** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **`full`** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

`params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

`primary_key`
Return the collection of Column objects which comprise the primary key of this FromClause.

`replace_selectable`(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

`select`(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

suffix_with(**expr*, ***kw*)
Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

```
stmt = select([col1, col2]).cte().suffix_with(
    "cycle empno set y_cycle to 1 default 0", dialect="oracle")
```

Multiple suffixes can be specified by multiple calls to `suffix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the target clause.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

unique_params(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**Executable**
Mark a ClauseElement as supporting execution.

`Executable` is a superclass for all "statement" types of objects, including `select()`, `delete()`, `update()`, `insert()`, `text()`.

bind
Returns the `Engine` or `Connection` to which this `Executable` is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

execute(**multiparams*, ***params*)
Compile and execute this `Executable`.

execution_options(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

`scalar`(**multiparams*, ***params*)
Compile and execute this `Executable`, returning the result's scalar representation.

`class sqlalchemy.sql.expression.FromClause`
Represent an element that can be used within the `FROM` clause of a `SELECT` statement.

The most common forms of `FromClause` are the `Table` and the `select()` constructs. Key features common to all `FromClause` objects include:

- a `c` collection, which provides per-name access to a collection of `ColumnElement` objects.
- a `primary_key` attribute, which is a collection of all those `ColumnElement` objects that indicate the `primary_key` flag.
- Methods to generate various derivations of a "from" clause, including `FromClause.alias()`, `FromClause.join()`, `FromClause.select()`.

`alias`(*name=None*, *flat=False*)
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

`c`
An alias for the `columns` attribute.

`columns`
A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

`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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

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*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

lateral(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

`primary_key`
Return the collection of Column objects which comprise the primary key of this FromClause.

`replace_selectable`(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

`schema = None`
Define the 'schema' attribute for this `FromClause`.

This is typically `None` for most objects except that of `Table`, where it is taken as the value of the `Table.schema` argument.

`select`(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

`tablesample`(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

class sqlalchemy.sql.expression.**GenerativeSelect**(*use_labels=False*, *for_update=False*, *limit=None*, *offset=None*, *order_by=None*, *group_by=None*, *bind=None*, *autocommit=None*)

Base class for SELECT statements where additional elements can be added.

This serves as the base for `Select` and `CompoundSelect` where elements such as ORDER BY, GROUP BY can be added and column rendering can be controlled. Compare to `TextAsFrom`, which, while it subclasses `SelectBase` and is also a SELECT construct, represents a fixed textual string which cannot be altered at this level, only wrapped as a subquery.

New in version 0.9.0: `GenerativeSelect` was added to provide functionality specific to `Select` and `CompoundSelect` while allowing `SelectBase` to be used for other SELECT-like objects, e.g. `TextAsFrom`.

alias(*name=None*, *flat=False*)

return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

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.

This is an **in-place** mutation method; the `group_by()` method is preferred, as it provides standard method chaining.

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.

This is an **in-place** mutation method; the `order_by()` method is preferred, as it provides standard method chaining.

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 True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

bind
Returns the `Engine` or `Connection` to which this `Executable` is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

c
An alias for the `columns` attribute.

columns
A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)
```

```
print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`cte`(*name=None*, *recursive=False*)
Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
```

```
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

`description`
a brief description of this FromClause.

Used primarily for error message formatting.

`execute`(**multiparams*, ***params*)
Compile and execute this `Executable`.

`execution_options`(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

`for_update`
Provide legacy dialect support for the `for_update` attribute.

`foreign_keys`
Return the collection of ForeignKey objects which this FromClause references.

`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).

`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.

`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*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
```

```
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`label`(*name*)
return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

`as_scalar()`.

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`limit`(*limit*)
return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a `LIMIT` expression in the resulting select. Backends that don't support `LIMIT` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.limit()` can now accept arbitrary SQL expressions as well as integer values.

Parameters **limit** – an integer LIMIT parameter, or a SQL expression that provides an integer result.

`offset`(*offset*)
return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an `OFFSET` expression in the resulting select. Backends that don't support `OFFSET` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.offset()` can now accept arbitrary SQL expressions as well as integer values.

Parameters **offset** – an integer OFFSET parameter, or a SQL expression that provides an integer result.

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.

outerjoin(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

params(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

`scalar`(**multiparams*, ***params*)
Compile and execute this `Executable`, returning the result's scalar representation.

`select`(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

`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.

`tablesample`(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

`unique_params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

`with_for_update`(*nowait=False*, *read=False*, *of=None*, *skip_locked=False*, *key_share=False*)
Specify a `FOR UPDATE` clause for this `GenerativeSelect`.

E.g.:

```
stmt = select([table]).with_for_update(nowait=True)
```

On a database like PostgreSQL or Oracle, the above would render a statement like:

```
SELECT table.a, table.b FROM table FOR UPDATE NOWAIT
```

on other backends, the `nowait` option is ignored and instead would produce:

```
SELECT table.a, table.b FROM table FOR UPDATE
```

When called with no arguments, the statement will render with the suffix `FOR UPDATE`. Additional arguments can then be provided which allow for common database-specific variants.

Parameters

- **nowait** – boolean; will render `FOR UPDATE NOWAIT` on Oracle and PostgreSQL dialects.

- **read** – boolean; will render `LOCK IN SHARE MODE` on MySQL, `FOR SHARE` on PostgreSQL. On PostgreSQL, when combined with `nowait`, will render `FOR SHARE NOWAIT`.
- **of** – SQL expression or list of SQL expression elements (typically `Column` objects or a compatible expression) which will render into a `FOR UPDATE OF` clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend.
- **skip_locked** – boolean, will render `FOR UPDATE SKIP LOCKED` on Oracle and PostgreSQL dialects or `FOR SHARE SKIP LOCKED` if `read=True` is also specified.

  New in version 1.1.0.
- **key_share** – boolean, will render `FOR NO KEY UPDATE`, or if combined with `read=True` will render `FOR KEY SHARE`, on the PostgreSQL dialect.

  New in version 1.1.0.

class sqlalchemy.sql.expression.**HasCTE**

Mixin that declares a class to include CTE support.

New in version 1.1.

cte(*name=None*, *recursive=False*)

Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
```

```
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
```

```
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

class sqlalchemy.sql.expression.**HasPrefixes**

prefix_with(**expr*, ***kw*)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

class sqlalchemy.sql.expression.**HasSuffixes**

suffix_with(**expr*, ***kw*)

Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

```
stmt = select([col1, col2]).cte().suffix_with(
    "cycle empno set y_cycle to 1 default 0", dialect="oracle")
```

Multiple suffixes can be specified by multiple calls to `suffix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the target clause.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect.

class sqlalchemy.sql.expression.**Join**(*left*, *right*, *onclause=None*, *isouter=False*, *full=False*)

represent a `JOIN` construct between two `FromClause` elements.

The public constructor function for `Join` is the module-level `join()` function, as well as the `FromClause.join()` method of any `FromClause` (e.g. such as `Table`).

See also:

`join()`

`FromClause.join()`

alias(*sqlutil*, *name=None*, *flat=False*)

return an alias of this `Join`.

The default behavior here is to first produce a SELECT construct from this `Join`, then to produce an `Alias` from that. So given a join of the form:

```
j = table_a.join(table_b, table_a.c.id == table_b.c.a_id)
```

The JOIN by itself would look like:

```
table_a JOIN table_b ON table_a.id = table_b.a_id
```

Whereas the alias of the above, `j.alias()`, would in a SELECT context look like:

```
(SELECT table_a.id AS table_a_id, table_b.id AS table_b_id,
    table_b.a_id AS table_b_a_id
    FROM table_a
    JOIN table_b ON table_a.id = table_b.a_id) AS anon_1
```

The equivalent long-hand form, given a `Join` object `j`, is:

```
from sqlalchemy import select, alias
j = alias(
    select([j.left, j.right]).\
        select_from(j).\
        with_labels(True).\
```

```
        correlate(False),
    name=name
)
```

The selectable produced by `Join.alias()` features the same columns as that of the two individual selectables presented under a single name - the individual columns are "auto-labeled", meaning the `.c.` collection of the resulting `Alias` represents the names of the individual columns using a `<tablename>_<columnname>` scheme:

```
j.c.table_a_id
j.c.table_b_a_id
```

`Join.alias()` also features an alternate option for aliasing joins which produces no enclosing SELECT and does not normally apply labels to the column names. The `flat=True` option will call `FromClause.alias()` against the left and right sides individually. Using this option, no new `SELECT` is produced; we instead, from a construct as below:

```
j = table_a.join(table_b, table_a.c.id == table_b.c.a_id)
j = j.alias(flat=True)
```

we get a result like this:

```
table_a AS table_a_1 JOIN table_b AS table_b_1 ON
table_a_1.id = table_b_1.a_id
```

The `flat=True` argument is also propagated to the contained selectables, so that a composite join such as:

```
j = table_a.join(
        table_b.join(table_c,
                table_b.c.id == table_c.c.b_id),
        table_b.c.a_id == table_a.c.id
    ).alias(flat=True)
```

Will produce an expression like:

```
table_a AS table_a_1 JOIN (
        table_b AS table_b_1 JOIN table_c AS table_c_1
        ON table_b_1.id = table_c_1.b_id
) ON table_a_1.id = table_b_1.a_id
```

The standalone `alias()` function as well as the base `FromClause.alias()` method also support the `flat=True` argument as a no-op, so that the argument can be passed to the `alias()` method of any selectable.

New in version 0.9.0: Added the `flat=True` option to create "aliases" of joins without enclosing inside of a SELECT subquery.

Parameters

- **`name`** – name given to the alias.
- **`flat`** – if True, produce an alias of the left and right sides of this `Join` and return the join of those two selectables. This produces join expression that does not include an enclosing SELECT.

  New in version 0.9.0.

See also:

`alias()`

`c`

An alias for the `columns` attribute.

columns
A named-based collection of ColumnElement objects maintained by this FromClause.

The columns, or c collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`foreign_keys`
Return the collection of ForeignKey objects which this FromClause references.

`join`(*right*, *onclause=None*, *isouter=False*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

`params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this FromClause.

select(*whereclause=None*, ***kwargs*)
Create a Select from this Join.

The equivalent long-hand form, given a Join object j, is:

```
from sqlalchemy import select
j = select([j.left, j.right], **kw).\
            where(whereclause).\
            select_from(j)
```

Parameters

- **whereclause** – the WHERE criterion that will be sent to the select() function
- ****kwargs** – all other kwargs are sent to the underlying select() function.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this FromClause.

The return value is the TableSample construct also provided by the top-level tablesample() function.

New in version 1.1.

See also:

tablesample() - usage guidelines and parameters

unique_params(**optionaldict*, ***kwargs*)
Return a copy with bindparam() elements replaced.

Same functionality as params(), except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**Lateral**(*selectable*, *name=None*)
Represent a LATERAL subquery.

This object is constructed from the lateral() module level function as well as the FromClause.lateral() method available on all FromClause subclasses.

While LATERAL is part of the SQL standard, curently only more recent PostgreSQL versions provide support for this keyword.

New in version 1.1.

See also:

lateral_selects - overview of usage.

alias(*name=None*, *flat=False*)
return an alias of this FromClause.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See alias() for details.

c
An alias for the columns attribute.

columns
A named-based collection of ColumnElement objects maintained by this FromClause.

The columns, or c collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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 ancestor 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 ancestor with one of the exported columns of this FromClause.

count(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this FromClause.

Deprecated since version 1.1: FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of func.count() should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

func

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

join(*right*, *onclause=None*, *isouter=False*, *full=False*)
Return a Join from this FromClause to another FromClause.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any FromClause object such as a Table object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies FromClause.join.isouter.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

`params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this FromClause.

select(*whereclause=None*, ***params*)
return a SELECT of this FromClause.

See also:

select() - general purpose method which allows for arbitrary column lists.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this FromClause.

The return value is the TableSample construct also provided by the top-level tablesample() function.

New in version 1.1.

See also:

tablesample() - usage guidelines and parameters

unique_params(**optionaldict*, ***kwargs*)
Return a copy with bindparam() elements replaced.

Same functionality as params(), except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**ScalarSelect**(*element*)

where(*crit*)
Apply a WHERE clause to the SELECT statement referred to by this ScalarSelect.

class sqlalchemy.sql.expression.**Select**(*columns=None*, *whereclause=None*, *from_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, ***kwargs*)
Represents a SELECT statement.

alias(*name=None*, *flat=False*)
return an alias of this FromClause.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See alias() for details.

append_column(*column*)
append the given column expression to the columns clause of this select() construct.

E.g.:

```
my_select.append_column(some_table.c.new_column)
```

This is an **in-place** mutation method; the column() method is preferred, as it provides standard method chaining.

See the documentation for Select.with_only_columns() for guidelines on adding /replacing the columns of a Select object.

`append_correlation`(*fromclause*)
append the given correlation expression to this select() construct.

This is an **in-place** mutation method; the `correlate()` method is preferred, as it provides standard method chaining.

`append_from`(*fromclause*)
append the given FromClause expression to this select() construct's FROM clause.

This is an **in-place** mutation method; the `select_from()` method is preferred, as it provides standard method chaining.

`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.

This is an **in-place** mutation method; the `group_by()` method is preferred, as it provides standard method chaining.

`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.

This is an **in-place** mutation method; the `having()` method is preferred, as it provides standard method chaining.

`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.

This is an **in-place** mutation method; the `order_by()` method is preferred, as it provides standard method chaining.

`append_prefix`(*clause*)
append the given columns clause prefix expression to this select() construct.

This is an **in-place** mutation method; the `prefix_with()` method is preferred, as it provides standard method chaining.

`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.

This is an **in-place** mutation method; the `where()` method is preferred, as it provides standard method chaining.

`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 True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

c

An alias for the `columns` attribute.

column(*column*)

return a new select() construct with the given column expression added to its columns clause.

E.g.:

```
my_select = my_select.column(table.c.new_column)
```

See the documentation for `Select.with_only_columns()` for guidelines on adding /replacing the columns of a `Select` object.

columns

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`correlate`(**fromclauses*)

return a new `Select` which will correlate the given FROM clauses to that of an enclosing `Select`.

Calling this method turns off the `Select` object's default behavior of "auto-correlation". Normally, FROM elements which appear in a `Select` that encloses this one via its WHERE clause, ORDER BY, HAVING or columns clause will be omitted from this `Select` object's FROM clause. Setting an explicit correlation collection using the `Select.correlate()` method provides a fixed list of FROM objects that can potentially take place in this process.

When `Select.correlate()` is used to apply specific FROM clauses for correlation, the FROM elements become candidates for correlation regardless of how deeply nested this `Select` object is, relative to an enclosing `Select` which refers to the same FROM object. This is in contrast to the behavior of "auto-correlation" which only correlates to an immediate enclosing `Select`. Multi-level correlation ensures that the link between enclosed and enclosing `Select` is always via at least one WHERE/ORDER BY/HAVING/columns clause in order for correlation to take place.

If `None` is passed, the `Select` object will correlate none of its FROM entries, and all will render unconditionally in the local FROM clause.

Parameters `*fromclauses` – a list of one or more `FromClause` constructs, or other compatible constructs (i.e. ORM-mapped classes) to become part of the correlate collection.

Changed in version 0.8.0: ORM-mapped classes are accepted by `Select.correlate()`.

Changed in version 0.8.0: The `Select.correlate()` method no longer unconditionally removes entries from the FROM clause; instead, the candidate FROM entries must also be matched by a FROM entry located in an enclosing `Select`, which ultimately encloses this one as present in the WHERE clause, ORDER BY clause, HAVING clause, or columns clause of an enclosing `Select()`.

Changed in version 0.8.2: explicit correlation takes place via any level of nesting of `Select` objects; in previous 0.8 versions, correlation would only occur relative to the immediate enclosing `Select` construct.

See also:

`Select.correlate_except()`

correlated_subqueries

`correlate_except`(**fromclauses*)

return a new `Select` which will omit the given FROM clauses from the auto-correlation process.

Calling `Select.correlate_except()` turns off the `Select` object's default behavior of "auto-correlation" for the given FROM elements. An element specified here will unconditionally appear in the FROM list, while all other FROM elements remain subject to normal auto-correlation behaviors.

Changed in version 0.8.2: The `Select.correlate_except()` method was improved to fully prevent FROM clauses specified here from being omitted from the immediate FROM clause of this `Select`.

If `None` is passed, the `Select` object will correlate all of its FROM entries.

Changed in version 0.8.2: calling `correlate_except(None)` will correctly auto-correlate all FROM clauses.

Parameters `*fromclauses` – a list of one or more `FromClause` constructs, or other compatible constructs (i.e. ORM-mapped classes) to become part of the correlate-exception collection.

See also:

`Select.correlate()`

correlated_subqueries

`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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`cte`(*name=None*, *recursive=False*)
Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **`name`** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **`recursive`** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()
```

```
parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)
```

```
upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

`description`
a brief description of this FromClause.

Used primarily for error message formatting.

`distinct`(**expr*)
Return a new select() construct which will apply DISTINCT to its columns clause.

Parameters `*expr` – optional column expressions. When present, the PostgreSQL dialect will render a `DISTINCT ON (<expressions>>)` construct.

`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.

`execute`(**multiparams*, ***params*)
Compile and execute this `Executable`.

`execution_options`(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include “autocommit” and “stream_results”, but not “isolation_level” or “compiled_cache”. See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

`for_update`
Provide legacy dialect support for the `for_update` attribute.

`foreign_keys`
Return the collection of ForeignKey objects which this FromClause references.

`froms`
Return the displayed list of FromClause elements.

`get_children`(*column_collections=True*, ***kwargs*)
return child elements as per the ClauseElement specification.

`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.

`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.

`join`(*right*, *onclause=None*, *isouter=False*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **`right`** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **`onclause`** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **`isouter`** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **`full`** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`label`(*name*)
return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

`as_scalar()`.

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`limit`(*limit*)
return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a `LIMIT` expression in the resulting select. Backends that don't support `LIMIT` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.limit()` can now accept arbitrary SQL expressions as well as integer values.

Parameters `limit` – an integer LIMIT parameter, or a SQL expression that provides an integer result.

`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.

`offset`(*offset*)
return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an `OFFSET` expression in the resulting select. Backends that don't support `OFFSET` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.offset()` can now accept arbitrary SQL expressions as well as integer values.

Parameters `offset` – an integer OFFSET parameter, or a SQL expression that provides an integer result.

`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.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- `right` – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.

- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

FromClause.join()

Join

params(**optionaldict*, ***kwargs*)

Return a copy with bindparam() elements 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}
```

prefix_with(**expr*, ***kw*)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to prefix_with().

Parameters

- ***expr** – textual or ClauseElement construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

primary_key

Return the collection of Column objects which comprise the primary key of this FromClause.

reduce_columns(*sqlutil*, *only_synonyms=True*)

Return a new :func'.select' construct with redundantly named, equivalently-valued columns removed from the columns clause.

"Redundant" here means two columns where one refers to the other either based on foreign key, or via a simple equality comparison in the WHERE clause of the statement. The primary purpose of this method is to automatically construct a select statement with all uniquely-named columns, without the need to use table-qualified labels as apply_labels() does.

When columns are omitted based on foreign key, the referred-to column is the one that's kept. When columns are omitted based on WHERE eqivalence, the first column in the columns clause is the one that's kept.

Parameters **only_synonyms** – when True, limit the removal of columns to those which have the same name as the equivalent. Otherwise, all columns that are equivalent to another are removed.

New in version 0.8.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this FromClause.

scalar(**multiparams*, ***params*)
Compile and execute this Executable, returning the result's scalar representation.

select(*whereclause=None*, ***params*)
return a SELECT of this FromClause.

See also:

select() - general purpose method which allows for arbitrary column lists.

select_from(*fromclause*)
return a new select() construct with the given FROM expression merged into its list of FROM objects.

E.g.:

```
table1 = table('t1', column('a'))
table2 = table('t2', column('b'))
s = select([table1.c.a]).\
    select_from(
        table1.join(table2, table1.c.a==table2.c.b)
    )
```

The "from" list is a unique set on the identity of each element, so adding an already present Table or other selectable will have no effect. Passing a Join that refers to an already present Table or other selectable will have the effect of concealing the presence of that selectable as an individual element in the rendered FROM list, instead rendering it into a JOIN clause.

While the typical purpose of Select.select_from() is to replace the default, derived FROM clause with a join, it can also be called with individual table elements, multiple times if desired, in the case that the FROM clause cannot be fully derived from the columns clause:

```
select([func.count('*')]).select_from(table1)
```

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 and should not require explicit use.

suffix_with(**expr*, ***kw*)
Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

```
stmt = select([col1, col2]).cte().suffix_with(
    "cycle empno set y_cycle to 1 default 0", dialect="oracle")
```

Multiple suffixes can be specified by multiple calls to suffix_with().

Parameters

- ***expr** – textual or ClauseElement construct which will be rendered following the target clause.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this FromClause.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

`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.

`unique_params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

`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_for_update`(*nowait=False*, *read=False*, *of=None*, *skip_locked=False*, *key_share=False*)
Specify a `FOR UPDATE` clause for this `GenerativeSelect`.

E.g.:

```
stmt = select([table]).with_for_update(nowait=True)
```

On a database like PostgreSQL or Oracle, the above would render a statement like:

```
SELECT table.a, table.b FROM table FOR UPDATE NOWAIT
```

on other backends, the `nowait` option is ignored and instead would produce:

```
SELECT table.a, table.b FROM table FOR UPDATE
```

When called with no arguments, the statement will render with the suffix `FOR UPDATE`. Additional arguments can then be provided which allow for common database-specific variants.

Parameters

- **`nowait`** – boolean; will render `FOR UPDATE NOWAIT` on Oracle and PostgreSQL dialects.
- **`read`** – boolean; will render `LOCK IN SHARE MODE` on MySQL, `FOR SHARE` on PostgreSQL. On PostgreSQL, when combined with `nowait`, will render `FOR SHARE NOWAIT`.
- **`of`** – SQL expression or list of SQL expression elements (typically `Column` objects or a compatible expression) which will render into a `FOR UPDATE OF` clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend.
- **`skip_locked`** – boolean, will render `FOR UPDATE SKIP LOCKED` on Oracle and PostgreSQL dialects or `FOR SHARE SKIP LOCKED` if `read=True` is also specified.

  New in version 1.1.0.
- **`key_share`** – boolean, will render `FOR NO KEY UPDATE`, or if combined with `read=True` will render `FOR KEY SHARE`, on the PostgreSQL dialect.

  New in version 1.1.0.

with_hint(*selectable*, *text*, *dialect_name='*'*)

Add an indexing or other executional context 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')
```

See also:

`Select.with_statement_hint()`

with_only_columns(*columns*)

Return a new `select()` construct with its columns clause replaced with the given columns.

This method is exactly equivalent to as if the original `select()` had been called with the given columns clause. I.e. a statement:

```
s = select([table1.c.a, table1.c.b])
s = s.with_only_columns([table1.c.b])
```

should be exactly equivalent to:

```
s = select([table1.c.b])
```

This means that FROM clauses which are only derived from the column list will be discarded if the new column list no longer contains that FROM:

```
>>> table1 = table('t1', column('a'), column('b'))
>>> table2 = table('t2', column('a'), column('b'))
>>> s1 = select([table1.c.a, table2.c.b])
>>> print s1
SELECT t1.a, t2.b FROM t1, t2
>>> s2 = s1.with_only_columns([table2.c.b])
>>> print s2
SELECT t2.b FROM t1
```

The preferred way to maintain a specific FROM clause in the construct, assuming it won't be represented anywhere else (i.e. not in the WHERE clause, etc.) is to set it using `Select.select_from()`:

```
>>> s1 = select([table1.c.a, table2.c.b]).\
...         select_from(table1.join(table2,
...                 table1.c.a==table2.c.a))
>>> s2 = s1.with_only_columns([table2.c.b])
>>> print s2
SELECT t2.b FROM t1 JOIN t2 ON t1.a=t2.a
```

Care should also be taken to use the correct set of column objects passed to `Select.with_only_columns()`. Since the method is essentially equivalent to calling the `select()`

construct in the first place with the given columns, the columns passed to `Select.with_only_columns()` should usually be a subset of those which were passed to the `select()` construct, not those which are available from the `.c` collection of that `select()`. That is:

```
s = select([table1.c.a, table1.c.b]).select_from(table1)
s = s.with_only_columns([table1.c.b])
```

and **not**:

```
# usually incorrect
s = s.with_only_columns([s.c.b])
```

The latter would produce the SQL:

```
SELECT b
FROM (SELECT t1.a AS a, t1.b AS b
FROM t1), t1
```

Since the `select()` construct is essentially being asked to select both from `table1` as well as itself.

with_statement_hint(*text*, *dialect_name='*'*)
add a statement hint to this `Select`.

This method is similar to `Select.with_hint()` except that it does not require an individual table, and instead applies to the statement as a whole.

Hints here are specific to the backend database and may include directives such as isolation levels, file directives, fetch directives, etc.

New in version 1.0.0.

See also:

`Select.with_hint()`

class sqlalchemy.sql.expression.Selectable
mark a class as being selectable

class sqlalchemy.sql.expression.SelectBase
Base class for SELECT statements.

This includes `Select`, `CompoundSelect` and `TextAsFrom`.

alias(*name=None*, *flat=False*)
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

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 True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

bind
Returns the Engine or Connection to which this Executable is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

c
An alias for the columns attribute.

columns
A named-based collection of ColumnElement objects maintained by this FromClause.

The columns, or c collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`cte`(*name=None*, *recursive=False*)
Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **`name`** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.

- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
```

```
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

description
a brief description of this FromClause.

Used primarily for error message formatting.

execute(**multiparams*, ***params*)
Compile and execute this `Executable`.

execution_options(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

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).

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*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any FromClause object such as a Table object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies FromClause.join.isouter.

  New in version 1.1.

See also:

join() - standalone function

Join - the type of object produced

label(*name*)
return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

as_scalar().

lateral(*name=None*)
Return a LATERAL alias of this FromClause.

The return value is the Lateral construct also provided by the top-level lateral() function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

outerjoin(*right*, *onclause=None*, *full=False*)
Return a Join from this FromClause to another FromClause, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any FromClause object such as a Table object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.

- `full` – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

`params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

`primary_key`
Return the collection of Column objects which comprise the primary key of this FromClause.

`replace_selectable`(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

`scalar`(**multiparams*, ***params*)
Compile and execute this `Executable`, returning the result's scalar representation.

`select`(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

`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.

`tablesample`(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

unique_params(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**TableClause**(*name*, **columns*)
Represents a minimal "table" construct.

This is a lightweight table object that has only a name and a collection of columns, which are typically produced by the `expression.column()` function:

```
from sqlalchemy import table, column

user = table("user",
        column("id"),
        column("name"),
        column("description"),
)
```

The `TableClause` construct serves as the base for the more commonly used `Table` object, providing the usual set of `FromClause` services including the `.c.` collection and statement generation methods.

It does **not** provide all the additional schema-level services of `Table`, including constraints, references to other tables, or support for `MetaData`-level services. It's useful on its own as an ad-hoc construct used to generate quick SQL statements when a more fully fledged `Table` is not on hand.

alias(*name=None*, *flat=False*)
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

c
An alias for the `columns` attribute.

columns
A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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 ancestor 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 ancestor with one of the exported columns of this FromClause.

count(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this FromClause.

Deprecated since version 1.1: FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of func.count() should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

delete(*dml*, *whereclause=None*, ***kwargs*)
Generate a `delete()` construct against this `TableClause`.

E.g.:

```
table.delete().where(table.c.id==7)
```

See `delete()` for argument and usage information.

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

implicit_returning = False
`TableClause` doesn't support having a primary key or column -level defaults, so implicit returning doesn't apply.

insert(*dml*, *values=None*, *inline=False*, ***kwargs*)
Generate an `insert()` construct against this `TableClause`.

E.g.:

```
table.insert().values(name='foo')
```

See `insert()` for argument and usage information.

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*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

lateral(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

outerjoin(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

select(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

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.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

update(*dml*, *whereclause=None*, *values=None*, *inline=False*, ***kwargs*)
Generate an `update()` construct against this `TableClause`.

E.g.:

```
table.update().where(table.c.id==7).values(name='foo')
```

See `update()` for argument and usage information.

class sqlalchemy.sql.expression.**TableSample**(*selectable*, *sampling*, *name=None*, *seed=None*)
Represent a TABLESAMPLE clause.

This object is constructed from the `tablesample()` module level function as well as the `FromClause.tablesample()` method available on all `FromClause` subclasses.

New in version 1.1.

See also:

`tablesample()`

alias(*name=None*, *flat=False*)
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

c
An alias for the `columns` attribute.

columns
A named-based collection of ColumnElement objects maintained by this FromClause.

The columns, or c collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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 ancestor 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 ancestor with one of the exported columns of this FromClause.

count(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this FromClause.

Deprecated since version 1.1: FromClause.count() is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate func.count() expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of func.count() should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

func

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

join(*right*, *onclause=None*, *isouter=False*, *full=False*)
Return a Join from this FromClause to another FromClause.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any FromClause object such as a Table object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies FromClause.join.isouter.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

`params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

`primary_key`
Return the collection of Column objects which comprise the primary key of this FromClause.

`replace_selectable`(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

`select`(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

`tablesample`(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

`unique_params`(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

`class sqlalchemy.sql.expression.TextAsFrom`(*text*, *columns*, *positional=False*)
Wrap a `TextClause` construct within a `SelectBase` interface.

This allows the `TextClause` object to gain a `.c` collection and other FROM-like capabilities such as `FromClause.alias()`, `SelectBase.cte()`, etc.

The `TextAsFrom` construct is produced via the `TextClause.columns()` method - see that method for details.

New in version 0.9.0.

See also:

`text()`

`TextClause.columns()`

`alias`(*name=None*, *flat=False*)
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

`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 True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

bind
Returns the `Engine` or `Connection` to which this `Executable` is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

c
An alias for the `columns` attribute.

columns
A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)
```

```
print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`cte`(*name=None*, *recursive=False*)
Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
```

```
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

description
a brief description of this FromClause.

Used primarily for error message formatting.

execute(**multiparams*, ***params*)
Compile and execute this `Executable`.

execution_options(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

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).

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*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

`label`(*name*)
return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

`as_scalar()`.

`lateral`(*name=None*)
Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

`outerjoin`(*right*, *onclause=None*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.

- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

`FromClause.join()`

`Join`

params(**optionaldict*, ***kwargs*)
Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

scalar(**multiparams*, ***params*)
Compile and execute this `Executable`, returning the result's scalar representation.

select(*whereclause=None*, ***params*)
return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

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.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this `FromClause`.

The return value is the `TableSample` construct also provided by the top-level `tablesample()` function.

New in version 1.1.

See also:

`tablesample()` - usage guidelines and parameters

unique_params(**optionaldict*, ***kwargs*)

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

3.2.3 Insert, Updates, Deletes

INSERT, UPDATE and DELETE statements build on a hierarchy starting with `UpdateBase`. The `Insert` and `Update` constructs build on the intermediary `ValuesBase`.

sqlalchemy.sql.expression.delete(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, ***dialect_kw*)

Construct `Delete` object.

Similar functionality is available via the `delete()` method on `Table`.

Parameters

- **table** – The table to delete rows from.
- **whereclause** –

  A `ClauseElement` describing the `WHERE` condition of the `DELETE` statement. Note that the `where()` generative method may be used instead.

  The WHERE clause can refer to multiple tables. For databases which support this, a `DELETE..USING` or similar clause will be generated. The statement will fail on databases that don't have support for multi-table delete statements. A SQL-standard method of referring to additional tables in the WHERE clause is to use a correlated subquery:

```
users.delete().where(
        users.c.name==select([addresses.c.email_address]).
↪                     where(addresses.c.user_id==users.c.id).
↪                     as_scalar()
        )
```

  Changed in version 1.2.0: The WHERE clause of DELETE can refer to multiple tables.

See also:

deletes - SQL Expression Tutorial

sqlalchemy.sql.expression.insert(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)

Construct an `Insert` object.

Similar functionality is available via the `insert()` method on `Table`.

Parameters

- **table** – `TableClause` which is the subject of the insert.
- **values** – collection of values to be inserted; see `Insert.values()` for a description of allowed formats here. Can be omitted entirely; a `Insert` construct will also dynamically render the VALUES clause at execution time based on the parameters passed to `Connection.execute()`.
- **inline** – if True, no attempt will be made to retrieve the SQL-generated default values to be provided within the statement; in particular, this allows SQL expressions to be rendered 'inline' within the statement without the need to pre-execute them beforehand; for backends that support "returning", this turns off the "implicit returning" feature for the statement.

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.

See also:

Insert Expressions - SQL Expression Tutorial

inserts_and_updates - SQL Expression Tutorial

`sqlalchemy.sql.expression.update`(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, *preserve_parameter_order=False*, ***dialect_kw*)

Construct an `Update` object.

E.g.:

```
from sqlalchemy import update

stmt = update(users).where(users.c.id==5).\
        values(name='user #5')
```

Similar functionality is available via the `update()` method on `Table`:

```
stmt = users.update().\
            where(users.c.id==5).\
            values(name='user #5')
```

Parameters

- **`table`** – A `Table` object representing the database table to be updated.
- **`whereclause`** – Optional SQL expression describing the `WHERE` condition of the `UPDATE` statement. Modern applications may prefer to use the generative `where()` method to specify the `WHERE` clause.

  The WHERE clause can refer to multiple tables. For databases which support this, an `UPDATE FROM` clause will be generated, or on MySQL, a multi-table update. The statement will fail on databases that don't have support for multi-table update statements. A SQL-standard method of referring to additional tables in the WHERE clause is to use a correlated subquery:

  ```
  users.update().values(name='ed').where(
          users.c.name==select([addresses.c.email_address]).\
                      where(addresses.c.user_id==users.c.id).\
                      as_scalar()
          )
  ```

  Changed in version 0.7.4: The WHERE clause of UPDATE can refer to multiple tables.

- **`values`** – Optional dictionary which specifies the `SET` conditions of the `UPDATE`. If left as `None`, the `SET` conditions are determined from those parameters passed to the statement during the execution and/or compilation of the statement. When

compiled standalone without any parameters, the SET clause generates for all columns.

Modern applications may prefer to use the generative Update.values() method to set the values of the UPDATE statement.

- **inline** – if True, SQL defaults present on Column objects via the default keyword will be compiled 'inline' into the statement and not pre-executed. This means that their values will not be available in the dictionary returned from ResultProxy.last_updated_params().
- **preserve_parameter_order** – if True, the update statement is expected to receive parameters **only** via the Update.values() method, and they must be passed as a Python list of 2-tuples. The rendered UPDATE statement will emit the SET clause for each referenced column maintaining this order.

  New in version 1.0.10.

  See also:

  updates_order_parameters - full example of the preserve_parameter_order flag

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 (specifically the "key" of the Column, normally but not necessarily equivalent to its "name"). Normally, the Column objects used here are expected to be part of the target Table that is the table to be updated. However when using MySQL, a multiple-table UPDATE statement can refer to columns from any of the tables referred to in the WHERE clause.

The values referred to in values are typically:

- a literal data value (i.e. string, number, etc.)
- a SQL expression, such as a related Column, a scalar-returning select() construct, etc.

When combining select() constructs within the values clause of an update() construct, the subquery represented by the select() should be *correlated* to the parent table, that is, providing criterion which links the table inside the subquery to the outer table being updated:

```
users.update().values(
        name=select([addresses.c.email_address]).\
                where(addresses.c.user_id==users.c.id).\
                as_scalar()
    )
```

See also:

inserts_and_updates - SQL Expression Language Tutorial

class sqlalchemy.sql.expression.**Delete**(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, ***dialect_kw*)

Represent a DELETE construct.

The Delete object is created using the delete() function.

argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The DialectKWArgs.argument_for() method is a per-argument way adding extra arguments to the DefaultDialect.construct_arguments dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a NoSuchModuleError is raised. The dialect must also include an existing DefaultDialect.construct_arguments collection, indicating that it participates in the keyword-argument validation and default system, else ArgumentError is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind
Return a 'bind' linked to this UpdateBase or a Table associated with it.

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any

custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`cte`(*name=None*, *recursive=False*)

Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **`name`** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **`recursive`** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
```

```
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

dialect_kwargs

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

execute(**multiparams*, ***params*)
Compile and execute this `Executable`.

execution_options(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

kwargs
A synonym for `DialectKWArgs.dialect_kwargs`.

params(**arg*, ***kw*)
Set the parameters for the statement.

This method raises `NotImplementedError` on the base class, and is overridden by `ValuesBase` to provide the SET/VALUES clause of UPDATE and INSERT.

prefix_with(**expr*, ***kw*)
Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

returning(**cols*)
Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While `Column` objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using `ResultProxy.fetchone()` and similar. For DBAPIs which do not natively support returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

`ValuesBase.return_defaults()` - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

scalar(**multiparams*, ***params*)
Compile and execute this `Executable`, returning 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()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

where(*whereclause*)
Add the given WHERE clause to a newly returned delete construct.

with_hint(*text*, *selectable=None*, *dialect_name='*'*)
Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **text** – Text of the hint.
- **selectable** – optional `Table` that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.
- **dialect_name** – defaults to `*`, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class sqlalchemy.sql.expression.**Insert**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)
Represent an INSERT construct.

The `Insert` object is created using the `insert()` function.

See also:

Insert Expressions

argument_for(*dialect_name*, *argument_name*, *default*)
Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All

dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.

- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind
Return a 'bind' linked to this UpdateBase or a Table associated with it.

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`cte`(*name=None*, *recursive=False*)

Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **`name`** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **`recursive`** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
```

```
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
```

```
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

`dialect_kwargs`
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

`dialect_options`
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

`execute`(**multiparams*, ***params*)
Compile and execute this `Executable`.

`execution_options`(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The execution_options() method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See Connection.execution_options() for a full list of possible options.

See also:

Connection.execution_options()

Query.execution_options()

from_select(*names*, *select*, *include_defaults=True*)

Return a new Insert construct which represents an INSERT...FROM SELECT statement.

e.g.:

```
sel = select([table1.c.a, table1.c.b]).where(table1.c.c > 5)
ins = table2.insert().from_select(['a', 'b'], sel)
```

Parameters

- **names** – a sequence of string column names or Column objects representing the target columns.
- **select** – a select() construct, FromClause or other construct which resolves into a FromClause, such as an ORM Query object, etc. The order of columns returned from this FROM clause should correspond to the order of columns sent as the names parameter; while this is not checked before passing along to the database, the database would normally raise an exception if these column lists don't correspond.
- **include_defaults** – if True, non-server default values and SQL expressions as specified on Column objects (as documented in metadata_defaults_toplevel) not otherwise specified in the list of names will be rendered into the INSERT and SELECT statements, so that these values are also included in the data to be inserted.

  Note: A Python-side default that uses a Python callable function will only be invoked **once** for the whole statement, and **not per row**.

  New in version 1.0.0: - Insert.from_select() now renders Python-side and SQL expression column defaults into the SELECT statement for columns otherwise not included in the list of column names.

Changed in version 1.0.0: an INSERT that uses FROM SELECT implies that the insert.inline flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the ResultProxy.inserted_primary_key accessor does not apply.

New in version 0.8.3.

kwargs

A synonym for DialectKWArgs.dialect_kwargs.

params(**arg*, ***kw*)

Set the parameters for the statement.

This method raises `NotImplementedError` on the base class, and is overridden by `ValuesBase` to provide the SET/VALUES clause of UPDATE and INSERT.

`prefix_with`(**expr*, ***kw*)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- **`*expr`** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- **`**kw`** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

`return_defaults`(**cols*)

Make use of a RETURNING clause for the purpose of fetching server-side expressions and defaults.

E.g.:

```
stmt = table.insert().values(data='newdata').return_defaults()

result = connection.execute(stmt)

server_created_at = result.returned_defaults['created_at']
```

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that `UpdateBase.returning()` is not used simultaneously. The column values will then be available on the result using the `ResultProxy.returned_defaults` accessor as a dictionary, referring to values keyed to the `Column` object as well as its `.key`.

This method differs from `UpdateBase.returning()` in these ways:

1. `ValuesBase.return_defaults()` is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), `ValuesBase.return_defaults()` is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when `ValuesBase.return_defaults()` is used. By contrast, `UpdateBase.returning()` leaves the RETURNING result-set intact with a collection of any number of rows.
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the `UpdateBase.returning()` method circumvents this behavior, `ValuesBase.return_defaults()` leaves it intact.
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of `ResultProxy.returned_defaults` will be `None`

`ValuesBase.return_defaults()` is used by the ORM to provide an efficient implementation for the `eager_defaults` feature of `mapper()`.

Parameters `cols` – optional list of column key names or `Column` objects. If omitted, all column expressions evaluated on the server are added to the returning list.

New in version 0.9.0.

See also:

`UpdateBase.returning()`

`ResultProxy.returned_defaults`

`returning(*cols)`

Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While `Column` objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using `ResultProxy.fetchone()` and similar. For DBAPIs which do not natively support returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

`ValuesBase.return_defaults()` - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

`scalar(*multiparams, **params)`

Compile and execute this `Executable`, returning 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()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

`values`(**args*, ***kwargs*)

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the `Insert` and `Update` constructs support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to `Connection.execute()`. However, the `ValuesBase.values()` method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to `ValuesBase.values()` will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

Parameters

- **`**kwargs`** – key value pairs representing the string key of a `Column` mapped to the value to be rendered into the VALUES or SET clause:

```
users.insert().values(name="some name")

users.update().where(users.c.id==5).values(name="some name")
```

- **`*args`** – As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an `Insert` or an `Update` construct.

  For either an `Insert` or `Update` construct, a single dictionary can be passed, which works the same as that of the kwargs form:

```
users.insert().values({"name": "some name"})

users.update().values({"name": "some new name"})
```

  Also for either form but more typically for the `Insert` construct, a tuple that contains an entry for every column in the table is also accepted:

```
users.insert().values((5, "some name"))
```

  The `Insert` construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, PostgreSQL, MySQL, but not necessarily others:

```
users.insert().values([
                    {"name": "some name"},
                    {"name": "some other name"},
                    {"name": "yet another name"},
                ])
```

The above form would render a multiple VALUES statement similar to:

```
INSERT INTO users (name) VALUES
                (:name_1),
                (:name_2),
                (:name_3)
```

It is essential to note that **passing multiple values is NOT the same as using traditional executemany() form**. The above syntax is a **special** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the `Connection.execute()` method, which is supported by all database backends and is generally more efficient for a very large number of parameters.

See also:

execute_multiple - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.

Changed in version 1.0.0: an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the `Insert.inline` flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the `ResultProxy.inserted_primary_key` accessor does not apply.

Changed in version 1.0.0: A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See bug_3288 for other details.

The `Update` construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with the `preserve_parameter_order` parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to `Update.values()`, rather than the ordering of columns given in the `Table`.

New in version 1.0.10: - added support for parameter-ordered UPDATE statements via the `preserve_parameter_order` flag.

See also:

updates_order_parameters - full example of the `preserve_parameter_order` flag

See also:

inserts_and_updates - SQL Expression Language Tutorial

`insert()` - produce an `INSERT` statement

`update()` - produce an `UPDATE` statement

`with_hint`(*text*, *selectable=None*, *dialect_name='*'*)

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **text** – Text of the hint.
- **selectable** – optional Table that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.
- **dialect_name** – defaults to *, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class sqlalchemy.sql.expression.**Update**(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, *preserve_parameter_order=False*, ***dialect_kw*)

Represent an Update construct.

The Update object is created using the update() function.

argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The DialectKWArgs.argument_for() method is a per-argument way adding extra arguments to the DefaultDialect.construct_arguments dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a NoSuchModuleError is raised. The dialect must also include an existing DefaultDialect.construct_arguments collection, indicating that it participates in the keyword-argument validation and default system, else ArgumentError is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind

Return a 'bind' linked to this UpdateBase or a Table associated with it.

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`(*default*, *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 from 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*.
- **`compile_kwargs`** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`cte`(*name=None*, *recursive=False*)
Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)
```

```
included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)
```

```
connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

execute(**multiparams*, ***params*)
Compile and execute this `Executable`.

execution_options(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

kwargs
A synonym for `DialectKWArgs.dialect_kwargs`.

params(**arg*, ***kw*)
Set the parameters for the statement.

This method raises `NotImplementedError` on the base class, and is overridden by `ValuesBase` to provide the SET/VALUES clause of UPDATE and INSERT.

prefix_with(**expr*, ***kw*)
Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

return_defaults(**cols*)
Make use of a RETURNING clause for the purpose of fetching server-side expressions and defaults.

E.g.:

```
stmt = table.insert().values(data='newdata').return_defaults()

result = connection.execute(stmt)

server_created_at = result.returned_defaults['created_at']
```

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that `UpdateBase.returning()` is not used simultaneously. The column values will then be available on the result using the `ResultProxy.returned_defaults` accessor as a dictionary, referring to values keyed to the `Column` object as well as its `.key`.

This method differs from `UpdateBase.returning()` in these ways:

1. `ValuesBase.return_defaults()` is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), `ValuesBase.return_defaults()` is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when `ValuesBase.return_defaults()` is used. By contrast, `UpdateBase.returning()` leaves the RETURNING result-set intact with a collection of any number of rows.
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the `UpdateBase.returning()` method circumvents this behavior, `ValuesBase.return_defaults()` leaves it intact.
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of `ResultProxy.returned_defaults` will be `None`

ValuesBase.return_defaults() is used by the ORM to provide an efficient implementation for the eager_defaults feature of mapper().

Parameters cols – optional list of column key names or Column objects. If omitted, all column expressions evaluated on the server are added to the returning list.

New in version 0.9.0.

See also:

UpdateBase.returning()

ResultProxy.returned_defaults

returning(*cols)

Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While Column objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using ResultProxy.fetchone() and similar. For DBAPIs which do not natively support returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

ValuesBase.return_defaults() - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

scalar(*multiparams, **params)

Compile and execute this Executable, returning 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()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

values(**args*, ***kwargs*)

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the `Insert` and `Update` constructs support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to `Connection.execute()`. However, the `ValuesBase.values()` method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to `ValuesBase.values()` will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

Parameters

- **`**kwargs`** – key value pairs representing the string key of a `Column` mapped to the value to be rendered into the VALUES or SET clause:

```
users.insert().values(name="some name")

users.update().where(users.c.id==5).values(name="some name")
```

- **`*args`** – As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an `Insert` or an `Update` construct.

  For either an `Insert` or `Update` construct, a single dictionary can be passed, which works the same as that of the kwargs form:

```
users.insert().values({"name": "some name"})

users.update().values({"name": "some new name"})
```

  Also for either form but more typically for the `Insert` construct, a tuple that contains an entry for every column in the table is also accepted:

```
users.insert().values((5, "some name"))
```

  The `Insert` construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, PostgreSQL, MySQL, but not necessarily others:

```
users.insert().values([
                    {"name": "some name"},
                    {"name": "some other name"},
```

```
                    {"name": "yet another name"},
                ])
```

The above form would render a multiple VALUES statement similar to:

```
INSERT INTO users (name) VALUES
                (:name_1),
                (:name_2),
                (:name_3)
```

It is essential to note that **passing multiple values is NOT the same as using traditional executemany() form**. The above syntax is a **special** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the `Connection.execute()` method, which is supported by all database backends and is generally more efficient for a very large number of parameters.

See also:

execute_multiple - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.

Changed in version 1.0.0: an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the `Insert.inline` flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the `ResultProxy.inserted_primary_key` accessor does not apply.

Changed in version 1.0.0: A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See bug_3288 for other details.

The `Update` construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with the `preserve_parameter_order` parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to `Update.values()`, rather than the ordering of columns given in the `Table`.

New in version 1.0.10: - added support for parameter-ordered UPDATE statements via the `preserve_parameter_order` flag.

See also:

updates_order_parameters - full example of the `preserve_parameter_order` flag

See also:

inserts_and_updates - SQL Expression Language Tutorial

`insert()` - produce an `INSERT` statement

`update()` - produce an `UPDATE` statement

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.

with_hint(*text*, *selectable=None*, *dialect_name='*'*)
Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **text** – Text of the hint.
- **selectable** – optional `Table` that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.
- **dialect_name** – defaults to `*`, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class sqlalchemy.sql.expression.**UpdateBase**

Form the base for `INSERT`, `UPDATE`, and `DELETE` statements.

argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind

Return a 'bind' linked to this `UpdateBase` or a `Table` associated with it.

`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`(*default*, *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 from 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*.
- **`compile_kwargs`** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`cte`(*name=None*, *recursive=False*)
Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

CTEs can also be applied to DML constructs UPDATE, INSERT and DELETE on some databases, both as a source of CTE rows when combined with RETURNING, as well as a

consumer of CTE rows.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

Changed in version 1.1: Added support for UPDATE/INSERT/DELETE as CTE, CTEs added to UPDATE/INSERT/DELETE.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples include two from PostgreSQL's documentation at http://www.postgresql.org/docs/current/static/queries-with.html, as well as additional examples.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)
```

```
metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

Example 3, an upsert using UPDATE and INSERT with CTEs:

```
from datetime import date
from sqlalchemy import (MetaData, Table, Column, Integer,
                        Date, select, literal, and_, exists)

metadata = MetaData()

visitors = Table('visitors', metadata,
    Column('product_id', Integer, primary_key=True),
    Column('date', Date, primary_key=True),
    Column('count', Integer),
)

# add 5 visitors for the product_id == 1
product_id = 1
day = date.today()
count = 5

update_cte = (
    visitors.update()
    .where(and_(visitors.c.product_id == product_id,
                visitors.c.date == day))
    .values(count=visitors.c.count + count)
    .returning(literal(1))
    .cte('update_cte')
```

```
)

upsert = visitors.insert().from_select(
    [visitors.c.product_id, visitors.c.date, visitors.c.count],
    select([literal(product_id), literal(day), literal(count)])
        .where(~exists(update_cte.select()))
)

connection.execute(upsert)
```

See also:

`orm.query.Query.cte()` - ORM version of `HasCTE.cte()`.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

execute(**multiparams*, ***params*)
Compile and execute this `Executable`.

execution_options(***kw*)
Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include “autocommit” and “stream_results”, but not “isolation_level” or “compiled_cache”. See `Connection.execution_options()` for a full list of possible options.

See also:

Connection.execution_options()

Query.execution_options()

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).

kwargs
A synonym for DialectKWArgs.dialect_kwargs.

params(**arg*, ***kw*)
Set the parameters for the statement.

This method raises NotImplementedError on the base class, and is overridden by ValuesBase to provide the SET/VALUES clause of UPDATE and INSERT.

prefix_with(**expr*, ***kw*)
Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to prefix_with().

Parameters

- ***expr** – textual or ClauseElement construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

returning(**cols*)
Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While Column objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using `ResultProxy.fetchone()` and similar. For DBAPIs which do not natively support returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

`ValuesBase.return_defaults()` - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

`scalar`(**multiparams*, ***params*)
Compile and execute this `Executable`, returning 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()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

`with_hint`(*text*, *selectable=None*, *dialect_name='*'*)
Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **`text`** – Text of the hint.

- **selectable** – optional Table that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.
- **dialect_name** – defaults to *, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class sqlalchemy.sql.expression.ValuesBase(*table*, *values*, *prefixes*)

Supplies support for ValuesBase.values() to INSERT and UPDATE constructs.

return_defaults(**cols*)

Make use of a RETURNING clause for the purpose of fetching server-side expressions and defaults.

E.g.:

```
stmt = table.insert().values(data='newdata').return_defaults()

result = connection.execute(stmt)

server_created_at = result.returned_defaults['created_at']
```

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that UpdateBase.returning() is not used simultaneously. The column values will then be available on the result using the ResultProxy.returned_defaults accessor as a dictionary, referring to values keyed to the Column object as well as its .key.

This method differs from UpdateBase.returning() in these ways:

1. ValuesBase.return_defaults() is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), ValuesBase.return_defaults() is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when ValuesBase.return_defaults() is used. By contrast, UpdateBase.returning() leaves the RETURNING result-set intact with a collection of any number of rows.
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the UpdateBase.returning() method circumvents this behavior, ValuesBase.return_defaults() leaves it intact.
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of ResultProxy.returned_defaults will be None

ValuesBase.return_defaults() is used by the ORM to provide an efficient implementation for the eager_defaults feature of mapper().

Parameters **cols** – optional list of column key names or Column objects. If omitted, all column expressions evaluated on the server are added to the returning list.

New in version 0.9.0.

See also:

UpdateBase.returning()

ResultProxy.returned_defaults

values(**args*, ***kwargs*)

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the `Insert` and `Update` constructs support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to `Connection.execute()`. However, the `ValuesBase.values()` method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to `ValuesBase.values()` will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

Parameters

- **`**kwargs`** – key value pairs representing the string key of a `Column` mapped to the value to be rendered into the VALUES or SET clause:

```
users.insert().values(name="some name")

users.update().where(users.c.id==5).values(name="some name")
```

- **`*args`** – As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an `Insert` or an `Update` construct.

  For either an `Insert` or `Update` construct, a single dictionary can be passed, which works the same as that of the kwargs form:

```
users.insert().values({"name": "some name"})

users.update().values({"name": "some new name"})
```

  Also for either form but more typically for the `Insert` construct, a tuple that contains an entry for every column in the table is also accepted:

```
users.insert().values((5, "some name"))
```

  The `Insert` construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, PostgreSQL, MySQL, but not necessarily others:

```
users.insert().values([
                    {"name": "some name"},
                    {"name": "some other name"},
                    {"name": "yet another name"},
                ])
```

  The above form would render a multiple VALUES statement similar to:

```
INSERT INTO users (name) VALUES
                (:name_1),
                (:name_2),
                (:name_3)
```

  It is essential to note that **passing multiple values is NOT the same as using traditional executemany() form**. The above syntax is a **special** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the `Connection.execute()` method, which is supported by all database backends and is generally more efficient for a very large number of parameters.

  See also:

execute_multiple - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.

Changed in version 1.0.0: an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the `Insert.inline` flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the `ResultProxy.inserted_primary_key` accessor does not apply.

Changed in version 1.0.0: A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See bug_3288 for other details.

The `Update` construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with the `preserve_parameter_order` parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to `Update.values()`, rather than the ordering of columns given in the `Table`.

New in version 1.0.10: - added support for parameter-ordered UPDATE statements via the `preserve_parameter_order` flag.

See also:

updates_order_parameters - full example of the `preserve_parameter_order` flag

See also:

inserts_and_updates - SQL Expression Language Tutorial

`insert()` - produce an `INSERT` statement

`update()` - produce an `UPDATE` statement

3.2.4 SQL and 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.

SQL function API, factories, and built-in functions.

class sqlalchemy.sql.functions.**AnsiFunction**(***kwargs*)

identifier = 'AnsiFunction'

name = 'AnsiFunction'

class sqlalchemy.sql.functions.**Function**(*name*, **clauses*, ***kw*)

Describe a named SQL function.

See the superclass `FunctionElement` for a description of public methods.

See also:

`func` - namespace which produces registered or ad-hoc `Function` instances.

GenericFunction - allows creation of registered function types.

class sqlalchemy.sql.functions.FunctionElement(*clauses, **kwargs)
Base for SQL function-oriented constructs.

See also:

Function - named SQL function.

func - namespace which produces registered or ad-hoc Function instances.

GenericFunction - allows creation of registered function types.

alias(*name=None*, *flat=False*)
Produce a Alias construct against this FunctionElement.

This construct wraps the function in a named alias which is suitable for the FROM clause, in the style accepted for example by PostgreSQL.

e.g.:

```
from sqlalchemy.sql import column

stmt = select([column('data_view')]).\
    select_from(SomeTable).\
    select_from(func.unnest(SomeTable.data).alias('data_view')
)
```

Would produce:

```
SELECT data_view
FROM sometable, unnest(sometable.data) AS data_view
```

New in version 0.9.8: The FunctionElement.alias() method is now supported. Previously, this method's behavior was undefined and did not behave consistently across versions.

clauses
Return the underlying ClauseList which contains the arguments for this FunctionElement.

columns
The set of columns exported by this FunctionElement.

Function objects currently have no result column names built in; this method returns a single-element column collection with an anonymously named column.

An interim approach to providing named columns for a function as a FROM clause is to build a select() with the desired columns:

```
from sqlalchemy.sql import column

stmt = select([column('x'), column('y')]).                select_from(func.
↪myfunction())
```

execute()
Execute this FunctionElement against an embedded 'bind'.

This first calls select() to produce a SELECT construct.

Note that FunctionElement can be passed to the Connectable.execute() method of Connection or Engine.

filter(**criterion*)
Produce a FILTER clause against this function.

Used against aggregate and window functions, for database backends that support the "FILTER" clause.

The expression:

```
func.count(1).filter(True)
```

is shorthand for:

```
from sqlalchemy import funcfilter
funcfilter(func.count(1), True)
```

New in version 1.0.0.

See also:

`FunctionFilter`

`funcfilter()`

get_children(***kwargs*)

over(*partition_by=None*, *order_by=None*, *rows=None*, *range_=None*)
Produce an OVER clause against this function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

The expression:

```
func.row_number().over(order_by='x')
```

is shorthand for:

```
from sqlalchemy import over
over(func.row_number(), order_by='x')
```

See `over()` for a full description.

New in version 0.7.

packagenames = ()

scalar()
Execute this `FunctionElement` against an embedded 'bind' and return a scalar value.

This first calls `select()` to produce a SELECT construct.

Note that `FunctionElement` can be passed to the `Connectable.scalar()` method of `Connection` or `Engine`.

select()
Produce a `select()` construct against this `FunctionElement`.

This is shorthand for:

```
s = select([function_element])
```

self_group(*against=None*)

within_group(**order_by*)
Produce a WITHIN GROUP (ORDER BY expr) clause against this function.

Used against so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including `percentile_cont`, `rank`, `dense_rank`, etc.

See `within_group()` for a full description.

New in version 1.1.

within_group_type(*within_group*)
For types that define their return type as based on the criteria within a WITHIN GROUP (ORDER BY) expression, called by the `WithinGroup` construct.

Returns None by default, in which case the function's normal `.type` is used.

class sqlalchemy.sql.functions.**GenericFunction**(**args*, ***kwargs*)

Define a 'generic' function.

A generic function is a pre-established `Function` class that is instantiated automatically when called by name from the `func` attribute. Note that calling any name from `func` has the effect that a new `Function` instance is created automatically, given that name. The primary use case for defining a `GenericFunction` class is so that a function of a particular name may be given a fixed return type. It can also include custom argument parsing schemes as well as additional methods.

Subclasses of `GenericFunction` are automatically registered under the name of the class. For example, a user-defined function `as_utc()` would be available immediately:

```
from sqlalchemy.sql.functions import GenericFunction
from sqlalchemy.types import DateTime

class as_utc(GenericFunction):
    type = DateTime

print select([func.as_utc()])
```

User-defined generic functions can be organized into packages by specifying the "package" attribute when defining `GenericFunction`. Third party libraries containing many functions may want to use this in order to avoid name conflicts with other systems. For example, if our `as_utc()` function were part of a package "time":

```
class as_utc(GenericFunction):
    type = DateTime
    package = "time"
```

The above function would be available from `func` using the package name `time`:

```
print select([func.time.as_utc()])
```

A final option is to allow the function to be accessed from one name in `func` but to render as a different name. The `identifier` attribute will override the name used to access the function as loaded from `func`, but will retain the usage of `name` as the rendered name:

```
class GeoBuffer(GenericFunction):
    type = Geometry
    package = "geo"
    name = "ST_Buffer"
    identifier = "buffer"
```

The above function will render as follows:

```
>>> print func.geo.buffer()
ST_Buffer()
```

New in version 0.8: `GenericFunction` now supports automatic registration of new functions as well as package and custom naming support.

Changed in version 0.8: The attribute name `type` is used to specify the function's return type at the class level. Previously, the name `__return_type__` was used. This name is still recognized for backwards-compatibility.

coerce_arguments = True

identifier = 'GenericFunction'

name = 'GenericFunction'

class sqlalchemy.sql.functions.**OrderedSetAgg**(**args*, ***kwargs*)
Define a function where the return type is based on the sort expression type as defined by the expression passed to the `FunctionElement.within_group()` method.

array_for_multi_clause = False

identifier = 'OrderedSetAgg'

name = 'OrderedSetAgg'

within_group_type(*within_group*)

class sqlalchemy.sql.functions.**ReturnTypeFromArgs**(**args*, ***kwargs*)
Define a function whose return type is the same as its arguments.

identifier = 'ReturnTypeFromArgs'

name = 'ReturnTypeFromArgs'

class sqlalchemy.sql.functions.**array_agg**(**args*, ***kwargs*)
support for the ARRAY_AGG function.

The `func.array_agg(expr)` construct returns an expression of type `types.ARRAY`.

e.g.:

```
stmt = select([func.array_agg(table.c.values)[2:5]])
```

New in version 1.1.

See also:

postgresql.array_agg() - PostgreSQL-specific version that returns *postgresql.ARRAY*, which has PG-specific operators added.

identifier = 'array_agg'

name = 'array_agg'

type
alias of `ARRAY`

class sqlalchemy.sql.functions.**char_length**(*arg*, ***kwargs*)

identifier = 'char_length'

name = 'char_length'

type
alias of `Integer`

class sqlalchemy.sql.functions.**coalesce**(**args*, ***kwargs*)

identifier = 'coalesce'

name = 'coalesce'

class sqlalchemy.sql.functions.**concat**(**args*, ***kwargs*)

identifier = 'concat'

name = 'concat'

type
alias of `String`

class sqlalchemy.sql.functions.**count**(*expression=None*, ***kwargs*)
The ANSI COUNT aggregate function. With no arguments, emits COUNT *.

identifier = 'count'

name = 'count'

type
alias of Integer

class sqlalchemy.sql.functions.cube(*args, **kwargs)
Implement the CUBE grouping operation.

This function is used as part of the GROUP BY of a statement, e.g. Select.group_by():

```
stmt = select(
    [func.sum(table.c.value), table.c.col_1, table.c.col_2]
    ).group_by(func.cube(table.c.col_1, table.c.col_2))
```

New in version 1.2.

identifier = 'cube'

name = 'cube'

class sqlalchemy.sql.functions.cume_dist(*args, **kwargs)
Implement the cume_dist hypothetical-set aggregate function.

This function must be used with the FunctionElement.within_group() modifier to supply a sort expression to operate upon.

The return type of this function is Numeric.

New in version 1.1.

identifier = 'cume_dist'

name = 'cume_dist'

type = Numeric()

class sqlalchemy.sql.functions.current_date(**kwargs)

identifier = 'current_date'

name = 'current_date'

type
alias of Date

class sqlalchemy.sql.functions.current_time(**kwargs)

identifier = 'current_time'

name = 'current_time'

type
alias of Time

class sqlalchemy.sql.functions.current_timestamp(**kwargs)

identifier = 'current_timestamp'

name = 'current_timestamp'

type
alias of DateTime

class sqlalchemy.sql.functions.current_user(**kwargs)

identifier = 'current_user'

name = 'current_user'

type
alias of String

class sqlalchemy.sql.functions.**dense_rank**(**args*, ***kwargs*)
Implement the dense_rank hypothetical-set aggregate function.

This function must be used with the FunctionElement.within_group() modifier to supply a sort expression to operate upon.

The return type of this function is Integer.

New in version 1.1.

identifier = 'dense_rank'

name = 'dense_rank'

type = Integer()

class sqlalchemy.sql.functions.**grouping_sets**(**args*, ***kwargs*)
Implement the GROUPING SETS grouping operation.

This function is used as part of the GROUP BY of a statement, e.g. Select.group_by():

```
stmt = select(
    [func.sum(table.c.value), table.c.col_1, table.c.col_2]
).group_by(func.grouping_sets(table.c.col_1, table.c.col_2))
```

In order to group by multiple sets, use the tuple_() construct:

```
from sqlalchemy import tuple_

stmt = select(
    [
        func.sum(table.c.value),
        table.c.col_1, table.c.col_2,
        table.c.col_3]
).group_by(
    func.grouping_sets(
        tuple_(table.c.col_1, table.c.col_2),
        tuple_(table.c.value, table.c.col_3),
    )
)
```

New in version 1.2.

identifier = 'grouping_sets'

name = 'grouping_sets'

class sqlalchemy.sql.functions.**localtime**(***kwargs*)

identifier = 'localtime'

name = 'localtime'

type
alias of DateTime

class sqlalchemy.sql.functions.**localtimestamp**(***kwargs*)

identifier = 'localtimestamp'

name = 'localtimestamp'

type
alias of DateTime

class sqlalchemy.sql.functions.**max**(**args*, ***kwargs*)

identifier = 'max'

name = 'max'

class sqlalchemy.sql.functions.**min**(**args*, ***kwargs*)

identifier = 'min'

name = 'min'

class sqlalchemy.sql.functions.**mode**(**args*, ***kwargs*)
implement the mode ordered-set aggregate function.

This function must be used with the FunctionElement.within_group() modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression.

New in version 1.1.

identifier = 'mode'

name = 'mode'

class sqlalchemy.sql.functions.**next_value**(*seq*, ***kw*)
Represent the 'next value', given a Sequence as its single argument.

Compiles into the appropriate function on each backend, or will raise NotImplementedError if used on a backend that does not provide support for sequences.

identifier = 'next_value'

name = 'next_value'

type = Integer()

class sqlalchemy.sql.functions.**now**(**args*, ***kwargs*)

identifier = 'now'

name = 'now'

type
alias of DateTime

class sqlalchemy.sql.functions.**percent_rank**(**args*, ***kwargs*)
Implement the percent_rank hypothetical-set aggregate function.

This function must be used with the FunctionElement.within_group() modifier to supply a sort expression to operate upon.

The return type of this function is Numeric.

New in version 1.1.

identifier = 'percent_rank'

name = 'percent_rank'

type = Numeric()

class sqlalchemy.sql.functions.**percentile_cont**(**args*, ***kwargs*)
implement the percentile_cont ordered-set aggregate function.

This function must be used with the `FunctionElement.within_group()` modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression, or if the arguments are an array, an `types.ARRAY` of the sort expression's type.

New in version 1.1.

array_for_multi_clause = True

identifier = 'percentile_cont'

name = 'percentile_cont'

class sqlalchemy.sql.functions.**percentile_disc**(**args*, ***kwargs*)
implement the `percentile_disc` ordered-set aggregate function.

This function must be used with the `FunctionElement.within_group()` modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression, or if the arguments are an array, an `types.ARRAY` of the sort expression's type.

New in version 1.1.

array_for_multi_clause = True

identifier = 'percentile_disc'

name = 'percentile_disc'

class sqlalchemy.sql.functions.**random**(**args*, ***kwargs*)

identifier = 'random'

name = 'random'

class sqlalchemy.sql.functions.**rank**(**args*, ***kwargs*)
Implement the `rank` hypothetical-set aggregate function.

This function must be used with the `FunctionElement.within_group()` modifier to supply a sort expression to operate upon.

The return type of this function is `Integer`.

New in version 1.1.

identifier = 'rank'

name = 'rank'

type = Integer()

sqlalchemy.sql.functions.**register_function**(*identifier*, *fn*, *package='_default'*)
Associate a callable with a particular func. name.

This is normally called by _GenericMeta, but is also available by itself so that a non-Function construct can be associated with the `func` accessor (i.e. CAST, EXTRACT).

class sqlalchemy.sql.functions.**rollup**(**args*, ***kwargs*)
Implement the `ROLLUP` grouping operation.

This function is used as part of the GROUP BY of a statement, e.g. `Select.group_by()`:

```
stmt = select(
    [func.sum(table.c.value), table.c.col_1, table.c.col_2]
).group_by(func.rollup(table.c.col_1, table.c.col_2))
```

New in version 1.2.

identifier = 'rollup'

name = 'rollup'

class sqlalchemy.sql.functions.session_user(***kwargs*)

identifier = 'session_user'

name = 'session_user'

type
alias of String

class sqlalchemy.sql.functions.sum(**args*, ***kwargs*)

identifier = 'sum'

name = 'sum'

class sqlalchemy.sql.functions.sysdate(***kwargs*)

identifier = 'sysdate'

name = 'sysdate'

type
alias of DateTime

class sqlalchemy.sql.functions.user(***kwargs*)

identifier = 'user'

name = 'user'

type
alias of String

3.2.5 Custom SQL Constructs and Compilation Extension

Provides an API for creation of custom ClauseElements and compilers.

Synopsis

Usage involves the creation of one or more `ClauseElement` subclasses and one or more callables defining its compilation:

```
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.sql.expression import ColumnClause

class MyColumn(ColumnClause):
    pass

@compiles(MyColumn)
def compile_mycolumn(element, compiler, **kw):
    return "[%s]" % element.name
```

Above, `MyColumn` extends `ColumnClause`, the base expression element for 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]
```

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.

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 > :x_1)"
```

Note: The above `InsertFromSelect` construct is only an example, this actual functionality is already available using the `Insert.from_select()` method.

Note: The above `InsertFromSelect` construct probably wants to have "autocommit" enabled. See enabling_compiled_autocommit for this step.

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, literal_binds=True)
    )
```

Above, we add an additional flag to the process step as called by `SQLCompiler.process()`, which is the `literal_binds` flag. This indicates that any SQL expression which refers to a `BindParameter` object or other "literal" object such as those which refer to strings or integers should be rendered **in-place**, rather than being referred to as a bound parameter; when emitting DDL, bound parameters are typically not supported.

Enabling Autocommit on a Construct

Recall from the section autocommit that the `Engine`, when asked to execute a construct in the absence of a user-defined transaction, detects if the given construct represents DML or DDL, that is, a data modification or data definition statement, which requires (or may require, in the case of DDL) that the transaction generated by the DBAPI be committed (recall that DBAPI always has a transaction going on regardless of what SQLAlchemy does). Checking for this is actually accomplished by checking for the "autocommit" execution option on the construct. When building a construct like an INSERT derivation, a new DDL type, or perhaps a stored procedure that alters data, the "autocommit" option needs to be set in order for the statement to function with "connectionless" execution (as described in dbengine_implicit).

Currently a quick way to do this is to subclass `Executable`, then add the "autocommit" flag to the `_execution_options` dictionary (note this is a "frozen" dictionary which supplies a generative `union()` method):

```
from sqlalchemy.sql.expression import Executable, ClauseElement

class MyInsertThing(Executable, ClauseElement):
    _execution_options = \
        Executable._execution_options.union({'autocommit': True})
```

More succinctly, if the construct is truly similar to an INSERT, UPDATE, or DELETE, `UpdateBase` can be used, which already is a subclass of `Executable`, `ClauseElement` and includes the `autocommit` flag:

```
from sqlalchemy.sql.expression import UpdateBase

class MyInsertThing(UpdateBase):
```

```
    def __init__(self, ...):
        ...
```

DDL elements that subclass `DDLElement` already have the "autocommit" flag turned on.

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.

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'))
)
```

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`.

Further Examples

"UTC timestamp" function

A function that works like "CURRENT_TIMESTAMP" except applies the appropriate conversions so that the time is in UTC time. Timestamps are best stored in relational databases as UTC, without time zones. UTC so that your database doesn't think time has gone backwards in the hour when daylight savings ends, without timezones because timezones are like character encodings - they're best applied only at the endpoints of an application (i.e. convert to UTC upon user input, re-apply desired timezone upon display).

For PostgreSQL and Microsoft SQL Server:

```
from sqlalchemy.sql import expression
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.types import DateTime

class utcnow(expression.FunctionElement):
    type = DateTime()

@compiles(utcnow, 'postgresql')
def pg_utcnow(element, compiler, **kw):
    return "TIMEZONE('utc', CURRENT_TIMESTAMP)"

@compiles(utcnow, 'mssql')
def ms_utcnow(element, compiler, **kw):
    return "GETUTCDATE()"
```

Example usage:

```
from sqlalchemy import (
            Table, Column, Integer, String, DateTime, MetaData
```

```
        )
metadata = MetaData()
event = Table("event", metadata,
    Column("id", Integer, primary_key=True),
    Column("description", String(50), nullable=False),
    Column("timestamp", DateTime, server_default=utcnow())
)
```

"GREATEST" function

The "GREATEST" function is given any number of arguments and returns the one that is of the highest value - its equivalent to Python's `max` function. A SQL standard version versus a CASE based version which only accommodates two arguments:

```
from sqlalchemy.sql import expression
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.types import Numeric

class greatest(expression.FunctionElement):
    type = Numeric()
    name = 'greatest'

@compiles(greatest)
def default_greatest(element, compiler, **kw):
    return compiler.visit_function(element)

@compiles(greatest, 'sqlite')
@compiles(greatest, 'mssql')
@compiles(greatest, 'oracle')
def case_greatest(element, compiler, **kw):
    arg1, arg2 = list(element.clauses)
    return "CASE WHEN %s > %s THEN %s ELSE %s END" % (
        compiler.process(arg1),
        compiler.process(arg2),
        compiler.process(arg1),
        compiler.process(arg2),
    )
```

Example usage:

```
Session.query(Account).\
        filter(
            greatest(
                Account.checking_balance,
                Account.savings_balance) > 10000
        )
```

"false" expression

Render a "false" constant expression, rendering as "0" on platforms that don't have a "false" constant:

```
from sqlalchemy.sql import expression
from sqlalchemy.ext.compiler import compiles

class sql_false(expression.ColumnElement):
    pass

@compiles(sql_false)
def default_false(element, compiler, **kw):
```

```
    return "false"

@compiles(sql_false, 'mssql')
@compiles(sql_false, 'mysql')
@compiles(sql_false, 'oracle')
def int_false(element, compiler, **kw):
    return "0"
```

Example usage:

```
from sqlalchemy import select, union_all

exp = union_all(
    select([users.c.name, sql_false().label("enrolled")]),
    select([customers.c.name, customers.c.enrolled])
)
```

`sqlalchemy.ext.compiler.compiles`(*class_*, **specs*)
Register a function as a compiler for a given `ClauseElement` type.

`sqlalchemy.ext.compiler.deregister`(*class_*)
Remove all custom compilers associated with a given `ClauseElement` type.

3.2.6 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=4*)

sqlalchemy.ext.serializer.**loads**(*data*, *metadata=None*, *scoped_session=None*, *engine=None*)

3.3 Schema Definition Language

This section references SQLAlchemy **schema metadata**, a comprehensive system of describing and inspecting database schemas.

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.

3.3.1 Describing Databases with MetaData

This section discusses the fundamental `Table`, `Column` and `MetaData` objects.

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_toplevel.

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 metadata_foreignkeys. 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))
)

{sql}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"))
)
{sql}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:

```
{sql}employees.drop(engine)
DROP TABLE employees
{}
```

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)
```

Altering Schemas through Migrations

While SQLAlchemy directly supports emitting CREATE and DROP statements for schema constructs, the ability to alter those constructs, usually via the ALTER statement as well as other database-specific constructs, is outside of the scope of SQLAlchemy itself. While it's easy enough to emit ALTER statements and similar by hand, such as by passing a string to `Connection.execute()` or by using the `DDL` construct, it's a common practice to automate the maintenance of database schemas in relation to application code using schema migration tools.

There are two major migration tools available for SQLAlchemy:

- Alembic - Written by the author of SQLAlchemy, Alembic features a highly customizable environment and a minimalistic usage pattern, supporting such features as transactional DDL, automatic generation of "candidate" migrations, an "offline" mode which generates SQL scripts, and support for branch resolution.
- SQLAlchemy-Migrate - The original migration tool for SQLAlchemy, SQLAlchemy-Migrate is widely used and continues under active development. SQLAlchemy-Migrate includes features such as SQL script generation, ORM class generation, ORM model comparison, and extensive support for SQLite migrations.

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.

Column, Table, MetaData API

`sqlalchemy.schema.BLANK_SCHEMA`
: Symbol indicating that a `Table` or `Sequence` should have 'None' for its schema, even if the parent `MetaData` has specified a schema.

  See also:

  `MetaData.schema`

  `Table.schema`

  `Sequence.schema`

  New in version 1.0.14.

`class sqlalchemy.schema.Column(*args, **kwargs)`
: Represents a column in a database table.

  `all_()`
  : Produce a `all_()` clause against the parent object.

    This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == mytable.c.somearray.all_()

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().all_()
```

See also:

`all_()` - standalone version

`any_()` - ANY operator

New in version 1.1.

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.

any_()
Produce a `any_()` clause against the parent object.

This operator is only appropriate against a scalar subquery object, or for some backends an column expression that is against the ARRAY type, e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == mytable.c.somearray.any_()

# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == select([table.c.value]).as_scalar().any_()
```

See also:

`any_()` - standalone version

`all_()` - ALL operator

New in version 1.1.

asc()
Produce a `asc()` clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)
Produce a `between()` clause against the parent object, given the lower and upper range.

bool_op(*opstring*, *precedence=0*)
Return a custom boolean operator.

This method is shorthand for calling `Operators.op()` and passing the `Operators.op.is_comparison` flag with True.

New in version 1.2.0b3.

See also:

`Operators.op()`

cast(*type_*)
Produce a type cast, i.e. `CAST(<expression> AS <type>)`.

This is a shortcut to the `cast()` function.

New in version 1.0.7.

`collate`(*collation*)
Produce a `collate()` clause against the parent object, given the collation string.

See also:

`collate()`

`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 corresponding 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.

`compile`(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

`concat`(*other*)
Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

`contains`(*other*, ***kwargs*)
Implement the 'contains' operator.

Produces a LIKE expression that tests against a match for the middle of a string value:

```
column LIKE '%' || <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.contains("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.contains.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.contains.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **`other`** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.contains.autoescape` flag is set to True.
- **`autoescape`** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.contains("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE '%' || :param || '%' ESCAPE '/'
  ```

  With the value of :param as `"foo/%bar"`.

  New in version 1.2.

  Changed in version 1.2.0: The `ColumnOperators.contains.autoescape` parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the `ColumnOperators.contains.escape` parameter.
- **`escape`** – a character which when given will render with the `ESCAPE` keyword to establish that character as the escape character. This character can then be placed preceding occurrences of `%` and `_` to allow them to act as themselves and not wildcard characters.

  An expression such as:

```
somecolumn.contains("foo/%bar", escape="^")
```

Will render as:

```
somecolumn LIKE '%' || :param || '%' ESCAPE '^'
```

The parameter may also be combined with `ColumnOperators.contains.autoescape`:

```
somecolumn.contains("foo%bar^bat", escape="^", autoescape=True)
```

Where above, the given literal parameter will be converted to `"foo^%bar^^bat"` before being passed to the database.

See also:

`ColumnOperators.startswith()`

`ColumnOperators.endswith()`

`ColumnOperators.like()`

copy(***kw*)
Create a copy of this `Column`, unitialized.

This is used in `Table.tometadata`.

desc()
Produce a `desc()` clause against the parent object.

distinct()
Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)
Implement the 'endswith' operator.

Produces a LIKE expression that tests against a match for the end of a string value:

```
column LIKE '%' || <other>
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.endswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.endswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.endswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters % and _ are not escaped by default unless the `ColumnOperators.endswith.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

```
somecolumn.endswith("foo%bar", autoescape=True)
```

Will render as:

```
somecolumn LIKE '%' || :param ESCAPE '/'
```

With the value of :param as "foo/%bar".

New in version 1.2.

Changed in version 1.2.0: The ColumnOperators.endswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.endswith.escape parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

```
somecolumn.endswith("foo/%bar", escape="^")
```

  Will render as:

```
somecolumn LIKE '%' || :param ESCAPE '^'
```

  The parameter may also be combined with ColumnOperators.endswith.autoescape:

```
somecolumn.endswith("foo%bar^bat", escape="^", autoescape=True)
```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.startswith()

ColumnOperators.contains()

ColumnOperators.like()

expression

Return a column expression.

Part of the inspection interface; returns self.

ilike(*other*, *escape=None*)

Implement the ilike operator, e.g. case insensitive LIKE.

In a column context, produces an expression either of the form:

```
lower(a) LIKE lower(other)
```

Or on backends that support the ILIKE operator:

```
a ILIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)
Implement the in operator.

In a column context, produces the clause a IN other. "other" may be a tuple/list of column expressions, or a select() construct.

In the case that other is an empty sequence, the compiler produces an "empty in" expression. This defaults to the expression "1 != 1" to produce false in all cases. The create_engine.empty_in_strategy may be used to alter this behavior.

Changed in version 1.2: The ColumnOperators.in_() and ColumnOperators.notin_() operators now produce a "static" expression for an empty IN sequence by default.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this SchemaItem.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as Table and Column.

is_(*other*)
Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

is_distinct_from(*other*)
Implement the IS DISTINCT FROM operator.

Renders "a IS DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS NOT b".

New in version 1.1.

isnot(*other*)
Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

isnot_distinct_from(*other*)
Implement the IS NOT DISTINCT FROM operator.

Renders "a IS NOT DISTINCT FROM b" on most platforms; on some such as SQLite may render "a IS b".

New in version 1.1.

`label`(*name*)
Produce a column label, i.e. `<columnname> AS <name>`.

This is a shortcut to the `label()` function.

if 'name' is None, an anonymous label name will be generated.

`like`(*other*, *escape=None*)
Implement the `like` operator.

In a column context, produces the expression:

```
a LIKE other
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

  ```
  somecolumn.like("foo/%bar", escape="/")
  ```

See also:

`ColumnOperators.ilike()`

`match`(*other*, ***kwargs*)
Implements a database-specific 'match' operator.

`match()` attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- PostgreSQL - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

`notilike`(*other*, *escape=None*)
implement the `NOT ILIKE` operator.

This is equivalent to using negation with `ColumnOperators.ilike()`, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

`ColumnOperators.ilike()`

`notin_`(*other*)
implement the `NOT IN` operator.

This is equivalent to using negation with `ColumnOperators.in_()`, i.e. `~x.in_(y)`.

In the case that `other` is an empty sequence, the compiler produces an "empty not in" expression. This defaults to the expression "1 = 1" to produce true in all cases. The `create_engine.empty_in_strategy` may be used to alter this behavior.

Changed in version 1.2: The `ColumnOperators.in_()` and `ColumnOperators.notin_()` operators now produce a "static" expression for an empty IN sequence by default.

See also:

`ColumnOperators.in_()`

notlike(*other*, *escape=None*)
implement the `NOT LIKE` operator.

This is equivalent to using negation with `ColumnOperators.like()`, i.e. `~x.like(y)`.

New in version 0.8.

See also:

`ColumnOperators.like()`

nullsfirst()
Produce a `nullsfirst()` clause against the parent object.

nullslast()
Produce a `nullslast()` clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*, *return_type=None*)
produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

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`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like `==`, `>`, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.
- **return_type** – a `TypeEngine` class or object that will force the return type of an expression produced by this operator to be of that type. By default,

operators that specify `Operators.op.is_comparison` will resolve to `Boolean`, and those that do not will be of the same type as the left-hand operand.

New in version 1.2.0b3: - added the `Operators.op.return_type` argument.

See also:

types_operators

relationship_custom_operator

quote
Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

references(*column*)
Return True if this Column references the given column via foreign key.

shares_lineage(*othercolumn*)
Return True if the given `ColumnElement` has a common ancestor to this `ColumnElement`.

startswith(*other*, ***kwargs*)
Implement the `startswith` operator.

Produces a LIKE expression that tests against a match for the start of a string value:

```
column LIKE <other> || '%'
```

E.g.:

```
stmt = select([sometable]).\
    where(sometable.c.column.startswith("foobar"))
```

Since the operator uses `LIKE`, wildcard characters `"%"` and `"_"` that are present inside the <other> expression will behave like wildcards as well. For literal string values, the `ColumnOperators.startswith.autoescape` flag may be set to `True` to apply escaping to occurences of these characters within the string value so that they match as themselves and not as wildcard characters. Alternatively, the `ColumnOperators.startswith.escape` parameter will establish a given character as an escape character which can be of use when the target expression is not a literal string.

Parameters

- **other** – expression to be compared. This is usually a plain string value, but can also be an arbitrary SQL expression. LIKE wildcard characters `%` and `_` are not escaped by default unless the `ColumnOperators.startswith.autoescape` flag is set to True.
- **autoescape** – boolean; when True, establishes an escape character within the LIKE expression, then applies it to all occurrences of `"%"`, `"_"` and the escape character itself within the comparison value, which is assumed to be a literal string and not a SQL expression.

  An expression such as:

  ```
  somecolumn.startswith("foo%bar", autoescape=True)
  ```

  Will render as:

  ```
  somecolumn LIKE :param || '%' ESCAPE '/'
  ```

  With the value of :param as `"foo/%bar"`.

  New in version 1.2.

Changed in version 1.2.0: The ColumnOperators.startswith.autoescape parameter is now a simple boolean rather than a character; the escape character itself is also escaped, and defaults to a forwards slash, which itself can be customized using the ColumnOperators.startswith.escape parameter.

- **escape** – a character which when given will render with the ESCAPE keyword to establish that character as the escape character. This character can then be placed preceding occurrences of % and _ to allow them to act as themselves and not wildcard characters.

  An expression such as:

```
somecolumn.startswith("foo/%bar", escape="^")
```

  Will render as:

```
somecolumn LIKE :param || '%' ESCAPE '^'
```

  The parameter may also be combined with ColumnOperators.startswith.autoescape:

```
somecolumn.startswith("foo%bar^bat", escape="^", autoescape=True)
```

  Where above, the given literal parameter will be converted to "foo^%bar^^bat" before being passed to the database.

See also:

ColumnOperators.endswith()

ColumnOperators.contains()

ColumnOperators.like()

class sqlalchemy.schema.MetaData(*bind=None*, *reflect=False*, *schema=None*, *quote_schema=None*, *naming_convention={'ix': 'ix_%(column_0_label)s'}*, *info=None*)

A collection of Table objects and their associated schema constructs.

Holds a collection of Table objects as well as 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.

MetaData is a thread-safe object for read operations. Construction of new tables within a single MetaData object, either explicitly or via reflection, may not be completely thread-safe.

See also:

metadata_describing - Introduction to database metadata

append_ddl_listener(*event_name*, *listener*)

Append a DDL event listener to this MetaData.

Deprecated since version 0.7: See DDLEvents.

bind

An Engine or Connection to which this MetaData is bound.

Typically, a Engine is assigned to this attribute so that "implicit execution" may be used, or alternatively as a means of providing engine binding information to an ORM Session object:

```
engine = create_engine("someurl://")
metadata.bind = engine
```

See also:

dbengine_implicit - background on "bound metadata"

`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*, *extend_existing=False*, *autoload_replace=True*, ***dialect_kwargs*`)`
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. If None, the schema associated with this `MetaData` is used, if any.
- **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.

- **`extend_existing`** – Passed along to each `Table` as `Table.extend_existing`.

  New in version 0.9.1.

- **`autoload_replace`** – Passed along to each `Table` as `Table.autoload_replace`.

  New in version 0.9.1.

- **`**dialect_kwargs`** – Additional keyword arguments not mentioned above are dialect specific, and passed in the form `<dialectname>_<argname>`. See the documentation regarding an individual dialect at dialect_toplevel for detail on documented arguments.

  New in version 0.9.2: - Added `MetaData.reflect.**dialect_kwargs` to support dialect-level reflection options for all `Table` objects reflected.

`remove`(*table*)
Remove the given Table object from this MetaData.

`sorted_tables`
Returns a list of `Table` objects sorted in order of foreign key dependency.

The sorting will place `Table` objects that have dependencies first, before the dependencies themselves, representing the order in which they can be created. To get the order in which the tables would be dropped, use the `reversed()` Python built-in.

> **Warning:** The `sorted_tables` accessor cannot by itself accommodate automatic resolution of dependency cycles between tables, which are usually caused by mutually dependent foreign key constraints. To resolve these cycles, either the `ForeignKeyConstraint.use_alter` parameter may be appled to those constraints, or use the `schema.sort_tables_and_constraints()` function which will break out foreign key constraints involved in cycles separately.

See also:

`schema.sort_tables()`

`schema.sort_tables_and_constraints()`

`MetaData.tables`

`Inspector.get_table_names()`

`Inspector.get_sorted_table_and_fkc_names()`

`tables = None`
A dictionary of `Table` objects keyed to their name or "table key".

The exact key is that determined by the `Table.key` attribute; for a table with no `Table.schema` attribute, this is the same as `Table.name`. For a table with a schema, it is typically of the form `schemaname.tablename`.

See also:

`MetaData.sorted_tables`

`class sqlalchemy.schema.SchemaItem`
Base class for items that define a database schema.

`get_children`(***kwargs*)
used to allow SchemaVisitor access

`info`
Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

`quote`
Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

`class sqlalchemy.schema.Table(*args, **kw)`
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 and optional schema name within the given `MetaData` object. Calling the `Table` constructor with the same name and same `MetaData` argument a second time will return the *same* `Table` object - in this way the `Table` constructor acts as a registry function.

See also:

metadata_describing - Introduction to database metadata

Constructor arguments are as follows:

Parameters

- **`name`** – The name of this table as represented in the database.

  The table name, along with the value of the `schema` parameter, forms a key which uniquely identifies this `Table` within the owning `MetaData` collection. 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 or contain special characters. A name with any number of upper case characters is considered to be case sensitive, and will be sent as quoted.

  To enable unconditional quoting for the table name, specify the flag `quote=True` to the constructor, or use the `quoted_name` construct to specify the name.

- **`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, unless `Table.autoload_with` is set in which case it defaults to True; `Column` objects for this table should be reflected from the database, possibly augmenting or replacing existing `Column` objects that were explicitly specified.

Changed in version 1.0.0: setting the `Table.autoload_with` parameter implies that `Table.autoload` will default to True.

See also:

metadata_reflection_toplevel

- **`autoload_replace`** – Defaults to `True`; when using `Table.autoload` in conjunction with `Table.extend_existing`, indicates that `Column` objects present in the already-existing `Table` object should be replaced with columns of the same name retrieved from the autoload process. When `False`, columns already present under existing names will be omitted from the reflection process.

  Note that this setting does not impact `Column` objects specified programmatically within the call to `Table` that also is autoloading; those `Column` objects will always replace existing columns of the same name when `Table.extend_existing` is `True`.

  New in version 0.7.5.

  See also:

  `Table.autoload`

  `Table.extend_existing`

- **`autoload_with`** – An `Engine` or `Connection` object with which this `Table` object will be reflected; when set to a non-None value, it implies that `Table.autoload` is `True`. If left unset, but `Table.autoload` is explicitly set to `True`, an autoload operation will attempt to proceed by locating an `Engine` or `Connection` bound to the underlying `MetaData` object.

  See also:

  `Table.autoload`

- **`extend_existing`** – 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 `Table.extend_existing` or `Table.keep_existing` are not set, and the given name of the new `Table` refers to a `Table` that is already present in the target `MetaData` collection, and this `Table` specifies additional columns or other constructs or flags that modify the table's state, an error is raised. The purpose of these two mutually-exclusive flags is to specify what action should be taken when a `Table` is specified that matches an existing `Table`, yet specifies additional constructs.

  `Table.extend_existing` will also work in conjunction with `Table.autoload` to run a new reflection operation against the database, even if a `Table` of the same name is already present in the target `MetaData`; newly reflected `Column` objects and other options will be added into the state of the `Table`, potentially overwriting existing columns and options of the same name.

  Changed in version 0.7.4: `Table.extend_existing` will invoke a new reflection operation when combined with `Table.autoload` set to True.

  As is always the case with `Table.autoload`, `Column` objects can be specified in the same `Table` constructor, which will take precedence. Below, the existing table `mytable` will be augmented with `Column` objects both reflected from the database, as well as the given `Column` named "y":

```
Table("mytable", metadata,
            Column('y', Integer),
            extend_existing=True,
            autoload=True,
```

```
            autoload_with=engine
        )
```

See also:

`Table.autoload`

`Table.autoload_replace`

`Table.keep_existing`

- **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** – Optional data dictionary which will be populated into the `SchemaItem.info` attribute of this object.
- **keep_existing** – When `True`, indicates that if this Table is already present in the given `MetaData`, ignore further arguments within the constructor to the existing `Table`, and return the `Table` object as originally created. This is to allow a function that wishes to define a new `Table` on first call, but on subsequent calls will return the same `Table`, without any of the declarations (particularly constraints) being applied a second time.

  If `Table.extend_existing` or `Table.keep_existing` are not set, and the given name of the new `Table` refers to a `Table` that is already present in the target `MetaData` collection, and this `Table` specifies additional columns or other constructs or flags that modify the table's state, an error is raised. The purpose of these two mutually-exclusive flags is to specify what action should be taken when a `Table` is specified that matches an existing `Table`, yet specifies additional constructs.

  See also:

  `Table.extend_existing`

- **listeners** – A list of tuples of the form `(<eventname>, <fn>)` which will be passed to `event.listen()` upon construction. This alternate hook to `event.listen()` allows the establishment of a listener function specific to this `Table` before the "autoload" process begins. Particularly useful for the `DDLEvents.column_reflect()` event:

```
def listen_for_reflect(table, column_info):
    "handle the column reflection event"
    # ...

t = Table(
    'sometable',
    autoload=True,
    listeners=[
        ('column_reflect', listen_for_reflect)
    ])
```

- **mustexist** – When `True`, indicates that this Table must already be present in the given `MetaData` collection, else an exception is raised.
- **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`.

  If the owning `MetaData` of this `Table` specifies its own `MetaData.schema` parameter, then that schema name will be applied to this `Table` if the schema parameter here is set to `None`. To set a blank schema name on a `Table` that would otherwise use the schema set on the owning `MetaData`, specify the special symbol `BLANK_SCHEMA`.

  New in version 1.0.14: Added the `BLANK_SCHEMA` symbol to allow a `Table` to have a blank schema name even when the parent `MetaData` specifies `MetaData.schema`.

  The quoting rules for the schema name are the same as those for the `name` parameter, in that quoting is applied for reserved words or case-sensitive names; to enable unconditional quoting for the schema name, specify the flag `quote_schema=True` to the constructor, or use the `quoted_name` construct to specify the name.
- **useexisting** – Deprecated. Use `Table.extend_existing`.
- **comment** – Optional string that will render an SQL comment on table creation.

  New in version 1.2: Added the `Table.comment` parameter to `Table`.
- ****kw** – Additional keyword arguments not mentioned above are dialect specific, and passed in the form `<dialectname>_<argname>`. See the documentation regarding an individual dialect at dialect_toplevel for detail on documented arguments.

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.

alias(*name=None*, *flat=False*)
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

append_column(*column*)
Append a `Column` to this `Table`.

The "key" of the newly added `Column`, i.e. the value of its `.key` attribute, will then be available in the `.c` collection of this `Table`, and the column definition will be included in any CREATE TABLE, SELECT, UPDATE, etc. statements generated from this `Table` construct.

Note that this does **not** change the definition of the table as it exists within any underlying database, assuming that table has already been created in the database. Relational databases support the addition of columns to existing tables using the SQL ALTER command, which would need to be emitted for an already-existing table that doesn't contain the newly added column.

append_constraint(*constraint*)
Append a `Constraint` to this `Table`.

This has the effect of the constraint being included in any future CREATE TABLE statement, assuming specific DDL creation events have not been associated with the given `Constraint` object.

Note that this does **not** produce the constraint within the relational database automatically, for a table that already exists in the database. To add a constraint to an existing relational database table, the SQL ALTER command must be used. SQLAlchemy also provides the `AddConstraint` construct which can produce this SQL when invoked as an executable clause.

append_ddl_listener(*event_name*, *listener*)
Append a DDL event listener to this `Table`.

Deprecated since version 0.7: See `DDLEvents`.

argument_for(*dialect_name*, *argument_name*, *default*)
Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind
Return the connectable associated with this Table.

c
An alias for the `columns` attribute.

columns
A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The columns, or c collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

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(*default*, *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 from 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*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

faq_sql_expression_string

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 ancestor 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 ancestor with one of the exported columns of this `FromClause`.

`count`(*functions*, *whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

Deprecated since version 1.1: `FromClause.count()` is deprecated. Counting rows requires that the correct column expression and accommodations for joins, DISTINCT, etc. must be made, otherwise results may not be what's expected. Please use an appropriate `func.count()` expression directly.

The function generates COUNT against the first column in the primary key of the table, or against the first column in the table overall. Explicit use of `func.count()` should be preferred:

```
row_count = conn.scalar(
    select([func.count('*')]).select_from(table)
)
```

See also:

`func`

`create`(*bind=None*, *checkfirst=False*)
Issue a `CREATE` statement for this `Table`, using the given `Connectable` for connectivity.

See also:

`MetaData.create_all()`.

`delete`(*dml*, *whereclause=None*, ***kwargs*)
Generate a `delete()` construct against this `TableClause`.

E.g.:

```
table.delete().where(table.c.id==7)
```

See `delete()` for argument and usage information.

`dialect_kwargs`
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

`dialect_options`
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

drop(*bind=None*, *checkfirst=False*)
Issue a `DROP` statement for this `Table`, using the given `Connectable` for connectivity.

See also:

`MetaData.drop_all()`.

exists(*bind=None*)
Return True if this table exists.

foreign_key_constraints
`ForeignKeyConstraint` objects referred to by this `Table`.

This list is produced from the collection of `ForeignKey` objects currently associated.

New in version 1.0.0.

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

insert(*dml*, *values=None*, *inline=False*, ***kwargs*)
Generate an `insert()` construct against this `TableClause`.

E.g.:

```
table.insert().values(name='foo')
```

See `insert()` for argument and usage information.

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*, *full=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN. Implies `FromClause.join.isouter`.

  New in version 1.1.

See also:

`join()` - standalone function

`Join` - the type of object produced

key

Return the 'key' for this `Table`.

This value is used as the dictionary key within the `MetaData.tables` collection. It is typically the same as that of `Table.name` for a table with no `Table.schema` set; otherwise it is typically of the form `schemaname.tablename`.

kwargs

A synonym for `DialectKWArgs.dialect_kwargs`.

lateral(*name=None*)

Return a LATERAL alias of this `FromClause`.

The return value is the `Lateral` construct also provided by the top-level `lateral()` function.

New in version 1.1.

See also:

lateral_selects - overview of usage.

outerjoin(*right*, *onclause=None*, *full=False*)

Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.

- **onclause** – a SQL expression representing the ON clause of the join. If left at None, FromClause.join() will attempt to join the two tables based on a foreign key relationship.
- **full** – if True, render a FULL OUTER JOIN, instead of LEFT OUTER JOIN.

  New in version 1.1.

See also:

FromClause.join()

Join

primary_key
Return the collection of Column objects which comprise the primary key of this FromClause.

quote
Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

Deprecated since version 0.9: Use <obj>.name.quote

quote_schema
Return the value of the quote_schema flag passed to this Table.

Deprecated since version 0.9: Use table.schema.quote

replace_selectable(*sqlutil*, *old*, *alias*)
replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this FromClause.

select(*whereclause=None*, ***params*)
return a SELECT of this FromClause.

See also:

select() - general purpose method which allows for arbitrary column lists.

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.

tablesample(*sampling*, *name=None*, *seed=None*)
Return a TABLESAMPLE alias of this FromClause.

The return value is the TableSample construct also provided by the top-level tablesample() function.

New in version 1.1.

See also:

tablesample() - usage guidelines and parameters

tometadata(*metadata*, *schema=symbol('retain_schema')*, *referred_schema_fn=None*, *name=None*)

Return a copy of this `Table` associated with a different `MetaData`.

E.g.:

```
m1 = MetaData()

user = Table('user', m1, Column('id', Integer, priamry_key=True))

m2 = MetaData()
user_copy = user.tometadata(m2)
```

Parameters

- **metadata** – Target `MetaData` object, into which the new `Table` object will be created.
- **schema** – optional string name indicating the target schema. Defaults to the special symbol `RETAIN_SCHEMA` which indicates that no change to the schema name should be made in the new `Table`. If set to a string name, the new `Table` will have this new name as the `.schema`. If set to `None`, the schema will be set to that of the schema set on the target `MetaData`, which is typically `None` as well, unless set explicitly:

```
m2 = MetaData(schema='newschema')

# user_copy_one will have "newschema" as the schema name
user_copy_one = user.tometadata(m2, schema=None)

m3 = MetaData()  # schema defaults to None

# user_copy_two will have None as the schema name
user_copy_two = user.tometadata(m3, schema=None)
```

- **referred_schema_fn** – optional callable which can be supplied in order to provide for the schema name that should be assigned to the referenced table of a `ForeignKeyConstraint`. The callable accepts this parent `Table`, the target schema that we are changing to, the `ForeignKeyConstraint` object, and the existing "target schema" of that constraint. The function should return the string schema name that should be applied. E.g.:

```
def referred_schema_fn(table, to_schema,
                                constraint, referred_schema):
    if referred_schema == 'base_tables':
        return referred_schema
    else:
        return to_schema

new_table = table.tometadata(m2, schema="alt_schema",
                        referred_schema_fn=referred_schema_fn)
```

  New in version 0.9.2.

- **name** – optional string name indicating the target table name. If not specified or None, the table name is retained. This allows a `Table` to be copied to the same `MetaData` target with a new name.

  New in version 1.0.0.

update(*dml*, *whereclause=None*, *values=None*, *inline=False*, ***kwargs*)

Generate an `update()` construct against this `TableClause`.

E.g.:

```
table.update().where(table.c.id==7).values(name='foo')
```

See `update()` for argument and usage information.

class sqlalchemy.schema.ThreadLocalMetaData

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`.

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.

3.3.2 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*. In the most simple case 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=engine)
>>> '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 key
... 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*)

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.

A `Inspector` object is usually created via the `inspect()` function:

```
from sqlalchemy import inspect, create_engine
engine = create_engine('...')
insp = inspect(engine)
```

The inspection method above is equivalent to using 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.

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_check_constraints(*table_name*, *schema=None*, ***kw*)

Return information about check constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return check constraint information as a list of dicts with these keys:

name the check constraint's name

sqltext the check constraint's SQL expression

Parameters

- **table_name** – string name of the table. For special quoting, use `quoted_name`.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

New in version 1.1.0.

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` - the type of this column; an instance of `TypeEngine`
- `nullable` - boolean flag if the column is NULL or NOT NULL
- `default` - the column's server default value - this is returned as a string SQL expression.
- `attrs` - dict containing optional column attributes

Parameters

- **`table_name`** – string name of the table. For special quoting, use `quoted_name`.
- **`schema`** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

Returns list of dictionaries, each representing the definition of a database column.

`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.

Parameters

- **`table_name`** – string name of the table. For special quoting, use `quoted_name`.
- **`schema`** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

`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

dialect_options dict of dialect-specific index options. May not be present for all dialects.

New in version 1.0.0.

Parameters

- **`table_name`** – string name of the table. For special quoting, use `quoted_name`.
- **`schema`** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

`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.

Parameters

- `table_name` – string name of the table. For special quoting, use `quoted_name`.
- `schema` – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

`get_primary_keys`(*table_name*, *schema=None*, ***kw*)
Return information about primary keys in *table_name*.

Deprecated since version 0.7: Call to deprecated method get_primary_keys. Use get_pk_constraint instead.

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_sorted_table_and_fkc_names`(*schema=None*)
Return dependency-sorted table and foreign key constraint names in referred to within a particular schema.

This will yield 2-tuples of `(tablename, [(tname, fkname), (tname, fkname), ...])` consisting of table names in CREATE order grouped with the foreign key constraint names that are not detected as belonging to a cycle. The final element will be `(None, [(tname, fkname), (tname, fkname), ..])` which will consist of remaining foreign key constraint names that would require a separate CREATE step after-the-fact, based on dependencies between tables.

New in version 1.0.-.

See also:

`Inspector.get_table_names()`

`sort_tables_and_constraints()` **- similar method which works** with an already-given `MetaData`.

`get_table_comment`(*table_name*, *schema=None*, ***kw*)
Return information about the table comment for `table_name`.

Given a string `table_name` and an optional string `schema`, return table comment information as a dictionary with these keys:

text text of the comment.

Raises `NotImplementedError` for a dialect that does not support comments.

New in version 1.2.

`get_table_names`(*schema=None*, *order_by=None*)
Return all table names in referred to within a particular schema.

The names are expected to be real tables only, not views. Views are instead returned using the `Inspector.get_view_names()` method.

Parameters

- **schema** – Schema name. If schema is left at None, the database's default schema is used, else the named schema is searched. If the database does not support named schemas, behavior is undefined if schema is not passed as None. For special quoting, use quoted_name.
- **order_by** – Optional, may be the string "foreign_key" to sort the result on foreign key dependencies. Does not automatically resolve cycles, and will raise CircularDependencyError if cycles exist.

  Deprecated since version 1.0.0: - see Inspector.get_sorted_table_and_fkc_names() for a version of this which resolves foreign key cycles between tables automatically.

  Changed in version 0.8: the "foreign_key" sorting sorts tables in order of dependee to dependent; that is, in creation order, rather than in drop order. This is to maintain consistency with similar features such as MetaData.sorted_tables and util.sort_tables().

See also:

Inspector.get_sorted_table_and_fkc_names()

MetaData.sorted_tables

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.

Parameters

- **table_name** – string name of the table. For special quoting, use quoted_name.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use quoted_name.

get_temp_table_names()
return a list of temporary table names for the current bind.

This method is unsupported by most dialects; currently only SQLite implements it.

New in version 1.0.0.

get_temp_view_names()
return a list of temporary view names for the current bind.

This method is unsupported by most dialects; currently only SQLite implements it.

New in version 1.0.0.

get_unique_constraints(*table_name*, *schema=None*, ***kw*)
Return information about unique constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return unique constraint information as a list of dicts with these keys:

name the unique constraint's name

column_names list of column names in order

Parameters

- **table_name** – string name of the table. For special quoting, use quoted_name.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use quoted_name.

New in version 0.8.4.

get_view_definition(*view_name*, *schema=None*)
Return definition for *view_name*.

Parameters **schema** – Optional, retrieve names from a non-default schema. For special quoting, use **quoted_name**.

get_view_names(*schema=None*)
Return all view names in *schema*.

Parameters **schema** – Optional, retrieve names from a non-default schema. For special quoting, use **quoted_name**.

reflecttable(*table*, *include_columns*, *exclude_columns=()*, *_extend_on=None*)
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.reflection import Inspector

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.

Limitations of Reflection

It's important to note that the reflection process recreates Table metadata using only information which is represented in the relational database. This process by definition cannot restore aspects of a schema that aren't actually stored in the database. State which is not available from reflection includes but is not limited to:

- Client side defaults, either Python functions or SQL expressions defined using the default keyword of Column (note this is separate from server_default, which specifically is what's available via reflection).
- Column information, e.g. data that might have been placed into the Column.info dictionary
- The value of the .quote setting for Column or Table
- The association of a particular Sequence with a given Column

The relational database also in many cases reports on table metadata in a different format than what was specified in SQLAlchemy. The Table objects returned from reflection cannot be always relied upon to produce the identical DDL as the original Python-defined Table objects. Areas where this occurs includes server defaults, column-associated sequences and various idosyncrasies regarding constraints and datatypes. Server side defaults may be returned with cast directives (typically PostgreSQL will include a ::<type> cast) or different quoting patterns than originally specified.

Another category of limitation includes schema structures for which reflection is only partially or not yet defined. Recent improvements to reflection allow things like views, indexes and foreign key options to be reflected. As of this writing, structures like CHECK constraints, table comments, and triggers are not reflected.

3.3.3 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 `Column.default` and `Column.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 `Column.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 `Column.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 `Column.default` and `Column.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.get_current_parameters()['counter'] + 12

t = Table('mytable', meta,
    Column('counter', Integer),
    Column('counter_plus_twelve', Integer, default=mydefault, onupdate=mydefault)
)
```

The above default generation function is applied so that it 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.

For a single statement that is being executed using "executemany" style, e.g. with multiple parameter sets passed to `Connection.execute()`, the user- defined function is called once for each set of parameters. For the use case of a multi-valued `Insert` construct (e.g. with more than one VALUES clause set up via the `Insert.values()` method), the user-defined function is also called once for each set of parameters.

When the function is invoked, the special method `DefaultExecutionContext.get_current_parameters()` is available from the context object (an subclass of `DefaultExecutionContext`). This method returns a dictionary of column-key to values that represents the full set of values for the INSERT or UPDATE statement. In the case of a multi-valued INSERT construct, the subset of parameters that corresponds to the individual VALUES clause is isolated from the full parameter dictionary and returned alone.

New in version 1.2: Added `DefaultExecutionContext.get_current_parameters()` method, which improves upon the still-present `DefaultExecutionContext.current_parameters` attribute by offering the service of organizing multiple VALUES clauses into individual parameter dictionaries.

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 `ResultProxy.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 `ResultProxy.last_inserted_params()` or `ResultProxy.last_updated_params()` collections on `ResultProxy`. The `ResultProxy.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 `Column.server_default`, which gets placed in the CREATE TABLE statement during a `Table.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 `Column.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 using `FetchedValue` as 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.

Note: It's generally not appropriate to use `FetchedValue` in conjunction with a primary key column, particularly when using the ORM or any other scenario where the `ResultProxy.inserted_primary_key` attribute is required. This is becaue the "post-fetch" operation requires that the primary key value already be available, so that the row can be selected on its primary key.

For a server-generated primary key value, all databases provide special accessors or other techniques in order to acquire the "last inserted primary key" column of a table. These mechanisms aren't affected by the presence of `FetchedValue`. For special situations where triggers are used to generate primary key values, and the database in use does not support the `RETURNING` clause, it may be necessary to forego the usage of the trigger and instead apply the SQL expression or function as a "pre execute" expression:

```
t = Table('test', meta,
        Column('abc', MyType, default=func.generate_new_value(), primary_key=True)
)
```

Where above, when `Table.insert()` is used, the `func.generate_new_value()` expression will be pre-executed in the context of a scalar `SELECT` statement, and the new value will be applied to the subsequent `INSERT`, while at the same time being made available to the `ResultProxy.inserted_primary_key` attribute.

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', metadata=meta), 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. Typically, the sequence function is embedded in the INSERT statement, which is combined with RETURNING so that the newly generated value can be returned to the Python code:

```
INSERT INTO cartitems (cart_id, description, createdate)
VALUES (next_val(cart_id_seq), 'some description', '2015-10-15 12:00:15')
RETURNING cart_id
```

When the `Sequence` is associated with a `Column` as its **Python-side** default generator, the `Sequence` will also be subject to "CREATE SEQUENCE" and "DROP SEQUENCE" DDL when similar DDL is emitted for the owning `Table`. This is a limited scope convenience feature that does not accommodate for inheritance of other aspects of the `MetaData`, such as the default schema. Therefore, it is best practice that for a `Sequence` which is local to a certain `Column` / `Table`, that it be explicitly associated with the `MetaData` using the `Sequence.metadata` parameter. See the section sequence_metadata for more background on this.

Associating a Sequence on a SERIAL column

PostgreSQL's SERIAL datatype is an auto-incrementing type that implies the implicit creation of a PostgreSQL sequence when CREATE TABLE is emitted. If a `Column` specifies an explicit `Sequence` object which also specifies a true value for the `Sequence.optional` boolean flag, the `Sequence` will not take effect under PostgreSQL, and the SERIAL datatype will proceed normally. Instead, the `Sequence` will only take effect when used against other sequence-supporting databases, currently Oracle and Firebird.

Executing a Sequence Standalone

A SEQUENCE is a first class schema object in SQL and can be used to generate values independently in the database. If you have a `Sequence` object, it can be invoked with its "next value" instruction by passing it directly to a SQL execution method:

```
with my_engine.connect() as conn:
    seq = Sequence('some_sequence')
    nextid = conn.execute(seq)
```

In order to embed the "next value" function of a `Sequence` inside of a SQL statement like a SELECT or INSERT, use the `Sequence.next_value()` method, which will render at statement compilation time a SQL function that is appropriate for the target backend:

```
>>> my_seq = Sequence('some_sequence')
>>> stmt = select([my_seq.next_value()])
>>> print stmt.compile(dialect=postgresql.dialect())
SELECT nextval('some_sequence') AS next_value_1
```

Associating a Sequence with the MetaData

For many years, the SQLAlchemy documentation referred to the example of associating a `Sequence` with a table as follows:

```
table = Table("cartitems", meta,
    Column("cart_id", Integer, Sequence('cart_id_seq'),
           primary_key=True),
    Column("description", String(40)),
    Column("createdate", DateTime())
)
```

While the above is a prominent idiomatic pattern, it is recommended that the `Sequence` in most cases be explicitly associated with the `MetaData`, using the `Sequence.metadata` parameter:

```
table = Table("cartitems", meta,
    Column(
        "cart_id",
        Integer,
        Sequence('cart_id_seq', metadata=meta), primary_key=True),
    Column("description", String(40)),
    Column("createdate", DateTime())
)
```

The `Sequence` object is a first class schema construct that can exist independently of any table in a database, and can also be shared among tables. Therefore SQLAlchemy does not implicitly modify the `Sequence` when it is associated with a `Column` object as either the Python-side or server-side default generator. While the CREATE SEQUENCE / DROP SEQUENCE DDL is emitted for a `Sequence` defined as a Python side generator at the same time the table itself is subject to CREATE or DROP, this is a convenience feature that does not imply that the `Sequence` is fully associated with the `MetaData` object.

Explicitly associating the `Sequence` with `MetaData` allows for the following behaviors:

- The `Sequence` will inherit the `MetaData.schema` parameter specified to the target `MetaData`, which affects the production of CREATE / DROP DDL, if any.
- The `Sequence.create()` and `Sequence.drop()` methods automatically use the engine bound to the `MetaData` object, if any.
- The `MetaData.create_all()` and `MetaData.drop_all()` methods will emit CREATE / DROP for this `Sequence`, even if the `Sequence` is not associated with any `Table` / `Column` that's a member of this `MetaData`.

Since the vast majority of cases that deal with `Sequence` expect that `Sequence` to be fully "owned" by the assocated `Table` and that options like default schema are propagated, setting the `Sequence.metadata` parameter should be considered a best practice.

Associating a Sequence as the Server Side Default

The preceding sections illustrate how to associate a `Sequence` with a `Column` as the **Python side default generator**:

```
Column(
    "cart_id", Integer, Sequence('cart_id_seq', metadata=meta),
    primary_key=True)
```

In the above case, the `Sequence` will automatically be subject to CREATE SEQUENCE / DROP SEQUENCE DDL when the related `Table` is subject to CREATE / DROP. However, the sequence will **not** be present as the server-side default for the column when CREATE TABLE is emitted.

If we want the sequence to be used as a server-side default, meaning it takes place even if we emit INSERT commands to the table from the SQL command line, we can use the `Column.server_default` parameter in conjunction with the value-generation function of the sequence, available from the `Sequence.next_value()` method. Below we illustrate the same `Sequence` being associated with the `Column` both as the Python-side default generator as well as the server-side default generator:

```
cart_id_seq = Sequence('cart_id_seq', metadata=meta)
table = Table("cartitems", meta,
    Column(
        "cart_id", Integer, cart_id_seq,
        server_default=cart_id_seq.next_value(), primary_key=True),
    Column("description", String(40)),
    Column("createdate", DateTime())
)
```

or with the ORM:

```
class CartItem(Base):
    __tablename__ = 'cartitems'

    cart_id_seq = Sequence('cart_id_seq', metadata=Base.metadata)
    cart_id = Column(
        Integer, cart_id_seq,
        server_default=cart_id_seq.next_value(), primary_key=True)
    description = Column(String(40))
    createdate = Column(DateTime)
```

When the "CREATE TABLE" statement is emitted, on PostgreSQL it would be emitted as:

```
CREATE TABLE cartitems (
    cart_id INTEGER DEFAULT nextval('cart_id_seq') NOT NULL,
    description VARCHAR(40),
    createdate TIMESTAMP WITHOUT TIME ZONE,
    PRIMARY KEY (cart_id)
)
```

Placement of the `Sequence` in both the Python-side and server-side default generation contexts ensures that the "primary key fetch" logic works in all cases. Typically, sequence-enabled databases also support RETURNING for INSERT statements, which is used automatically by SQLAlchemy when emitting this statement. However if RETURNING is not used for a particular insert, then SQLAlchemy would prefer to "pre-execute" the sequence outside of the INSERT statement itself, which only works if the sequence is included as the Python-side default generator function.

The example also associates the `Sequence` with the enclosing `MetaData` directly, which again ensures that the `Sequence` is fully associated with the parameters of the `MetaData` collection including the default schema, if any.

See also:

Sequences/SERIAL/IDENTITY - in the PostgreSQL dialect documentation

RETURNING Support - in the Oracle dialect documentation

Default Objects API

`class sqlalchemy.schema.ColumnDefault(`*arg*, ***kwargs*`)`

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*, *_reflected=False*)
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*)
Base class for column *default* values.

class sqlalchemy.schema.**FetchedValue**(*for_update=False*)
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.

See also:

triggered_columns

class sqlalchemy.schema.**PassiveDefault**(**arg*, ***kw*)
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*, *minvalue=None*, *maxvalue=None*, *nominvalue=None*, *nomaxvalue=None*, *cycle=None*, *schema=None*, *cache=None*, *order=None*, *optional=False*, *quote=None*, *metadata=None*, *quote_schema=None*, *for_update=False*)
Represents a named database sequence.

The `Sequence` object represents the name and configurational parameters of a database sequence. It also represents a construct that can be "executed" by a SQLAlchemy `Engine` or `Connection`, rendering the appropriate "next value" function for the target database and returning a result.

The `Sequence` is typically associated with a primary key column:

```
some_table = Table(
    'some_table', metadata,
    Column('id', Integer, Sequence('some_table_seq'),
    primary_key=True)
)
```

When CREATE TABLE is emitted for the above `Table`, if the target platform supports sequences, a CREATE SEQUENCE statement will be emitted as well. For platforms that don't support sequences, the `Sequence` construct is ignored.

See also:

`CreateSequence`

`DropSequence`

`create`(*bind=None*, *checkfirst=True*)
Creates this sequence in the database.

`drop`(*bind=None*, *checkfirst=True*)
Drops this sequence from the database.

`next_value`(*func*)
Return a `next_value` function element which will render the appropriate increment function for this `Sequence` within any SQL expression.

3.3.4 Defining Constraints and Indexes

This section will discuss SQL constraints and indexes. In SQLAlchemy the key classes include `ForeignKeyConstraint` and `Index`.

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

The behavior we've seen in tutorials and elsewhere involving foreign keys with DDL illustrates that the constraints are typically rendered "inline" within the CREATE TABLE statement, such as:

```
CREATE TABLE addresses (
    id INTEGER NOT NULL,
    user_id INTEGER,
    email_address VARCHAR NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT user_id_fk FOREIGN KEY(user_id) REFERENCES users (id)
)
```

The `CONSTRAINT .. FOREIGN KEY` directive is used to create the constraint in an "inline" fashion within the CREATE TABLE definition. The `MetaData.create_all()` and `MetaData.drop_all()` methods do this by default, using a topological sort of all the `Table` objects involved such that tables are created and dropped in order of their foreign key dependency (this sort is also available via the `MetaData.sorted_tables` accessor).

This approach can't work when two or more foreign key constraints are involved in a "dependency cycle", where a set of tables are mutually dependent on each other, assuming the backend enforces foreign keys (always the case except on SQLite, MySQL/MyISAM). The methods will therefore break out constraints in such a cycle into separate ALTER statements, on all backends other than SQLite which does not support most forms of ALTER. Given a schema like:

```
node = Table(
    'node', metadata,
    Column('node_id', Integer, primary_key=True),
    Column(
        'primary_element', Integer,
        ForeignKey('element.element_id')
    )
)
```

```
element = Table(
    'element', metadata,
    Column('element_id', Integer, primary_key=True),
    Column('parent_node_id', Integer),
    ForeignKeyConstraint(
        ['parent_node_id'], ['node.node_id'],
        name='fk_element_parent_node_id'
    )
)
```

When we call upon `MetaData.create_all()` on a backend such as the PostgreSQL backend, the cycle between these two tables is resolved and the constraints are created separately:

```
>>> with engine.connect() as conn:
...     metadata.create_all(conn, checkfirst=False)
{opensql}CREATE TABLE element (
    element_id SERIAL NOT NULL,
    parent_node_id INTEGER,
    PRIMARY KEY (element_id)
)

CREATE TABLE node (
    node_id SERIAL NOT NULL,
    primary_element INTEGER,
    PRIMARY KEY (node_id)
)

ALTER TABLE element ADD CONSTRAINT fk_element_parent_node_id
    FOREIGN KEY(parent_node_id) REFERENCES node (node_id)
ALTER TABLE node ADD FOREIGN KEY(primary_element)
    REFERENCES element (element_id)
{stop}
```

In order to emit DROP for these tables, the same logic applies, however note here that in SQL, to emit DROP CONSTRAINT requires that the constraint has a name. In the case of the `'node'` table above, we haven't named this constraint; the system will therefore attempt to emit DROP for only those constraints that are named:

```
>>> with engine.connect() as conn:
...     metadata.drop_all(conn, checkfirst=False)
{opensql}ALTER TABLE element DROP CONSTRAINT fk_element_parent_node_id
DROP TABLE node
DROP TABLE element
{stop}
```

In the case where the cycle cannot be resolved, such as if we hadn't applied a name to either constraint here, we will receive the following error:

```
sqlalchemy.exc.CircularDependencyError: Can't sort tables for DROP;
an unresolvable foreign key dependency exists between tables:
element, node.  Please ensure that the ForeignKey and ForeignKeyConstraint
objects involved in the cycle have names so that they can be dropped
using DROP CONSTRAINT.
```

This error only applies to the DROP case as we can emit "ADD CONSTRAINT" in the CREATE case without a name; the database typically assigns one automatically.

The `ForeignKeyConstraint.use_alter` and `ForeignKey.use_alter` keyword arguments can be used to manually resolve dependency cycles. We can add this flag only to the `'element'` table as follows:

```
element = Table(
    'element', metadata,
```

```
    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'
    )
)
```

in our CREATE DDL we will see the ALTER statement only for this constraint, and not the other one:

```
>>> with engine.connect() as conn:
...     metadata.create_all(conn, checkfirst=False)
{opensql}CREATE TABLE element (
    element_id SERIAL NOT NULL,
    parent_node_id INTEGER,
    PRIMARY KEY (element_id)
)

CREATE TABLE node (
    node_id SERIAL NOT NULL,
    primary_element INTEGER,
    PRIMARY KEY (node_id),
    FOREIGN KEY(primary_element) REFERENCES element (element_id)
)

ALTER TABLE element ADD CONSTRAINT fk_element_parent_node_id
FOREIGN KEY(parent_node_id) REFERENCES node (node_id)
{stop}
```

`ForeignKeyConstraint.use_alter` and `ForeignKey.use_alter`, when used in conjunction with a drop operation, will require that the constraint is named, else an error like the following is generated:

```
sqlalchemy.exc.CompileError: Can't emit DROP CONSTRAINT for constraint
ForeignKeyConstraint(...); it has no name
```

Changed in version 1.0.0: - The DDL system invoked by `MetaData.create_all()` and `MetaData.drop_all()` will now automatically resolve mutually depdendent foreign keys between tables declared by `ForeignKeyConstraint` and `ForeignKey` objects, without the need to explicitly set the `ForeignKeyConstraint.use_alter` flag.

Changed in version 1.0.0: - The `ForeignKeyConstraint.use_alter` flag can be used with an un-named constraint; only the DROP operation will emit a specific error when actually called upon.

See also:

constraint_naming_conventions

`sort_tables_and_constraints()`

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.

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.

```
from sqlalchemy import UniqueConstraint

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')
    )
```

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.

```
from sqlalchemy import CheckConstraint

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')
    )

{sql}mytable.create(engine)
CREATE TABLE mytable (
    col1 INTEGER  CHECK (col1>5),
    col2 INTEGER,
    col3 INTEGER,
    CONSTRAINT check1  CHECK (col2 > col3 + 5)
){stop}
```

PRIMARY KEY Constraint

The primary key constraint of any `Table` object is implicitly present, based on the `Column` objects that are marked with the `Column.primary_key` flag. The `PrimaryKeyConstraint` object provides explicit access to this constraint, which includes the option of being configured directly:

```
from sqlalchemy import PrimaryKeyConstraint

my_table = Table('mytable', metadata,
            Column('id', Integer),
            Column('version_id', Integer),
            Column('data', String(50)),
            PrimaryKeyConstraint('id', 'version_id', name='mytable_pk')
        )
```

See also:

`PrimaryKeyConstraint` - detailed API documentation.

Setting up Constraints when using the Declarative ORM Extension

The `Table` is the SQLAlchemy Core construct that allows one to define table metadata, which among other things can be used by the SQLAlchemy ORM as a target to map a class. The Declarative extension allows the `Table` object to be created automatically, given the contents of the table primarily as a mapping of `Column` objects.

To apply table-level constraint objects such as `ForeignKeyConstraint` to a table defined using Declarative, use the `__table_args__` attribute, described at declarative_table_args.

Configuring Constraint Naming Conventions

Relational databases typically assign explicit names to all constraints and indexes. In the common case that a table is created using `CREATE TABLE` where constraints such as CHECK, UNIQUE, and PRIMARY KEY constraints are produced inline with the table definition, the database usually has a system in place in which names are automatically assigned to these constraints, if a name is not otherwise specified. When an existing database table is altered in a database using a command such as `ALTER TABLE`, this command typically needs to specify explicit names for new constraints as well as be able to specify the name of an existing constraint that is to be dropped or modified.

Constraints can be named explicitly using the `Constraint.name` parameter, and for indexes the `Index.name` parameter. However, in the case of constraints this parameter is optional. There are also the use cases of using the `Column.unique` and `Column.index` parameters which create `UniqueConstraint` and `Index` objects without an explicit name being specified.

The use case of alteration of existing tables and constraints can be handled by schema migration tools such as Alembic. However, neither Alembic nor SQLAlchemy currently create names for constraint objects where the name is otherwise unspecified, leading to the case where being able to alter existing constraints means that one must reverse-engineer the naming system used by the relational database to auto-assign names, or that care must be taken to ensure that all constraints are named.

In contrast to having to assign explicit names to all `Constraint` and `Index` objects, automated naming schemes can be constructed using events. This approach has the advantage that constraints will get a consistent naming scheme without the need for explicit name parameters throughout the code, and also that the convention takes place just as well for those constraints and indexes produced by the `Column.unique` and `Column.index` parameters. As of SQLAlchemy 0.9.2 this event-based approach is included, and can be configured using the argument `MetaData.naming_convention`.

`MetaData.naming_convention` refers to a dictionary which accepts the `Index` class or individual `Constraint` classes as keys, and Python string templates as values. It also accepts a series of string-codes as alternative keys, `"fk"`, `"pk"`, `"ix"`, `"ck"`, `"uq"` for foreign key, primary key, index, check, and unique constraint, respectively. The string templates in this dictionary are used whenever a constraint or index is associated with this `MetaData` object that does not have an existing name given (including one exception case where an existing name can be further embellished).

An example naming convention that suits basic cases is as follows:

```
convention = {
  "ix": 'ix_%(column_0_label)s',
  "uq": "uq_%(table_name)s_%(column_0_name)s",
  "ck": "ck_%(table_name)s_%(constraint_name)s",
  "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
  "pk": "pk_%(table_name)s"
}

metadata = MetaData(naming_convention=convention)
```

The above convention will establish names for all constraints within the target `MetaData` collection. For example, we can observe the name produced when we create an unnamed `UniqueConstraint`:

```
>>> user_table = Table('user', metadata,
...                 Column('id', Integer, primary_key=True),
...                 Column('name', String(30), nullable=False),
...                 UniqueConstraint('name')
... )
>>> list(user_table.constraints)[1].name
'uq_user_name'
```

This same feature takes effect even if we just use the `Column.unique` flag:

```
>>> user_table = Table('user', metadata,
...                  Column('id', Integer, primary_key=True),
...                  Column('name', String(30), nullable=False, unique=True)
...     )
>>> list(user_table.constraints)[1].name
'uq_user_name'
```

A key advantage to the naming convention approach is that the names are established at Python construction time, rather than at DDL emit time. The effect this has when using Alembic's `--autogenerate` feature is that the naming convention will be explicit when a new migration script is generated:

```
def upgrade():
    op.create_unique_constraint("uq_user_name", "user", ["name"])
```

The above `"uq_user_name"` string was copied from the `UniqueConstraint` object that `--autogenerate` located in our metadata.

The default value for `MetaData.naming_convention` handles the long-standing SQLAlchemy behavior of assigning a name to a `Index` object that is created using the `Column.index` parameter:

```
>>> from sqlalchemy.sql.schema import DEFAULT_NAMING_CONVENTION
>>> DEFAULT_NAMING_CONVENTION
immutabledict({'ix': 'ix_%(column_0_label)s'})
```

The tokens available include `%(table_name)s`, `%(referred_table_name)s`, `%(column_0_name)s`, `%(column_0_label)s`, `%(column_0_key)s`, `%(referred_column_0_name)s`, and `%(constraint_name)s`; the documentation for `MetaData.naming_convention` describes each individually. New tokens can also be added, by specifying an additional token and a callable within the naming_convention dictionary. For example, if we wanted to name our foreign key constraints using a GUID scheme, we could do that as follows:

```
import uuid

def fk_guid(constraint, table):
    str_tokens = [
        table.name,
    ] + [
        element.parent.name for element in constraint.elements
    ] + [
        element.target_fullname for element in constraint.elements
    ]
    guid = uuid.uuid5(uuid.NAMESPACE_OID, "_".join(str_tokens).encode('ascii'))
    return str(guid)

convention = {
    "fk_guid": fk_guid,
    "ix": 'ix_%(column_0_label)s',
    "fk": "fk_%(fk_guid)s",
}
```

Above, when we create a new `ForeignKeyConstraint`, we will get a name as follows:

```
>>> metadata = MetaData(naming_convention=convention)

>>> user_table = Table('user', metadata,
...         Column('id', Integer, primary_key=True),
...         Column('version', Integer, primary_key=True),
...         Column('data', String(30))
...     )
>>> address_table = Table('address', metadata,
...        Column('id', Integer, primary_key=True),
...        Column('user_id', Integer),
...        Column('user_version_id', Integer)
...    )
>>> fk = ForeignKeyConstraint(['user_id', 'user_version_id'],
...                ['user.id', 'user.version'])
>>> address_table.append_constraint(fk)
>>> fk.name
fk_0cd51ab5-8d70-56e8-a83c-86661737766d
```

See also:

`MetaData.naming_convention` - for additional usage details as well as a listing of all available naming components.

The Importance of Naming Constraints - in the Alembic documentation.

New in version 0.9.2: Added the `MetaData.naming_convention` argument.

Naming CHECK Constraints

The `CheckConstraint` object is configured against an arbitrary SQL expression, which can have any number of columns present, and additionally is often configured using a raw SQL string. Therefore a common convention to use with `CheckConstraint` is one where we expect the object to have a name already, and we then enhance it with other convention elements. A typical convention is `"ck_%(table_name)s_%(constraint_name)s"`:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(constraint_name)s"}
)

Table('foo', metadata,
    Column('value', Integer),
    CheckConstraint('value > 5', name='value_gt_5')
)
```

The above table will produce the name `ck_foo_value_gt_5`:

```
CREATE TABLE foo (
    value INTEGER,
    CONSTRAINT ck_foo_value_gt_5 CHECK (value > 5)
)
```

`CheckConstraint` also supports the `%(columns_0_name)s` token; we can make use of this by ensuring we use a `Column` or `sql.expression.column()` element within the constraint's expression, either by declaring the constraint separate from the table:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(column_0_name)s"}
)

foo = Table('foo', metadata,
    Column('value', Integer)
)

CheckConstraint(foo.c.value > 5)
```

or by using a `sql.expression.column()` inline:

```
from sqlalchemy import column

metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(column_0_name)s"}
)

foo = Table('foo', metadata,
    Column('value', Integer),
    CheckConstraint(column('value') > 5)
)
```

Both will produce the name `ck_foo_value`:

```
CREATE TABLE foo (
    value INTEGER,
    CONSTRAINT ck_foo_value CHECK (value > 5)
)
```

The determination of the name of "column zero" is performed by scanning the given expression for column objects. If the expression has more than one column present, the scan does use a deterministic search, however the structure of the expression will determine which column is noted as "column zero".

New in version 1.0.0: The `CheckConstraint` object now supports the `column_0_name` naming convention token.

Configuring Naming for Boolean, Enum, and other schema types

The `SchemaType` class refers to type objects such as `Boolean` and `Enum` which generate a CHECK constraint accompanying the type. The name for the constraint here is most directly set up by sending the "name" parameter, e.g. `Boolean.name`:

```
Table('foo', metadata,
    Column('flag', Boolean(name='ck_foo_flag'))
)
```

The naming convention feature may be combined with these types as well, normally by using a convention which includes `%(constraint_name)s` and then applying a name to the type:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(constraint_name)s"}
)

Table('foo', metadata,
    Column('flag', Boolean(name='flag_bool'))
)
```

The above table will produce the constraint name `ck_foo_flag_bool`:

```
CREATE TABLE foo (
    flag BOOL,
    CONSTRAINT ck_foo_flag_bool CHECK (flag IN (0, 1))
)
```

The `SchemaType` classes use special internal symbols so that the naming convention is only determined at DDL compile time. On PostgreSQL, there's a native BOOLEAN type, so the CHECK constraint of `Boolean` is not needed; we are safe to set up a `Boolean` type without a name, even though a naming convention is in place for check constraints. This convention will only be consulted for the CHECK constraint if we run against a database without a native BOOLEAN type like SQLite or MySQL.

The CHECK constraint may also make use of the `column_0_name` token, which works nicely with `SchemaType` since these constraints have only one column:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(column_0_name)s"}
)

Table('foo', metadata,
    Column('flag', Boolean())
)
```

The above schema will produce:

```
CREATE TABLE foo (
    flag BOOL,
    CONSTRAINT ck_foo_flag CHECK (flag IN (0, 1))
)
```

Changed in version 1.0: Constraint naming conventions that don't include `%(constraint_name)s` again work with `SchemaType` constraints.

Constraints API

class sqlalchemy.schema.Constraint(*name=None*, *deferrable=None*, *initially=None*, *_create_rule=None*, *info=None*, *_type_bound=False*, ***dialect_kw*)

A table-level SQL constraint.

class sqlalchemy.schema.ColumnCollectionMixin(**columns*, ***kw*)

columns = None

A `ColumnCollection` of `Column` objects.

This collection represents the columns which are referred to by this object.

class sqlalchemy.schema.ColumnCollectionConstraint(**columns*, ***kw*)

A constraint that proxies a ColumnCollection.

argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

columns = None

A `ColumnCollection` representing the set of columns for this constraint.

contains_column(*col*)

Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a `ColumnCollection` of `Column` objects.

dialect_kwargs

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)
used to allow SchemaVisitor access

info
Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs
A synonym for `DialectKWArgs.dialect_kwargs`.

quote
Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

class sqlalchemy.schema.CheckConstraint(*sqltext*, *name=None*, *deferrable=None*, *initially=None*, *table=None*, *info=None*, *_create_rule=None*, *_autoattach=True*, *_type_bound=False*)
A table- or column-level CHECK constraint.

Can be included in the definition of a Table or Column.

argument_for(*dialect_name*, *argument_name*, *default*)
Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a NoSuchModuleError is raised. The dialect must also include an existing DefaultDialect.construct_arguments collection, indicating that it participates in the keyword-argument validation and default system, else ArgumentError is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

contains_column(*col*)
Return True if this constraint contains the given column.

Note that this object also contains an attribute .columns which is a ColumnCollection of Column objects.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>_<kwarg> format. Only arguments that were actually passed are included; unlike the DialectKWArgs.dialect_options collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>_<kwarg> where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The DialectKWArgs.dialect_kwargs collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect_name> and <argument_name>. For example, the postgresql_where argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

get_children(***kwargs*)
used to allow SchemaVisitor access

info
Info dictionary associated with the object, allowing user-defined data to be associated with this SchemaItem.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as Table and Column.

kwargs
A synonym for DialectKWArgs.dialect_kwargs.

quote

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

class sqlalchemy.schema.ForeignKey(*column*, *_constraint=None*, *use_alter=False*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *link_to_name=False*, *match=None*, *info=None*, ***dialect_kw*)

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 metadata_foreignkeys.

argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

column
Return the target Column referenced by this ForeignKey.

If no target column has been established, an exception is raised.

Changed in version 0.9.0: Foreign key target column resolution now occurs as soon as both the ForeignKey object and the remote Column to which it refers are both associated with the same MetaData object.

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.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>_<kwarg> format. Only arguments that were actually passed are included; unlike the DialectKWArgs.dialect_options collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>_<kwarg> where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The DialectKWArgs.dialect_kwargs collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect_name> and <argument_name>. For example, the postgresql_where argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

get_children(***kwargs*)
used to allow SchemaVisitor access

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.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this SchemaItem.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as Table and Column.

kwargs
A synonym for DialectKWArgs.dialect_kwargs.

quote
Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

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*, *match=None*, *table=None*, *info=None*, ***dialect_kw*)

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 metadata_foreignkeys.

argument_for(*dialect_name*, *argument_name*, *default*)
Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

column_keys
Return a list of string keys representing the local columns in this `ForeignKeyConstraint`.

This list is either the original string arguments sent to the constructor of the ForeignKeyConstraint, or if the constraint has been initialized with Column objects, is the string .key of each element.

New in version 1.0.0.

columns = None
A ColumnCollection representing the set of columns for this constraint.

contains_column(*col*)
Return True if this constraint contains the given column.

Note that this object also contains an attribute .columns which is a ColumnCollection of Column objects.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original <dialect>_<kwarg> format. Only arguments that were actually passed are included; unlike the DialectKWArgs.dialect_options collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form <dialect>_<kwarg> where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The DialectKWArgs.dialect_kwargs collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to <dialect_name> and <argument_name>. For example, the postgresql_where argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

elements = None
A sequence of ForeignKey objects.

Each ForeignKey represents a single referring column/referred column pair.

This collection is intended to be read-only.

get_children(***kwargs*)
used to allow SchemaVisitor access

info
Info dictionary associated with the object, allowing user-defined data to be associated with this SchemaItem.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as Table and Column.

kwargs
A synonym for DialectKWArgs.dialect_kwargs.

quote
Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

Deprecated since version 0.9: Use `<obj>.name.quote`

referred_table
The `Table` object to which this `ForeignKeyConstraint` references.

This is a dynamically calculated attribute which may not be available if the constraint and/or parent table is not yet associated with a metadata collection that contains the referred table.

New in version 1.0.0.

class sqlalchemy.schema.**PrimaryKeyConstraint**(**columns*, ***kw*)
A table-level PRIMARY KEY constraint.

The `PrimaryKeyConstraint` object is present automatically on any `Table` object; it is assigned a set of `Column` objects corresponding to those marked with the `Column.primary_key` flag:

```
>>> my_table = Table('mytable', metadata,
...                 Column('id', Integer, primary_key=True),
...                 Column('version_id', Integer, primary_key=True),
...                 Column('data', String(50))
...     )
>>> my_table.primary_key
PrimaryKeyConstraint(
    Column('id', Integer(), table=<mytable>,
           primary_key=True, nullable=False),
    Column('version_id', Integer(), table=<mytable>,
           primary_key=True, nullable=False)
)
```

The primary key of a `Table` can also be specified by using a `PrimaryKeyConstraint` object explicitly; in this mode of usage, the "name" of the constraint can also be specified, as well as other options which may be recognized by dialects:

```
my_table = Table('mytable', metadata,
            Column('id', Integer),
            Column('version_id', Integer),
            Column('data', String(50)),
            PrimaryKeyConstraint('id', 'version_id',
                                 name='mytable_pk')
        )
```

The two styles of column-specification should generally not be mixed. An warning is emitted if the columns present in the `PrimaryKeyConstraint` don't match the columns that were marked as `primary_key=True`, if both are present; in this case, the columns are taken strictly from the `PrimaryKeyConstraint` declaration, and those columns otherwise marked as `primary_key=True` are ignored. This behavior is intended to be backwards compatible with previous behavior.

Changed in version 0.9.2: Using a mixture of columns within a `PrimaryKeyConstraint` in addition to columns marked as `primary_key=True` now emits a warning if the lists don't match. The ultimate behavior of ignoring those columns marked with the flag only is currently maintained for backwards compatibility; this warning may raise an exception in a future release.

For the use case where specific options are to be specified on the `PrimaryKeyConstraint`, but the usual style of using `primary_key=True` flags is still desirable, an empty `PrimaryKeyConstraint` may be specified, which will take on the primary key column collection from the `Table` based on the flags:

```
my_table = Table('mytable', metadata,
            Column('id', Integer, primary_key=True),
            Column('version_id', Integer, primary_key=True),
            Column('data', String(50)),
            PrimaryKeyConstraint(name='mytable_pk',
                                 mssql_clustered=True)
        )
```

New in version 0.9.2: an empty `PrimaryKeyConstraint` may now be specified for the purposes of establishing keyword arguments with the constraint, independently of the specification of "primary key" columns within the `Table` itself; columns marked as `primary_key=True` will be gathered into the empty constraint's column collection.

argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

contains_column(*col*)

Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a `ColumnCollection` of `Column` objects.

dialect_kwargs

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)
used to allow SchemaVisitor access

info
Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs
A synonym for `DialectKWArgs.dialect_kwargs`.

quote
Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

class sqlalchemy.schema.**UniqueConstraint**(**columns*, ***kw*)
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.

argument_for(*dialect_name*, *argument_name*, *default*)
Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

contains_column(*col*)
Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a `ColumnCollection` of `Column` objects.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)
used to allow SchemaVisitor access

info
Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs
A synonym for `DialectKWArgs.dialect_kwargs`.

quote
Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

sqlalchemy.schema.conv(*value*, *quote=None*)
Mark a string indicating that a name has already been converted by a naming convention.

This is a string subclass that indicates a name that should not be subject to any further naming conventions.

E.g. when we create a `Constraint` using a naming convention as follows:

```
m = MetaData(naming_convention={
    "ck": "ck_%(table_name)s_%(constraint_name)s"
})
```

```
t = Table('t', m, Column('x', Integer),
                CheckConstraint('x > 5', name='x5'))
```

The name of the above constraint will be rendered as `"ck_t_x5"`. That is, the existing name `x5` is used in the naming convention as the `constraint_name` token.

In some situations, such as in migration scripts, we may be rendering the above `CheckConstraint` with a name that's already been converted. In order to make sure the name isn't double-modified, the new name is applied using the `schema.conv()` marker. We can use this explicitly as follows:

```
m = MetaData(naming_convention={
    "ck": "ck_%(table_name)s_%(constraint_name)s"
})
t = Table('t', m, Column('x', Integer),
                CheckConstraint('x > 5', name=conv('ck_t_x5')))
```

Where above, the `schema.conv()` marker indicates that the constraint name here is final, and the name will render as `"ck_t_x5"` and not `"ck_t_ck_t_x5"`

New in version 0.9.4.

See also:

constraint_naming_conventions

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.

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)

{sql}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){stop}
```

Note in the example above, the `Index` construct is created externally to the table which it corresponds, using `Column` objects directly. `Index` also supports "inline" definition inside the `Table`, using string names to identify columns:

```
meta = MetaData()
mytable = Table('mytable', meta,
    Column('col1', Integer),

    Column('col2', Integer),

    Column('col3', Integer),
    Column('col4', Integer),

    # place an index on col1, col2
    Index('idx_col12', 'col1', 'col2'),

    # place a unique index on col3, col4
    Index('idx_col34', 'col3', 'col4', unique=True)
)
```

New in version 0.7: Support of "inline" definition inside the `Table` for `Index`.

The `Index` object also supports its own `create()` method:

```
i = Index('someindex', mytable.c.col5)
{sql}i.create(engine)
CREATE INDEX someindex ON mytable (col5){stop}
```

Functional Indexes

`Index` supports SQL and function expressions, as supported by the target backend. To create an index against a column using a descending value, the `ColumnElement.desc()` modifier may be used:

```
from sqlalchemy import Index

Index('someindex', mytable.c.somecol.desc())
```

Or with a backend that supports functional indexes such as PostgreSQL, a "case insensitive" index can be created using the `lower()` function:

```
from sqlalchemy import func, Index

Index('someindex', func.lower(mytable.c.somecol))
```

New in version 0.8: `Index` supports SQL expressions and functions as well as plain columns.

Index API

class sqlalchemy.schema.**Index**(*name*, **expressions*, ***kw*)
A table-level INDEX.

Defines a composite (one or more column) INDEX.

E.g.:

```
sometable = Table("sometable", metadata,
                Column("name", String(50)),
                Column("address", String(100))
            )

Index("some_index", sometable.c.name)
```

For a no-frills, single column index, adding `Column` also supports `index=True`:

```
sometable = Table("sometable", metadata,
                Column("name", String(50), index=True)
            )
```

For a composite index, multiple columns can be specified:

```
Index("some_index", sometable.c.name, sometable.c.address)
```

Functional indexes are supported as well, typically by using the `func` construct in conjunction with table-bound `Column` objects:

```
Index("some_index", func.lower(sometable.c.name))
```

New in version 0.8: support for functional and expression-based indexes.

An `Index` can also be manually associated with a `Table`, either through inline declaration or using `Table.append_constraint()`. When this approach is used, the names of the indexed columns can be specified as strings:

```
Table("sometable", metadata,
                Column("name", String(50)),
                Column("address", String(100)),
                Index("some_index", "name", "address")
        )
```

To support functional or expression-based indexes in this form, the `text()` construct may be used:

```
from sqlalchemy import text

Table("sometable", metadata,
                Column("name", String(50)),
                Column("address", String(100)),
                Index("some_index", text("lower(name)"))
        )
```

New in version 0.9.5: the `text()` construct may be used to specify `Index` expressions, provided the `Index` is explicitly associated with the `Table`.

See also:

schema_indexes - General information on `Index`.

PostgreSQL-Specific Index Options - PostgreSQL-specific options available for the `Index` construct.

MySQL Specific Index Options - MySQL-specific options available for the `Index` construct.

Clustered Index Support - MSSQL-specific options available for the `Index` construct.

argument_for(*dialect_name*, *argument_name*, *default*)

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind
Return the connectable associated with this Index.

create(*bind=None*)
Issue a `CREATE` statement for this `Index`, using the given `Connectable` for connectivity.

See also:

`MetaData.create_all()`.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options
A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

drop(*bind=None*)
Issue a DROP statement for this Index, using the given Connectable for connectivity.

See also:

MetaData.drop_all().

get_children(***kwargs*)
used to allow SchemaVisitor access

info
Info dictionary associated with the object, allowing user-defined data to be associated with this SchemaItem.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as Table and Column.

kwargs
A synonym for DialectKWArgs.dialect_kwargs.

quote
Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

Deprecated since version 0.9: Use <obj>.name.quote

3.3.5 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.

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:

```
event.listen(
    metadata,
    "after_create",
    DDL("ALTER TABLE users ADD CONSTRAINT "
        "cst_user_name_length "
        " CHECK (length(user_name) >= 8)")
)
```

A more comprehensive method of creating libraries of DDL constructs is to use custom compilation - see sqlalchemy.ext.compiler_toplevel for details.

Controlling DDL Sequences

The DDL construct introduced previously also has the ability to be invoked conditionally based on inspection of the database. This feature is available using the DDLElement.execute_if() method. For example, if we wanted to create a trigger but only on the PostgreSQL backend, we could invoke this as:

```
mytable = Table(
    'mytable', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', String(50))
)

trigger = DDL(
    "CREATE TRIGGER dt_ins BEFORE INSERT ON mytable "
    "FOR EACH ROW BEGIN SET NEW.data='ins'; END"
)

event.listen(
    mytable,
    'after_create',
    trigger.execute_if(dialect='postgresql')
)
```

The `DDLElement.execute_if.dialect` keyword also accepts a tuple of string dialect names:

```
event.listen(
    mytable,
    "after_create",
    trigger.execute_if(dialect=('postgresql', 'mysql'))
)
event.listen(
    mytable,
    "before_drop",
    trigger.execute_if(dialect=('postgresql', 'mysql'))
)
```

The `DDLElement.execute_if()` method can also work against a callable function that will receive the database connection in use. In the example below, we use this to conditionally create a CHECK constraint, first looking within the PostgreSQL catalogs to see if it exists:

```
def should_create(ddl, target, connection, **kw):
    row = connection.execute(
        "select conname from pg_constraint where conname='%s'" %
        ddl.element.name).scalar()
    return not bool(row)

def should_drop(ddl, target, connection, **kw):
    return not should_create(ddl, target, connection, **kw)

event.listen(
    users,
    "after_create",
    DDL(
        "ALTER TABLE users ADD CONSTRAINT "
        "cst_user_name_length CHECK (length(user_name) >= 8)"
    ).execute_if(callable_=should_create)
)
event.listen(
    users,
    "before_drop",
    DDL(
        "ALTER TABLE users DROP CONSTRAINT cst_user_name_length"
    ).execute_if(callable_=should_drop)
)

{sql}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){stop}

{sql}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 users{stop}
```

Using the built-in DDLElement Classes

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
{sql}engine.execute(CreateTable(mytable))
CREATE TABLE mytable (
    col1 INTEGER,
    col2 INTEGER,
    col3 INTEGER,
    col4 INTEGER,
    col5 INTEGER,
    col6 INTEGER
){stop}
```

Above, the `CreateTable` construct works like any other expression construct (such as `select()`, `table.insert()`, etc.). All of SQLAlchemy's DDL oriented constructs are subclasses of the `DDLElement` base class; this is the base of all the objects corresponding to CREATE and DROP as well as ALTER, not only in SQLAlchemy but in Alembic Migrations as well. A full reference of available constructs is in schema_api_ddl.

User-defined DDL constructs may also be created as subclasses of `DDLElement` itself. The documentation in sqlalchemy.ext.compiler_toplevel has several examples of this.

The event-driven DDL system described in the previous section schema_ddl_sequences is available with other `DDLElement` objects as well. However, when dealing with the built-in constructs such as `CreateIndex`, `CreateSequence`, etc, the event system is of **limited** use, as methods like `Table.create()` and `MetaData.create_all()` will invoke these constructs unconditionally. In a future SQLAlchemy release, the DDL event system including conditional execution will taken into account for built-in constructs that currently invoke in all cases.

We can illustrate an event-driven example with the `AddConstraint` and `DropConstraint` constructs, as the event-driven system will work for CHECK and UNIQUE constraints, using these as we did in our previous example of `DDLElement.execute_if()`:

```
def should_create(ddl, target, connection, **kw):
    row = connection.execute(
        "select conname from pg_constraint where conname='%s'" %
        ddl.element.name).scalar()
    return not bool(row)

def should_drop(ddl, target, connection, **kw):
    return not should_create(ddl, target, connection, **kw)

event.listen(
    users,
    "after_create",
    AddConstraint(constraint).execute_if(callable_=should_create)
```

```
)
event.listen(
    users,
    "before_drop",
    DropConstraint(constraint).execute_if(callable_=should_drop)
)

{sql}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){stop}

{sql}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 users{stop}
```

While the above example is against the built-in `AddConstraint` and `DropConstraint` objects, the main usefulness of DDL events for now remains focused on the use of the `DDL` construct itself, as well as with user-defined subclasses of `DDLElement` that aren't already part of the `MetaData.create_all()`, `Table.create()`, and corresponding "drop" processes.

DDL Expression Constructs API

`sqlalchemy.schema.sort_tables`(*tables*, *skip_fn=None*, *extra_dependencies=None*)

sort a collection of `Table` objects based on dependency.

This is a dependency-ordered sort which will emit `Table` objects such that they will follow their dependent `Table` objects. Tables are dependent on another based on the presence of `ForeignKeyConstraint` objects as well as explicit dependencies added by `Table.add_is_dependent_on()`.

> **Warning:** The `sort_tables()` function cannot by itself accommodate automatic resolution of dependency cycles between tables, which are usually caused by mutually dependent foreign key constraints. To resolve these cycles, either the `ForeignKeyConstraint.use_alter` parameter may be appled to those constraints, or use the `sql.sort_tables_and_constraints()` function which will break out foreign key constraints involved in cycles separately.

Parameters

- **`tables`** – a sequence of `Table` objects.
- **`skip_fn`** – optional callable which will be passed a `ForeignKey` object; if it returns True, this constraint will not be considered as a dependency. Note this is **different** from the same parameter in `sort_tables_and_constraints()`, which is instead passed the owning `ForeignKeyConstraint` object.
- **`extra_dependencies`** – a sequence of 2-tuples of tables which will also be considered as dependent on each other.

See also:

`sort_tables_and_constraints()`

`MetaData.sorted_tables()` - uses this function to sort

sqlalchemy.schema.**sort_tables_and_constraints**(*tables*, *filter_fn=None*, *extra_dependencies=None*)

sort a collection of `Table` / `ForeignKeyConstraint` objects.

This is a dependency-ordered sort which will emit tuples of `(Table, [ForeignKeyConstraint, ...])` such that each `Table` follows its dependent `Table` objects. Remaining `ForeignKeyConstraint` objects that are separate due to dependency rules not satisfied by the sort are emitted afterwards as `(None, [ForeignKeyConstraint ...])`.

Tables are dependent on another based on the presence of `ForeignKeyConstraint` objects, explicit dependencies added by `Table.add_is_dependent_on()`, as well as dependencies stated here using the `skip_fn` and/or `extra_dependencies` parameters.

Parameters

- **tables** – a sequence of `Table` objects.
- **filter_fn** – optional callable which will be passed a `ForeignKeyConstraint` object, and returns a value based on whether this constraint should definitely be included or excluded as an inline constraint, or neither. If it returns False, the constraint will definitely be included as a dependency that cannot be subject to ALTER; if True, it will **only** be included as an ALTER result at the end. Returning None means the constraint is included in the table-based result unless it is detected as part of a dependency cycle.
- **extra_dependencies** – a sequence of 2-tuples of tables which will also be considered as dependent on each other.

New in version 1.0.0.

See also:

`sort_tables()`

class sqlalchemy.schema.**DDLElement**

Base class for DDL expression constructs.

This class is the base for the general purpose `DDL` class, as well as the various create/drop clause constructs such as `CreateTable`, `DropTable`, `AddConstraint`, etc.

`DDLElement` integrates closely with SQLAlchemy events, introduced in event_toplevel. An instance of one is itself an event receiving callable:

```
event.listen(
    users,
    'after_create',
    AddConstraint(constraint).execute_if(dialect='postgresql')
)
```

See also:

`DDL`

`DDLEvents`

event_toplevel

schema_ddl_sequences

against(*target*)

Return a copy of this DDL against a specific schema item.

bind

callable_ = None

dialect = None

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_name*, *target*)
Link execution of this DDL to the DDL lifecycle of a SchemaItem.

Deprecated since version 0.7: See `DDLEvents`, as well as `DDLElement.execute_if()`.

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.

execute_if(*dialect=None*, *callable_=None*, *state=None*)
Return a callable that will execute this DDLElement conditionally.

Used to provide a wrapper for event listening:

```
event.listen(
            metadata,
            'before_create',
            DDL("my_ddl").execute_if(dialect='postgresql')
        )
```

Parameters

- **dialect** – 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').execute_if(dialect='postgresql')
  ```

  If a tuple, specifies multiple dialect names:

  ```
  DDL('something').execute_if(dialect=('postgresql', 'mysql'))
  ```

- **callable_** – A callable, which will be invoked with four positional arguments as well as optional keyword arguments:

  ddl This DDL element.

target The `Table` or `MetaData` object which is the target of this event. May be None if the DDL is executed explicitly.

bind 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.

state Optional keyword argument - will be the `state` argument passed to this function.

checkfirst Keyword argument, will be True if the 'checkfirst' flag was set during the call to `create()`, `create_all()`, `drop()`, `drop_all()`.

If the callable returns a true value, the DDL statement will be executed.

- **state** – any value which will be passed to the callable_ as the `state` keyword argument.

See also:

`DDLEvents`

event_toplevel

on = None

target = None

class sqlalchemy.schema.**DDL**(*statement*, *on=None*, *context=None*, *bind=None*)

A literal DDL statement.

Specifies literal SQL DDL to be executed by the database. DDL objects function as DDL event listeners, and can be subscribed to those events listed in `DDLEvents`, using either `Table` or `MetaData` objects as targets. Basic templating support allows a single DDL instance to handle repetitive tasks for multiple tables.

Examples:

```
from sqlalchemy import event, DDL

tbl = Table('users', metadata, Column('uid', Integer))
event.listen(tbl, 'before_create', DDL('DROP TRIGGER users_trigger'))

spow = DDL('ALTER TABLE %(table)s SET secretpowers TRUE')
event.listen(tbl, 'after_create', spow.execute_if(dialect='somedb'))

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 substitutions noted above. Keys present in the context will override the standard substitutions.

class sqlalchemy.schema.**_CreateDropBase**(*element*, *on=None*, *bind=None*)

Base class for DDL constructs that represent CREATE and DROP or equivalents.

The common theme of _CreateDropBase is a single `element` attribute which refers to the element to be created or dropped.

class sqlalchemy.schema.**CreateTable**(*element*, *on=None*, *bind=None*, *include_foreign_key_constraints=None*)
Represent a CREATE TABLE statement.

class sqlalchemy.schema.**DropTable**(*element*, *on=None*, *bind=None*)
Represent a DROP TABLE statement.

class sqlalchemy.schema.**CreateColumn**(*element*)
Represent a `Column` as rendered in a CREATE TABLE statement, via the `CreateTable` construct.

This is provided to support custom column DDL within the generation of CREATE TABLE statements, by using the compiler extension documented in sqlalchemy.ext.compiler_toplevel to extend `CreateColumn`.

Typical integration is to examine the incoming `Column` object, and to redirect compilation if a particular flag or condition is found:

```
from sqlalchemy import schema
from sqlalchemy.ext.compiler import compiles

@compiles(schema.CreateColumn)
def compile(element, compiler, **kw):
    column = element.element

    if "special" not in column.info:
        return compiler.visit_create_column(element, **kw)

    text = "%s SPECIAL DIRECTIVE %s" % (
            column.name,
            compiler.type_compiler.process(column.type)
        )
    default = compiler.get_column_default_string(column)
    if default is not None:
        text += " DEFAULT " + default

    if not column.nullable:
        text += " NOT NULL"

    if column.constraints:
        text += " ".join(
                    compiler.process(const)
                    for const in column.constraints)
    return text
```

The above construct can be applied to a `Table` as follows:

```
from sqlalchemy import Table, Metadata, Column, Integer, String
from sqlalchemy import schema

metadata = MetaData()

table = Table('mytable', MetaData(),
        Column('x', Integer, info={"special":True}, primary_key=True),
        Column('y', String(50)),
        Column('z', String(20), info={"special":True})
    )

metadata.create_all(conn)
```

Above, the directives we've added to the `Column.info` collection will be detected by our custom compilation scheme:

```
CREATE TABLE mytable (
        x SPECIAL DIRECTIVE INTEGER NOT NULL,
```

```
        y VARCHAR(50),
        z SPECIAL DIRECTIVE VARCHAR(20),
    PRIMARY KEY (x)
)
```

The `CreateColumn` construct can also be used to skip certain columns when producing a `CREATE TABLE`. This is accomplished by creating a compilation rule that conditionally returns `None`. This is essentially how to produce the same effect as using the `system=True` argument on `Column`, which marks a column as an implicitly-present "system" column.

For example, suppose we wish to produce a `Table` which skips rendering of the PostgreSQL `xmin` column against the PostgreSQL backend, but on other backends does render it, in anticipation of a triggered rule. A conditional compilation rule could skip this name only on PostgreSQL:

```
from sqlalchemy.schema import CreateColumn

@compiles(CreateColumn, "postgresql")
def skip_xmin(element, compiler, **kw):
    if element.element.name == 'xmin':
        return None
    else:
        return compiler.visit_create_column(element, **kw)


my_table = Table('mytable', metadata,
            Column('id', Integer, primary_key=True),
            Column('xmin', Integer)
        )
```

Above, a `CreateTable` construct will generate a `CREATE TABLE` which only includes the `id` column in the string; the `xmin` column will be omitted, but only against the PostgreSQL backend.

New in version 0.8.3: The `CreateColumn` construct supports skipping of columns by returning `None` from a custom compilation rule.

New in version 0.8: The `CreateColumn` construct was added to support custom column creation styles.

class sqlalchemy.schema.**CreateSequence**(*element*, *on=None*, *bind=None*)
Represent a CREATE SEQUENCE statement.

class sqlalchemy.schema.**DropSequence**(*element*, *on=None*, *bind=None*)
Represent a DROP SEQUENCE statement.

class sqlalchemy.schema.**CreateIndex**(*element*, *on=None*, *bind=None*)
Represent a CREATE INDEX statement.

class sqlalchemy.schema.**DropIndex**(*element*, *on=None*, *bind=None*)
Represent a DROP INDEX statement.

class sqlalchemy.schema.**AddConstraint**(*element*, **args*, ***kw*)
Represent an ALTER TABLE ADD CONSTRAINT statement.

class sqlalchemy.schema.**DropConstraint**(*element*, *cascade=False*, ***kw*)
Represent an ALTER TABLE DROP CONSTRAINT statement.

class sqlalchemy.schema.**CreateSchema**(*name*, *quote=None*, ***kw*)
Represent a CREATE SCHEMA statement.

New in version 0.7.4.

The argument here is the string name of the schema.

class sqlalchemy.schema.**DropSchema**(*name*, *quote=None*, *cascade=False*, ***kw*)
Represent a DROP SCHEMA statement.

The argument here is the string name of the schema.

New in version 0.7.4.

3.4 Column and Data Types

3.4.1 Column and Data Types

SQLAlchemy provides abstractions for most common database data types, and a mechanism for specifying your own custom data types.

The methods and attributes of type objects are rarely used directly. Type objects are supplied to `Table` definitions and can be supplied as type hints to *functions* for occasions where the database driver returns an incorrect type.

```
>>> users = Table('users', metadata,
...               Column('id', Integer, primary_key=True)
...               Column('login', String(32))
...              )
```

SQLAlchemy will use the `Integer` and `String(32)` type information when issuing a `CREATE TABLE` statement and will use it again when reading back rows `SELECTed` from the database. Functions that accept a type (such as `Column()`) will typically accept a type class or instance; `Integer` is equivalent to `Integer()` with no construction arguments in this case.

Generic Types

Generic types specify a column that can read, write and store a particular type of Python data. SQLAlchemy will choose the best database column type available on the target database when issuing a `CREATE TABLE` statement. For complete control over which column type is emitted in `CREATE TABLE`, such as `VARCHAR` see types_sqlstandard and the other sections of this chapter.

class sqlalchemy.types.**BigInteger**
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*, *_create_events=True*)
A bool datatype.

`Boolean` typically uses BOOLEAN or SMALLINT on the DDL side, and on the Python side deals in `True` or `False`.

The `Boolean` datatype currently has two levels of assertion that the values persisted are simple true/false values. For all backends, only the Python values `None`, `True`, `False`, `1` or `0` are accepted as parameter values. For those backends that don't support a "native boolean" datatype, a CHECK constraint is also created on the target column. Production of the CHECK constraint can be disabled by passing the `Boolean.create_constraint` flag set to `False`.

Changed in version 1.2: the `Boolean` datatype now asserts that incoming Python values are already in pure boolean form.

class sqlalchemy.types.**Date**
A type for `datetime.date()` objects.

class sqlalchemy.types.**DateTime**(*timezone=False*)
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.

For the time representation within the datetime type, some backends include additional options, such as timezone support and fractional seconds support. For fractional seconds, use the dialect-specific datatype, such as *mysql.TIME*. For timezone support, use at least the TIMESTAMP datatype, if not the dialect-specific datatype object.

class sqlalchemy.types.Enum(*enums, **kw)

Generic Enum Type.

The Enum type provides a set of possible string values which the column is constrained towards.

The Enum type will make use of the backend's native "ENUM" type if one is available; otherwise, it uses a VARCHAR datatype and produces a CHECK constraint. Use of the backend-native enum type can be disabled using the Enum.native_enum flag, and the production of the CHECK constraint is configurable using the Enum.create_constraint flag.

The Enum type also provides in-Python validation of string values during both read and write operations. When reading a value from the database in a result set, the string value is always checked against the list of possible values and a LookupError is raised if no match is found. When passing a value to the database as a plain string within a SQL statement, if the Enum.validate_strings parameter is set to True, a LookupError is raised for any string value that's not located in the given list of possible values; note that this impacts usage of LIKE expressions with enumerated values (an unusual use case).

Changed in version 1.1: the Enum type now provides in-Python validation of input values as well as on data being returned by the database.

The source of enumerated values may be a list of string values, or alternatively a PEP-435-compliant enumerated class. For the purposes of the Enum datatype, this class need only provide a __members__ method.

When using an enumerated class, the enumerated objects are used both for input and output, rather than strings as is the case with a plain-string enumerated type:

```
import enum
class MyEnum(enum.Enum):
    one = 1
    two = 2
    three = 3


t = Table(
    'data', MetaData(),
    Column('value', Enum(MyEnum))
)

connection.execute(t.insert(), {"value": MyEnum.two})
assert connection.scalar(t.select()) is MyEnum.two
```

Above, the string names of each element, e.g. "one", "two", "three", are persisted to the database; the values of the Python Enum, here indicated as integers, are **not** used; the value of each enum can therefore be any kind of Python object whether or not it is persistable.

In order to persist the values and not the names, the Enum.values_callable parameter may be used. The value of this parameter is a user-supplied callable, which is intended to be used with a PEP-435-compliant enumerated class and returns a list of string values to be persisted. For a simple enumeration that uses string values, a callable such as `lambda x: [e.value for e in x]` is sufficient.

New in version 1.1: - support for PEP-435-style enumerated classes.

See also:

postgresql.ENUM - PostgreSQL-specific type, which has additional functionality.

mysql.ENUM - MySQL-specific type

`__init__`(**enums*, ***kw*)
Construct an enum.

Keyword arguments which don't apply to a specific backend are ignored by that backend.

Parameters

- **`*enums`** – either exactly one PEP-435 compliant enumerated type or one or more string or unicode enumeration labels. If unicode labels are present, the *convert_unicode* flag is auto-enabled.

  New in version 1.1: a PEP-435 style enumerated class may be passed.

- **`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.

- **`create_constraint`** – defaults to True. When creating a non-native enumerated type, also build a CHECK constraint on the database against the valid values.

  New in version 1.1: - added `Enum.create_constraint` which provides the option to disable the production of the CHECK constraint for a non-native enumerated type.

- **`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 its 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. If a PEP-435 enumerated class was used, its name (converted to lower case) is used by default.

- **`native_enum`** – Use the database's native ENUM type when available. Defaults to True. When False, uses VARCHAR + check constraint for all backends.

- **`schema`** – Schema name 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.

  Note: The `schema` of the `Enum` type does not by default make use of the `schema` established on the owning `Table`. If this behavior is desired, set the `inherit_schema` flag to `True`.

- **`quote`** – Set explicit quoting preferences for the type's name.

- **`inherit_schema`** – When `True`, the "schema" from the owning `Table` will be copied to the "schema" attribute of this `Enum`, replacing whatever value was passed for the `schema` attribute. This also takes effect when using the `Table.tometadata()` operation.

- **validate_strings** – when True, string values that are being passed to the database in a SQL statement will be checked for validity against the list of enumerated values. Unrecognized values will result in a LookupError being raised.

  New in version 1.1.0b2.

- **values_callable** – A callable which will be passed the PEP-435 compliant enumerated type, which should then return a list of string values to be persisted. This allows for alternate usages such as using the string value of an enum to be persisted to the database instead of its name.

  New in version 1.2.3.

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.Float(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
Type representing floating point types, such as FLOAT or REAL.

This type returns Python float objects by default, unless the Float.asdecimal flag is set to True, in which case they are coerced to decimal.Decimal objects.

Note: The Float type is designed to receive data from a database type that is explicitly known to be a floating point type (e.g. FLOAT, REAL, others) and not a decimal type (e.g. DECIMAL, NUMERIC, others). If the database column on the server is in fact a Numeric type, such as DECIMAL or NUMERIC, use the Numeric type or a subclass, otherwise numeric coercion between float/Decimal may or may not function as expected.

---

class sqlalchemy.types.Integer
A type for int integers.

class sqlalchemy.types.Interval(*native=True*, *second_precision=None*, *day_precision=None*)
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).

impl
alias of DateTime

class sqlalchemy.types.LargeBinary(*length=None*)
A type for large binary byte data.

The LargeBinary type corresponds to a large and/or unlengthed binary type for the target platform, such as BLOB on MySQL and BYTEA for PostgreSQL. It also handles the necessary conversions for the DBAPI.

class sqlalchemy.types.MatchType(*create_constraint=True*, *name=None*, *_create_events=True*)
Refers to the return type of the MATCH operator.

As the ColumnOperators.match() is probably the most open-ended operator in generic SQLAlchemy Core, we can't assume the return type at SQL evaluation time, as MySQL returns a floating point, not a boolean, and other backends might do something different. So this type acts as

a placeholder, currently subclassing `Boolean`. The type allows dialects to inject result-processing functionality if needed, and on MySQL will return floating-point values.

New in version 1.0.0.

class sqlalchemy.types.**Numeric**(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)

A type for fixed precision numbers, such as `NUMERIC` or `DECIMAL`.

This type returns Python `decimal.Decimal` objects by default, unless the `Numeric.asdecimal` flag is set to False, in which case they are coerced to Python `float` objects.

Note: The `Numeric` type is designed to receive data from a database type that is explicitly known to be a decimal type (e.g. `DECIMAL`, `NUMERIC`, others) and not a floating point type (e.g. `FLOAT`, `REAL`, others). If the database column on the server is in fact a floating-point type type, such as `FLOAT` or `REAL`, use the `Float` type or a subclass, otherwise numeric coercion between `float`/`Decimal` may or may not function as expected.

Note: The Python `decimal.Decimal` class is generally slow performing; cPython 3.3 has now switched to use the cdecimal library natively. For older Python versions, the `cdecimal` library can be patched into any application where it will replace the `decimal` library fully, however this needs to be applied globally and before any other modules have been imported, as follows:

```
import sys
import cdecimal
sys.modules["decimal"] = cdecimal
```

Note that the `cdecimal` and `decimal` libraries are **not compatible with each other**, so patching `cdecimal` at the global level is the only way it can be used effectively with various DBAPIs that hardcode to import the `decimal` library.

---

class sqlalchemy.types.**PickleType**(*protocol=4*, *pickler=None*, *comparator=None*)

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.

To allow ORM change events to propagate for elements associated with `PickleType`, see mutable_toplevel.

impl

alias of `LargeBinary`

class sqlalchemy.types.**SchemaType**(*name=None*, *schema=None*, *metadata=None*, *inherit_schema=False*, *quote=None*, *_create_events=True*)

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.

`SchemaType` classes can also be targets for the `DDLEvents.before_parent_attach()` and `DDLEvents.after_parent_attach()` events, where the events fire off surrounding the association of the type object with a parent `Column`.

See also:

`Enum`

`Boolean`

adapt(*impltype*, ***kw*)

bind

copy(***kw*)

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
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*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
The base for all string and character types.

In SQL, corresponds to VARCHAR. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.)

The *length* field is usually required when the *String* type is used within a CREATE TABLE statement, as VARCHAR requires a length on most databases.

class sqlalchemy.types.Text(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
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.) In general, TEXT objects do not have a length; while some databases will accept a length argument here, it will be rejected by others.

class sqlalchemy.types.Time(*timezone=False*)
A type for `datetime.time()` objects.

class sqlalchemy.types.Unicode(*length=None*, ***kwargs*)
A variable length Unicode string type.

The `Unicode` type is a `String` subclass that assumes input and output as Python `unicode` data, and in that regard is equivalent to the usage of the `convert_unicode` flag with the `String` type. However, unlike plain `String`, it also implies an underlying column type that is explicitly supporting of non-ASCII data, such as `NVARCHAR` on Oracle and SQL Server. This can impact the output of `CREATE TABLE` statements and `CAST` functions at the dialect level, and can also affect the handling of bound parameters in some specific DBAPI scenarios.

The encoding used by the `Unicode` type is usually determined by the DBAPI itself; most modern DBAPIs feature support for Python `unicode` objects as bound values and result set values, and the encoding should be configured as detailed in the notes for the target DBAPI in the dialect_toplevel section.

For those DBAPIs which do not support, or are not configured to accommodate Python `unicode` objects directly, SQLAlchemy does the encoding and decoding outside of the DBAPI. The encoding in this scenario is determined by the `encoding` flag passed to `create_engine()`.

When using the `Unicode` type, it is only appropriate to pass Python `unicode` objects, and not plain `str`. If a plain `str` is passed under Python 2, a warning is emitted. If you notice your application emitting these warnings but you're not sure of the source of them, the Python `warnings` filter, documented at http://docs.python.org/library/warnings.html, can be used to turn these warnings into exceptions which will illustrate a stack trace:

```
import warnings
warnings.simplefilter('error')
```

For an application that wishes to pass plain bytestrings and Python `unicode` objects to the `Unicode` type equally, the bytestrings must first be decoded into unicode. The recipe at coerce_to_unicode illustrates how this is done.

See also:

`UnicodeText` - unlengthed textual counterpart to `Unicode`.

class sqlalchemy.types.**UnicodeText**(*length=None*, ***kwargs*)

An unbounded-length Unicode string type.

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`, `NTEXT`.

SQL Standard and Multiple Vendor Types

This category of types refers to types that are either part of the SQL standard, or are potentially found within a subset of database backends. Unlike the "generic" types, the SQL standard/multi-vendor types have **no** guarantee of working on all backends, and will only work on those backends that explicitly support them by name. That is, the type will always emit its exact name in DDL with `CREATE TABLE` is issued.

class sqlalchemy.types.**ARRAY**(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)

Represent a SQL Array type.

Note: This type serves as the basis for all ARRAY operations. However, currently **only the PostgreSQL backend has support for SQL arrays in SQLAlchemy**. It is recommended to use the *postgresql.ARRAY* type directly when using ARRAY types with PostgreSQL, as it provides additional operators specific to that backend.

`types.ARRAY` is part of the Core in support of various SQL standard functions such as `array_agg` which explicitly involve arrays; however, with the exception of the PostgreSQL backend and possibly some third-party dialects, no other SQLAlchemy built-in dialect has support for this type.

An `types.ARRAY` type is constructed given the "type" of element:

```
mytable = Table("mytable", metadata,
        Column("data", ARRAY(Integer))
    )
```

The above type represents an N-dimensional array, meaning a supporting backend such as PostgreSQL will interpret values with any number of dimensions automatically. To produce an INSERT construct that passes in a 1-dimensional array of integers:

```
connection.execute(
        mytable.insert(),
        data=[1,2,3]
)
```

The `types.ARRAY` type can be constructed given a fixed number of dimensions:

```
mytable = Table("mytable", metadata,
        Column("data", ARRAY(Integer, dimensions=2))
    )
```

Sending a number of dimensions is optional, but recommended if the datatype is to represent arrays of more than one dimension. This number is used:

- When emitting the type declaration itself to the database, e.g. `INTEGER[][]`

- When translating Python values to database values, and vice versa, e.g. an ARRAY of `Unicode` objects uses this number to efficiently access the string values inside of array structures without resorting to per-row type inspection
- When used with the Python `getitem` accessor, the number of dimensions serves to define the kind of type that the `[]` operator should return, e.g. for an ARRAY of INTEGER with two dimensions:

```
>>> expr = table.c.column[5]  # returns ARRAY(Integer, dimensions=1)
>>> expr = expr[6]  # returns Integer
```

For 1-dimensional arrays, an `types.ARRAY` instance with no dimension parameter will generally assume single-dimensional behaviors.

SQL expressions of type `types.ARRAY` have support for "index" and "slice" behavior. The Python `[]` operator works normally here, given integer indexes or slices. Arrays default to 1-based indexing. The operator produces binary expression constructs which will produce the appropriate SQL, both for SELECT statements:

```
select([mytable.c.data[5], mytable.c.data[2:7]])
```

as well as UPDATE statements when the `Update.values()` method is used:

```
mytable.update().values({
    mytable.c.data[5]: 7,
    mytable.c.data[2:7]: [1, 2, 3]
})
```

The `types.ARRAY` type also provides for the operators `types.ARRAY.Comparator.any()` and `types.ARRAY.Comparator.all()`. The PostgreSQL-specific version of `types.ARRAY` also provides additional operators.

New in version 1.1.0.

See also:

postgresql.ARRAY

`class Comparator(`*expr*`)`

Define comparison operations for `types.ARRAY`.

More operators are available on the dialect-specific form of this type. See *postgresql.ARRAY.Comparator*.

`all(`*elements*, *other*, *operator=None*`)`

Return `other operator ALL (array)` clause.

Argument places are switched, because ALL requires array expression to be on the right hand-side.

E.g.:

```
from sqlalchemy.sql import operators

conn.execute(
    select([table.c.data]).where(
            table.c.data.all(7, operator=operators.lt)
        )
)
```

Parameters

- **other** – expression to be compared
- **operator** – an operator object from the `sqlalchemy.sql.operators` package, defaults to `operators.eq()`.

See also:

`sql.expression.all_()`

`types.ARRAY.Comparator.any()`

any(*elements*, *other*, *operator=None*)
Return `other operator ANY (array)` clause.

Argument places are switched, because ANY requires array expression to be on the right hand-side.

E.g.:

```
from sqlalchemy.sql import operators

conn.execute(
    select([table.c.data]).where(
            table.c.data.any(7, operator=operators.lt)
        )
)
```

Parameters

- **other** – expression to be compared
- **operator** – an operator object from the `sqlalchemy.sql.operators` package, defaults to `operators.eq()`.

See also:

`sql.expression.any_()`

`types.ARRAY.Comparator.all()`

comparator_factory
alias of `Comparator`

zero_indexes = False
if True, Python zero-based indexes should be interpreted as one-based on the SQL expression side.

class sqlalchemy.types.BIGINT
The SQL BIGINT type.

class sqlalchemy.types.BINARY(*length=None*)
The SQL BINARY type.

class sqlalchemy.types.BLOB(*length=None*)
The SQL BLOB type.

class sqlalchemy.types.BOOLEAN(*create_constraint=True*, *name=None*, *_create_events=True*)
The SQL BOOLEAN type.

class sqlalchemy.types.CHAR(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
The SQL CHAR type.

class sqlalchemy.types.CLOB(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
The CLOB type.

This type is found in Oracle and Informix.

class sqlalchemy.types.DATE
The SQL DATE type.

class sqlalchemy.types.DATETIME(*timezone=False*)
The SQL DATETIME type.

class sqlalchemy.types.DECIMAL(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)
The SQL DECIMAL type.

class sqlalchemy.types.FLOAT(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
The SQL FLOAT type.

sqlalchemy.types.INT
alias of INTEGER

class sqlalchemy.types.JSON(*none_as_null=False*)
Represent a SQL JSON type.

Note: types.JSON is provided as a facade for vendor-specific JSON types. Since it supports JSON SQL operations, it only works on backends that have an actual JSON type, currently PostgreSQL as well as certain versions of MySQL.

---

types.JSON is part of the Core in support of the growing popularity of native JSON datatypes.

The types.JSON type stores arbitrary JSON format data, e.g.:

```
data_table = Table('data_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', JSON)
)

with engine.connect() as conn:
    conn.execute(
        data_table.insert(),
        data = {"key1": "value1", "key2": "value2"}
    )
```

The base types.JSON provides these two operations:

- Keyed index operations:

```
data_table.c.data['some key']
```

- Integer index operations:

```
data_table.c.data[3]
```

- Path index operations:

```
data_table.c.data[('key_1', 'key_2', 5, ..., 'key_n')]
```

Additional operations are available from the dialect-specific versions of types.JSON, such as *postgresql.JSON* and *postgresql.JSONB*, each of which offer more operators than just the basic type.

Index operations return an expression object whose type defaults to JSON by default, so that further JSON-oriented instructions may be called upon the result type. Note that there are backend-specific idiosyncracies here, including that the Postgresql database does not generally compare a "json" to a "json" structure without type casts. These idiosyncracies can be accommodated in a backend-neutral way by by making explicit use of the cast() and type_coerce() constructs. Comparison of specific index elements of a JSON object to other objects work best if the **left hand side is CAST to a string** and the **right hand side is rendered as a json string**; a future SQLAlchemy feature such as a generic "astext" modifier may simplify this at some point:

- **Compare an element of a JSON structure to a string**:

```
from sqlalchemy import cast, type_coerce
from sqlalchemy import String, JSON

cast(
    data_table.c.data['some_key'], String
) == '"some_value"'

cast(
    data_table.c.data['some_key'], String
) == type_coerce("some_value", JSON)
```

- **Compare an element of a JSON structure to an integer**:

```
from sqlalchemy import cast, type_coerce
from sqlalchemy import String, JSON

cast(data_table.c.data['some_key'], String) == '55'

cast(
    data_table.c.data['some_key'], String
) == type_coerce(55, JSON)
```

- **Compare an element of a JSON structure to some other JSON structure** - note that Python dictionaries are typically not ordered so care should be taken here to assert that the JSON structures are identical:

```
from sqlalchemy import cast, type_coerce
from sqlalchemy import String, JSON
import json

cast(
    data_table.c.data['some_key'], String
) == json.dumps({"foo": "bar"})

cast(
    data_table.c.data['some_key'], String
) == type_coerce({"foo": "bar"}, JSON)
```

The `JSON` type, when used with the SQLAlchemy ORM, does not detect in-place mutations to the structure. In order to detect these, the `sqlalchemy.ext.mutable` extension must be used. This extension will allow "in-place" changes to the datastructure to produce events which will be detected by the unit of work. See the example at *HSTORE* for a simple example involving a dictionary.

When working with NULL values, the `JSON` type recommends the use of two specific constants in order to differentiate between a column that evaluates to SQL NULL, e.g. no value, vs. the JSON-encoded string of `"null"`. To insert or select against a value that is SQL NULL, use the constant `null()`:

```
from sqlalchemy import null
conn.execute(table.insert(), json_value=null())
```

To insert or select against a value that is JSON `"null"`, use the constant `JSON.NULL`:

```
conn.execute(table.insert(), json_value=JSON.NULL)
```

The `JSON` type supports a flag `JSON.none_as_null` which when set to True will result in the Python constant `None` evaluating to the value of SQL NULL, and when set to False results in the Python constant `None` evaluating to the value of JSON `"null"`. The Python value `None` may be used in conjunction with either `JSON.NULL` and `null()` in order to indicate NULL values, but care must be taken as to the value of the `JSON.none_as_null` in these cases.

See also:

postgresql.JSON

postgresql.JSONB

mysql.JSON

New in version 1.1.

class Comparator(*expr*)
Define comparison operations for `types.JSON`.

class JSONElementType
common function for index / path elements in a JSON expression.

class JSONIndexType
Placeholder for the datatype of a JSON index value.

This allows execution-time processing of JSON index values for special syntaxes.

class JSONPathType
Placeholder type for JSON path operations.

This allows execution-time processing of a path-based index value into a specific SQL syntax.

NULL = symbol('JSON_NULL')
Describe the json value of NULL.

This value is used to force the JSON value of `"null"` to be used as the value. A value of Python `None` will be recognized either as SQL NULL or JSON `"null"`, based on the setting of the `JSON.none_as_null` flag; the `JSON.NULL` constant can be used to always resolve to JSON `"null"` regardless of this setting. This is in contrast to the `sql.null()` construct, which always resolves to SQL NULL. E.g.:

```
from sqlalchemy import null
from sqlalchemy.dialects.postgresql import JSON

obj1 = MyObject(json_value=null())  # will *always* insert SQL NULL
obj2 = MyObject(json_value=JSON.NULL)  # will *always* insert JSON string "null"

session.add_all([obj1, obj2])
session.commit()
```

In order to set JSON NULL as a default value for a column, the most transparent method is to use `text()`:

```
Table(
    'my_table', metadata,
    Column('json_data', JSON, default=text("'null'"))
)
```

While it is possible to use `JSON.NULL` in this context, the `JSON.NULL` value will be returned as the value of the column, which in the context of the ORM or other repurposing of the default value, may not be desirable. Using a SQL expression means the value will be re-fetched from the database within the context of retrieving generated defaults.

comparator_factory
alias of `Comparator`

class sqlalchemy.types.INTEGER
The SQL INT or INTEGER type.

class sqlalchemy.types.NCHAR(*length=None*, ***kwargs*)
The SQL NCHAR type.

class sqlalchemy.types.NVARCHAR(*length=None*, ***kwargs*)
The SQL NVARCHAR type.

class sqlalchemy.types.**NUMERIC**(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)
The SQL NUMERIC type.

class sqlalchemy.types.**REAL**(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
The SQL REAL type.

class sqlalchemy.types.**SMALLINT**
The SQL SMALLINT type.

class sqlalchemy.types.**TEXT**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
The SQL TEXT type.

class sqlalchemy.types.**TIME**(*timezone=False*)
The SQL TIME type.

class sqlalchemy.types.**TIMESTAMP**(*timezone=False*)
The SQL TIMESTAMP type.

`TIMESTAMP` datatypes have support for timezone storage on some backends, such as PostgreSQL and Oracle. Use the `timezone` argument in order to enable "TIMESTAMP WITH TIMEZONE" for these backends.

class sqlalchemy.types.**VARBINARY**(*length=None*)
The SQL VARBINARY type.

class sqlalchemy.types.**VARCHAR**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
The SQL VARCHAR type.

Vendor-Specific Types

Database-specific types are also available for import from each database's dialect module. See the dialect_toplevel reference for the database you're interested in.

For example, MySQL has a `BIGINT` type and PostgreSQL has an `INET` type. To use these, import them from the module explicitly:

```
from sqlalchemy.dialects import mysql

table = Table('foo', metadata,
    Column('id', mysql.BIGINT),
    Column('enumerates', mysql.ENUM('a', 'b', 'c'))
)
```

Or some PostgreSQL types:

```
from sqlalchemy.dialects import postgresql

table = Table('foo', metadata,
    Column('ipaddress', postgresql.INET),
    Column('elements', postgresql.ARRAY(String))
)
```

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'))
)
```

3.4.2 Custom Types

A variety of methods exist to redefine the behavior of existing types as well as to provide new ones.

Overriding Type Compilation

A frequent need is to force the "string" version of a type, that is the one rendered in a CREATE TABLE statement or other SQL function like CAST, to be changed. For example, an application may want to force the rendering of `BINARY` for all platforms except for one, in which is wants `BLOB` to be rendered. Usage of an existing generic type, in this case `LargeBinary`, is preferred for most use cases. But to control types more accurately, a compilation directive that is per-dialect can be associated with any type:

```
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.types import BINARY

@compiles(BINARY, "sqlite")
def compile_binary_sqlite(type_, compiler, **kw):
    return "BLOB"
```

The above code allows the usage of `types.BINARY`, which will produce the string `BINARY` against all backends except SQLite, in which case it will produce `BLOB`.

See the section type_compilation_extension, a subsection of sqlalchemy.ext.compiler_toplevel, for additional examples.

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 marshaling of data to and from the database is required.

Note: The bind- and result-processing of `TypeDecorator` is *in addition* to the processing already performed by the hosted type, which is customized by SQLAlchemy on a per-DBAPI basis to perform processing specific to that DBAPI. To change the DBAPI-level processing for an existing type, see the section replacing_processors.

---

class sqlalchemy.types.**TypeDecorator**(**args*, ***kwargs*)

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, **kw):
        return MyType(self.impl.length)
```

The class-level "impl" attribute 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
```

Warning: Note that the **behavior of coerce_compared_value is not inherited by default from that of the base type**. If the `TypeDecorator` is augmenting a type that requires special logic for certain types of operators, this method **must** be overridden. A key example is when decorating the *postgresql.JSON* and *postgresql.JSONB* types; the default rules of `TypeEngine.coerce_compared_value()` should be used in order to deal with operators like index operations:

```
class MyJsonType(TypeDecorator):
    impl = postgresql.JSON

    def coerce_compared_value(self, op, value):
        return self.impl.coerce_compared_value(op, value)
```

Without the above step, index operations such as `mycol['foo']` will cause the index value `'foo'` to be JSON encoded.

`adapt`(*cls*, ***kw*)
Produce an "adapted" form of this type, given an "impl" class to work with.

This method is used internally to associate generic types with "implementation" types that are specific to a particular dialect.

`bind_expression`(*bindvalue*)
"Given a bind value (i.e. a `BindParameter` instance), return a SQL expression in its place.

This is typically a SQL function that wraps the existing bound parameter within the statement. It is used for special data types that require literals being wrapped in some special database function in order to coerce an application-level value into a database-specific format. It is the SQL analogue of the `TypeEngine.bind_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

Note that this method, when implemented, should always return the exact same structure, without any conditional logic, as it may be used in an executemany() call against an arbitrary number of bound parameter sets.

See also:

types_sql_value_processing

`bind_processor`(*dialect*)
Provide a bound value processing function for the given `Dialect`.

This is the method that fulfills the `TypeEngine` contract for bound value conversion. `TypeDecorator` will wrap a user-defined implementation of `process_bind_param()` here.

User-defined code can override this method directly, though its likely best to use `process_bind_param()` so that the processing provided by `self.impl` is maintained.

Parameters `dialect` – Dialect instance in use.

This method is the reverse counterpart to the `result_processor()` method of this class.

`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.

coerce_to_is_types = (<class 'NoneType'>,)

Specify those Python types which should be coerced at the expression level to "IS <constant>" when compared using `==` (and same for `IS NOT` in conjunction with `!=`.

For most SQLAlchemy types, this includes `NoneType`, as well as `bool`.

`TypeDecorator` modifies this list to only include `NoneType`, as typedecorator implementations that deal with boolean types are common.

Custom `TypeDecorator` classes can override this attribute to return an empty tuple, in which case no values will be coerced to constants.

New in version 0.8.2: Added `TypeDecorator.coerce_to_is_types` to allow for easier control of `__eq__()` `__ne__()` operations.

column_expression(*colexpr*)

Given a SELECT column expression, return a wrapping SQL expression.

This is typically a SQL function that wraps a column expression as rendered in the columns clause of a SELECT statement. It is used for special data types that require columns to be wrapped in some special database function in order to coerce the value before being sent back to the application. It is the SQL analogue of the `TypeEngine.result_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

See also:

types_sql_value_processing

compare_against_backend(*dialect*, *conn_type*)

Compare this type against the given backend type.

This function is currently not implemented for SQLAlchemy types, and for all built in types will return `None`. However, it can be implemented by a user-defined type where it can be consumed by schema comparison tools such as Alembic autogenerate.

A future release of SQLAlchemy will potentially impement this method for builtin types as well.

The function should return True if this type is equivalent to the given type; the type is typically reflected from the database so should be database specific. The dialect in use is also passed. It can also return False to assert that the type is not equivalent.

Parameters

- **dialect** – a `Dialect` that is involved in the comparison.
- **conn_type** – the type object reflected from the backend.

New in version 1.0.3.

compare_values(*x*, *y*)

Given two values, compare them for equality.

By default this calls upon `TypeEngine.compare_values()` of the underlying "impl", which in turn usually uses the Python equals operator `==`.

This function is used by the ORM to compare an original-loaded value with an intercepted "changed" value, to determine if a net change has occurred.

`compile`(*dialect=None*)
Produce a string-compiled form of this `TypeEngine`.

When called with no arguments, uses a "default" dialect to produce a string result.

Parameters `dialect` – a `Dialect` instance.

`copy`(***kw*)
Produce a copy of this `TypeDecorator` instance.

This is a shallow copy and is provided to fulfill part of the `TypeEngine` contract. It usually does not need to be overridden unless the user-defined `TypeDecorator` has local state that should be deep-copied.

`dialect_impl`(*dialect*)
Return a dialect-specific implementation for this `TypeEngine`.

`evaluates_none`()
Return a copy of this type which has the `should_evaluate_none` flag set to True.

E.g.:

```
Table(
    'some_table', metadata,
    Column(
        String(50).evaluates_none(),
        nullable=True,
        server_default='no value')
)
```

The ORM uses this flag to indicate that a positive value of `None` is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows for types which have special behavior associated with the Python None value to indicate that the value doesn't necessarily translate into SQL NULL; a prime example of this is a JSON type which may wish to persist the JSON value `'null'`.

In all cases, the actual NULL SQL value can be always be persisted in any column by using the `null` SQL construct in an INSERT statement or associated with an ORM-mapped attribute.

Note: The "evaulates none" flag does **not** apply to a value of `None` passed to `Column.default` or `Column.server_default`; in these cases, `None` still means "no default".

New in version 1.1.

See also:

session_forcing_null - in the ORM documentation

`postgresql.JSON.none_as_null` - PostgreSQL JSON interaction with this flag.

`TypeEngine.should_evaluate_none` - class-level flag

`get_dbapi_type`(*dbapi*)
Return the DBAPI type object represented by this `TypeDecorator`.

By default this calls upon `TypeEngine.get_dbapi_type()` of the underlying "impl".

`literal_processor`(*dialect*)
Provide a literal processing function for the given `Dialect`.

Subclasses here will typically override `TypeDecorator.process_literal_param()` instead of this method directly.

By default, this method makes use of `TypeDecorator.process_bind_param()` if that method is implemented, where `TypeDecorator.process_literal_param()` is not. The rationale here is that `TypeDecorator` typically deals with Python conversions of data that are above the layer of database presentation. With the value converted by `TypeDecorator.process_bind_param()`, the underlying type will then handle whether it needs to be presented to the DBAPI as a bound parameter or to the database as an inline SQL value.

New in version 0.9.0.

`load_dialect_impl`(*dialect*)
Return a `TypeEngine` object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the `TypeDecorator` implementation of `type_engine()` to help determine what type should ultimately be returned for a given `TypeDecorator`.

By default returns `self.impl`.

`process_bind_param`(*value*, *dialect*)
Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying `TypeEngine` object, and from there to the DBAPI `execute()` method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **`value`** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **`dialect`** – the `Dialect` in use.

`process_literal_param`(*value*, *dialect*)
Receive a literal parameter value to be rendered inline within a statement.

This method is used when the compiler renders a literal value without using binds, typically within DDL such as in the "server default" of a column or an expression within a CHECK constraint.

The returned string will be rendered into the output string.

New in version 0.9.0.

`process_result_value`(*value*, *dialect*)
Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying `TypeEngine` object, originally from the DBAPI cursor method `fetchone()` or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **`value`** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **`dialect`** – the `Dialect` in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

`python_type`
Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`result_processor`(*dialect*, *coltype*)
Provide a result value processing function for the given `Dialect`.

This is the method that fulfills the `TypeEngine` contract for result value conversion. `TypeDecorator` will wrap a user-defined implementation of `process_result_value()` here.

User-defined code can override this method directly, though its likely best to use `process_result_value()` so that the processing provided by `self.impl` is maintained.

Parameters

- **`dialect`** – Dialect instance in use.
- **`coltype`** – A SQLAlchemy data type

This method is the reverse counterpart to the `bind_processor()` method of this class.

`type_engine`(*dialect*)
Return a dialect-specific `TypeEngine` instance for this `TypeDecorator`.

In most cases this returns a dialect-adapted form of the `TypeEngine` type represented by `self.impl`. Makes usage of `dialect_impl()` but also traverses into wrapped `TypeDecorator` instances. Behavior can be customized here by overriding `load_dialect_impl()`.

`with_variant`(*type_*, *dialect_name*)
Produce a new type object that will utilize the given type when applied to the dialect of the given name.

e.g.:

```
from sqlalchemy.types import String
from sqlalchemy.dialects import mysql

s = String()

s = s.with_variant(mysql.VARCHAR(collation='foo'), 'mysql')
```

The construction of `TypeEngine.with_variant()` is always from the "fallback" type to that which is dialect specific. The returned type is an instance of `Variant`, which itself provides a `Variant.with_variant()` that can be called repeatedly.

Parameters

- **`type_`** – a `TypeEngine` that will be selected as a variant from the originating type, when a dialect of the given name is in use.
- **`dialect_name`** – base name of the dialect which uses this type. (i.e. `'postgresql'`, `'mysql'`, etc.)

New in version 0.7.2.

TypeDecorator Recipes

A few key `TypeDecorator` recipes follow.

Coercing Encoded Strings to Unicode

A common source of confusion regarding the `Unicode` type is that it is intended to deal *only* with Python `unicode` objects on the Python side, meaning values passed to it as bind parameters must be of the form `u'some string'` if using Python 2 and not 3. The encoding/decoding functions it performs are only to suit what the DBAPI in use requires, and are primarily a private implementation detail.

The use case of a type that can safely receive Python bytestrings, that is strings that contain non-ASCII characters and are not `u''` objects in Python 2, can be achieved using a `TypeDecorator` which coerces as needed:

```
from sqlalchemy.types import TypeDecorator, Unicode

class CoerceUTF8(TypeDecorator):
    """Safely coerce Python bytestrings to Unicode
    before passing off to the database."""

    impl = Unicode

    def process_bind_param(self, value, dialect):
        if isinstance(value, str):
            value = value.decode('utf-8')
        return value
```

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.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).int
            else:
                # hexstring
                return "%.32x" % value.int

    def process_result_value(self, value, dialect):
        if value is None:
            return value
        else:
            if not isinstance(value, uuid.UUID):
                value = uuid.UUID(value)
            return 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:

```
from sqlalchemy.types import TypeDecorator, VARCHAR
import json

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 = json.dumps(value)

        return value

    def process_result_value(self, value, dialect):
        if value is not None:
```

```
            value = json.loads(value)
        return value
```

Adding Mutability

The ORM by default will not detect "mutability" on such a type as above - meaning, in-place changes to values will not be detected and will not be flushed. Without further steps, you instead would need to replace the existing value with a new one on each parent object to detect changes:

```
obj.json_value["key"] = "value"  # will *not* be detected by the ORM

obj.json_value = {"key": "value"}  # *will* be detected by the ORM
```

The above limitation may be fine, as many applications may not require that the values are ever mutated once created. For those which do have this requirement, support for mutability is best applied using the `sqlalchemy.ext.mutable` extension. For a dictionary-oriented JSON structure, we can apply this as:

```
json_type = MutableDict.as_mutable(JSONEncodedDict)

class MyClass(Base):
    #  ...

    json_data = Column(json_type)
```

See also:

mutable_toplevel

Dealing with Comparison Operations

The default behavior of `TypeDecorator` is to coerce the "right hand side" of any expression into the same type. For a type like JSON, this means that any operator used must make sense in terms of JSON. For some cases, users may wish for the type to behave like JSON in some circumstances, and as plain text in others. One example is if one wanted to handle the LIKE operator for the JSON type. LIKE makes no sense against a JSON structure, but it does make sense against the underlying textual representation. To get at this with a type like `JSONEncodedDict`, we need to **coerce** the column to a textual form using `cast()` or `type_coerce()` before attempting to use this operator:

```
from sqlalchemy import type_coerce, String

stmt = select([my_table]).where(
    type_coerce(my_table.c.json_data, String).like('%foo%'))
```

`TypeDecorator` provides a built-in system for working up type translations like these based on operators. If we wanted to frequently use the LIKE operator with our JSON object interpreted as a string, we can build it into the type by overriding the `TypeDecorator.coerce_compared_value()` method:

```
from sqlalchemy.sql import operators
from sqlalchemy import String

class JSONEncodedDict(TypeDecorator):

    impl = VARCHAR

    def coerce_compared_value(self, op, value):
        if op in (operators.like_op, operators.notlike_op):
            return String()
        else:
```

```
            return self

    def process_bind_param(self, value, dialect):
        if value is not None:
            value = json.dumps(value)

        return value

    def process_result_value(self, value, dialect):
        if value is not None:
            value = json.loads(value)
        return value
```

Above is just one approach to handling an operator like "LIKE". Other applications may wish to raise `NotImplementedError` for operators that have no meaning with a JSON object such as "LIKE", rather than automatically coercing to text.

Replacing the Bind/Result Processing of Existing Types

Most augmentation of type behavior at the bind/result level is achieved using `TypeDecorator`. For the rare scenario where the specific processing applied by SQLAlchemy at the DBAPI level needs to be replaced, the SQLAlchemy type can be subclassed directly, and the `bind_processor()` or `result_processor()` methods can be overridden. Doing so requires that the `adapt()` method also be overridden. This method is the mechanism by which SQLAlchemy produces DBAPI-specific type behavior during statement execution. Overriding it allows a copy of the custom type to be used in lieu of a DBAPI-specific type. Below we subclass the `types.TIME` type to have custom result processing behavior. The `process()` function will receive `value` from the DBAPI cursor directly:

```
class MySpecialTime(TIME):
    def __init__(self, special_argument):
        super(MySpecialTime, self).__init__()
        self.special_argument = special_argument

    def result_processor(self, dialect, coltype):
        import datetime
        time = datetime.time
        def process(value):
            if value is not None:
                microseconds = value.microseconds
                seconds = value.seconds
                minutes = seconds / 60
                return time(
                          minutes / 60,
                          minutes % 60,
                          seconds - minutes * 60,
                          microseconds)
            else:
                return None
        return process

    def adapt(self, impltype):
        return MySpecialTime(self.special_argument)
```

Applying SQL-level Bind/Result Processing

As seen in the sections types_typedecorator and replacing_processors, SQLAlchemy allows Python functions to be invoked both when parameters are sent to a statement, as well as when result rows are loaded from the database, to apply transformations to the values as they are sent to or from the database. It is also possible to define SQL-level transformations as well. The rationale here is when only

the relational database contains a particular series of functions that are necessary to coerce incoming and outgoing data between an application and persistence format. Examples include using database-defined encryption/decryption functions, as well as stored procedures that handle geographic data. The PostGIS extension to PostgreSQL includes an extensive array of SQL functions that are necessary for coercing data into particular formats.

Any `TypeEngine`, `UserDefinedType` or `TypeDecorator` subclass can include implementations of `TypeEngine.bind_expression()` and/or `TypeEngine.column_expression()`, which when defined to return a non-`None` value should return a `ColumnElement` expression to be injected into the SQL statement, either surrounding bound parameters or a column expression. For example, to build a `Geometry` type which will apply the PostGIS function `ST_GeomFromText` to all outgoing values and the function `ST_AsText` to all incoming data, we can create our own subclass of `UserDefinedType` which provides these methods in conjunction with `func`:

```
from sqlalchemy import func
from sqlalchemy.types import UserDefinedType

class Geometry(UserDefinedType):
    def get_col_spec(self):
        return "GEOMETRY"

    def bind_expression(self, bindvalue):
        return func.ST_GeomFromText(bindvalue, type_=self)

    def column_expression(self, col):
        return func.ST_AsText(col, type_=self)
```

We can apply the `Geometry` type into `Table` metadata and use it in a `select()` construct:

```
geometry = Table('geometry', metadata,
              Column('geom_id', Integer, primary_key=True),
              Column('geom_data', Geometry)
            )

print(select([geometry]).where(
  geometry.c.geom_data == 'LINESTRING(189412 252431,189631 259122)'))
```

The resulting SQL embeds both functions as appropriate. `ST_AsText` is applied to the columns clause so that the return value is run through the function before passing into a result set, and `ST_GeomFromText` is run on the bound parameter so that the passed-in value is converted:

```
SELECT geometry.geom_id, ST_AsText(geometry.geom_data) AS geom_data_1
FROM geometry
WHERE geometry.geom_data = ST_GeomFromText(:geom_data_2)
```

The `TypeEngine.column_expression()` method interacts with the mechanics of the compiler such that the SQL expression does not interfere with the labeling of the wrapped expression. Such as, if we rendered a `select()` against a `label()` of our expression, the string label is moved to the outside of the wrapped expression:

```
print(select([geometry.c.geom_data.label('my_data')]))
```

Output:

```
SELECT ST_AsText(geometry.geom_data) AS my_data
FROM geometry
```

For an example of subclassing a built in type directly, we subclass *postgresql.BYTEA* to provide a `PGPString`, which will make use of the PostgreSQL `pgcrypto` extension to encrypt/decrypt values transparently:

```
from sqlalchemy import create_engine, String, select, func, \
        MetaData, Table, Column, type_coerce

from sqlalchemy.dialects.postgresql import BYTEA

class PGPString(BYTEA):
    def __init__(self, passphrase, length=None):
        super(PGPString, self).__init__(length)
        self.passphrase = passphrase

    def bind_expression(self, bindvalue):
        # convert the bind's type from PGPString to
        # String, so that it's passed to psycopg2 as is without
        # a dbapi.Binary wrapper
        bindvalue = type_coerce(bindvalue, String)
        return func.pgp_sym_encrypt(bindvalue, self.passphrase)

    def column_expression(self, col):
        return func.pgp_sym_decrypt(col, self.passphrase)

metadata = MetaData()
message = Table('message', metadata,
                Column('username', String(50)),
                Column('message',
                    PGPString("this is my passphrase", length=1000)),
            )

engine = create_engine("postgresql://scott:tiger@localhost/test", echo=True)
with engine.begin() as conn:
    metadata.create_all(conn)

    conn.execute(message.insert(), username="some user",
                                message="this is my message")

    print(conn.scalar(
            select([message.c.message]).\
                where(message.c.username == "some user")
        ))
```

The `pgp_sym_encrypt` and `pgp_sym_decrypt` functions are applied to the INSERT and SELECT statements:

```
INSERT INTO message (username, message)
  VALUES (%(username)s, pgp_sym_encrypt(%(message)s, %(pgp_sym_encrypt_1)s))
  {'username': 'some user', 'message': 'this is my message',
    'pgp_sym_encrypt_1': 'this is my passphrase'}

SELECT pgp_sym_decrypt(message.message, %(pgp_sym_decrypt_1)s) AS message_1
  FROM message
  WHERE message.username = %(username_1)s
  {'pgp_sym_decrypt_1': 'this is my passphrase', 'username_1': 'some user'}
```

New in version 0.8: Added the `TypeEngine.bind_expression()` and `TypeEngine.column_expression()` methods.

See also:

examples_postgis

Redefining and Creating New Operators

SQLAlchemy Core defines a fixed set of expression operators available to all column expressions. Some of these operations have the effect of overloading Python's built in operators; examples of such operators include `ColumnOperators.__eq__()` (`table.c.somecolumn == 'foo'`), `ColumnOperators.__invert__()` (`~table.c.flag`), and `ColumnOperators.__add__()` (`table.c.x + table.c.y`). Other operators are exposed as explicit methods on column expressions, such as `ColumnOperators.in_()` (`table.c.value.in_(['x', 'y'])`) and `ColumnOperators.like()` (`table.c.value.like('%ed%')`).

The Core expression constructs in all cases consult the type of the expression in order to determine the behavior of existing operators, as well as to locate additional operators that aren't part of the built in set. The `TypeEngine` base class defines a root "comparison" implementation `TypeEngine.Comparator`, and many specific types provide their own sub-implementations of this class. User-defined `TypeEngine.Comparator` implementations can be built directly into a simple subclass of a particular type in order to override or define new operations. Below, we create a `Integer` subclass which overrides the `ColumnOperators.__add__()` operator:

```
from sqlalchemy import Integer

class MyInt(Integer):
    class comparator_factory(Integer.Comparator):
        def __add__(self, other):
            return self.op("goofy")(other)
```

The above configuration creates a new class `MyInt`, which establishes the `TypeEngine.comparator_factory` attribute as referring to a new class, subclassing the `TypeEngine.Comparator` class associated with the `Integer` type.

Usage:

```
>>> sometable = Table("sometable", metadata, Column("data", MyInt))
>>> print(sometable.c.data + 5)
sometable.data goofy :data_1
```

The implementation for `ColumnOperators.__add__()` is consulted by an owning SQL expression, by instantiating the `TypeEngine.Comparator` with itself as the `expr` attribute. The mechanics of the expression system are such that operations continue recursively until an expression object produces a new SQL expression construct. Above, we could just as well have said `self.expr.op("goofy")(other)` instead of `self.op("goofy")(other)`.

When using `Operators.op()` for comparison operations that return a boolean result, the `Operators.op.is_comparison` flag should be set to `True`:

```
class MyInt(Integer):
    class comparator_factory(Integer.Comparator):
        def is_frobnozzled(self, other):
            return self.op("--is_frobnozzled->", is_comparison=True)(other)
```

New methods added to a `TypeEngine.Comparator` are exposed on an owning SQL expression using a `__getattr__` scheme, which exposes methods added to `TypeEngine.Comparator` onto the owning `ColumnElement`. For example, to add a `log()` function to integers:

```
from sqlalchemy import Integer, func

class MyInt(Integer):
    class comparator_factory(Integer.Comparator):
        def log(self, other):
            return func.log(self.expr, other)
```

Using the above type:

```
>>> print(sometable.c.data.log(5))
log(:log_1, :log_2)
```

Unary operations are also possible. For example, to add an implementation of the PostgreSQL factorial operator, we combine the `UnaryExpression` construct along with a `custom_op` to produce the factorial expression:

```
from sqlalchemy import Integer
from sqlalchemy.sql.expression import UnaryExpression
from sqlalchemy.sql import operators

class MyInteger(Integer):
    class comparator_factory(Integer.Comparator):
        def factorial(self):
            return UnaryExpression(self.expr,
                        modifier=operators.custom_op("!"),
                        type_=MyInteger)
```

Using the above type:

```
>>> from sqlalchemy.sql import column
>>> print(column('x', MyInteger).factorial())
x !
```

See also:

`Operators.op()`

`TypeEngine.comparator_factory`

Creating New Types

The `UserDefinedType` class is provided as a simple base class for defining entirely new database types. Use this to represent native database types not known by SQLAlchemy. If only Python translation behavior is needed, use `TypeDecorator` instead.

class sqlalchemy.types.UserDefinedType

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, **kw):
        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))
    )
```

The `get_col_spec()` method will in most cases receive a keyword argument `type_expression` which refers to the owning expression of the type as being compiled, such as a `Column` or `cast()` construct. This keyword is only sent if the method accepts keyword arguments (e.g. `**kw`) in its argument signature; introspection is used to check for this in order to support legacy forms of this function.

New in version 1.0.0: the owning expression is passed to the `get_col_spec()` method via the keyword argument `type_expression`, if it receives `**kw` in its signature.

`coerce_compared_value`(*op*, *value*)
Suggest a type for a 'coerced' Python value in an expression.

Default behavior for `UserDefinedType` is the same as that of `TypeDecorator`; by default it returns `self`, assuming the compared value should be coerced into the same type as this one. See `TypeDecorator.coerce_compared_value()` for more detail.

Changed in version 0.8: `UserDefinedType.coerce_compared_value()` now returns `self` by default, rather than falling onto the more fundamental behavior of `TypeEngine.coerce_compared_value()`.

3.4.3 Base Type API

`class sqlalchemy.types.TypeEngine`
The ultimate base class for all SQL datatypes.

Common subclasses of `TypeEngine` include `String`, `Integer`, and `Boolean`.

For an overview of the SQLAlchemy typing system, see types_toplevel.

See also:

types_toplevel

`class Comparator`(*expr*)
Base class for custom comparison operations defined at the type level. See `TypeEngine.comparator_factory`.

`adapt`(*cls*, ***kw*)
Produce an "adapted" form of this type, given an "impl" class to work with.

This method is used internally to associate generic types with "implementation" types that are specific to a particular dialect.

`bind_expression`(*bindvalue*)
"Given a bind value (i.e. a `BindParameter` instance), return a SQL expression in its place.

This is typically a SQL function that wraps the existing bound parameter within the statement. It is used for special data types that require literals being wrapped in some special database function in order to coerce an application-level value into a database-specific format. It is the SQL analogue of the `TypeEngine.bind_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

Note that this method, when implemented, should always return the exact same structure, without any conditional logic, as it may be used in an executemany() call against an arbitrary number of bound parameter sets.

See also:

types_sql_value_processing

`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`.

Parameters `dialect` – Dialect instance in use.

`coerce_compared_value`(*op*, *value*)
Suggest a type for a 'coerced' Python value in an expression.

Given an operator and value, gives the type a chance to return a type which the value should be coerced into.

The default behavior here is conservative; if the right-hand side is already coerced into a SQL type based on its Python type, it is usually left alone.

End-user functionality extension here should generally be via `TypeDecorator`, which provides more liberal behavior in that it defaults to coercing the other side of the expression into this type, thus applying special Python conversions above and beyond those needed by the DBAPI to both ides. It also provides the public method `TypeDecorator.coerce_compared_value()` which is intended for end-user customization of this behavior.

`column_expression`(*colexpr*)
Given a SELECT column expression, return a wrapping SQL expression.

This is typically a SQL function that wraps a column expression as rendered in the columns clause of a SELECT statement. It is used for special data types that require columns to be wrapped in some special database function in order to coerce the value before being sent back to the application. It is the SQL analogue of the `TypeEngine.result_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

See also:

types_sql_value_processing

`comparator_factory`
A `TypeEngine.Comparator` class which will apply to operations performed by owning `ColumnElement` objects.

The `comparator_factory` attribute is a hook consulted by the core expression system when column and SQL expression operations are performed. When a `TypeEngine.Comparator` class is associated with this attribute, it allows custom re-definition of all existing operators, as well as definition of new operators. Existing operators include those provided by Python operator overloading such as `operators.ColumnOperators.__add__()` and `operators.ColumnOperators.__eq__()`, those provided as standard attributes of `operators.ColumnOperators` such as `operators.ColumnOperators.like()` and `operators.ColumnOperators.in_()`.

Rudimentary usage of this hook is allowed through simple subclassing of existing types, or alternatively by using `TypeDecorator`. See the documentation section types_operators for examples.

New in version 0.8: The expression system was enhanced to support customization of operators on a per-type level.

alias of `Comparator`

`compare_against_backend`(*dialect*, *conn_type*)
Compare this type against the given backend type.

This function is currently not implemented for SQLAlchemy types, and for all built in types will return `None`. However, it can be implemented by a user-defined type where it can be consumed by schema comparison tools such as Alembic autogenerate.

A future release of SQLAlchemy will potentially impement this method for builtin types as well.

The function should return True if this type is equivalent to the given type; the type is typically reflected from the database so should be database specific. The dialect in use is also passed. It can also return False to assert that the type is not equivalent.

Parameters

- **dialect** – a `Dialect` that is involved in the comparison.
- **conn_type** – the type object reflected from the backend.

New in version 1.0.3.

compare_values(*x*, *y*)
Compare two values for equality.

compile(*dialect=None*)
Produce a string-compiled form of this `TypeEngine`.

When called with no arguments, uses a "default" dialect to produce a string result.

Parameters **dialect** – a `Dialect` instance.

dialect_impl(*dialect*)
Return a dialect-specific implementation for this `TypeEngine`.

evaluates_none()
Return a copy of this type which has the `should_evaluate_none` flag set to True.

E.g.:

```
Table(
    'some_table', metadata,
    Column(
        String(50).evaluates_none(),
        nullable=True,
        server_default='no value')
)
```

The ORM uses this flag to indicate that a positive value of `None` is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows for types which have special behavior associated with the Python None value to indicate that the value doesn't necessarily translate into SQL NULL; a prime example of this is a JSON type which may wish to persist the JSON value `'null'`.

In all cases, the actual NULL SQL value can be always be persisted in any column by using the `null` SQL construct in an INSERT statement or associated with an ORM-mapped attribute.

Note: The "evaulates none" flag does **not** apply to a value of `None` passed to `Column.default` or `Column.server_default`; in these cases, `None` still means "no default".

New in version 1.1.

See also:

session_forcing_null - in the ORM documentation

postgresql.JSON.none_as_null - PostgreSQL JSON interaction with this flag.

`TypeEngine.should_evaluate_none` - class-level flag

`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.

`hashable = True`
Flag, if False, means values from this type aren't hashable.

Used by the ORM when uniquing result lists.

`literal_processor`(*dialect*)
Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the "literal_binds" flag, typically used in DDL generation as well as in certain scenarios where backends don't accept bound parameters.

New in version 0.9.0.

`python_type`
Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`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`.

Parameters

- `dialect` – Dialect instance in use.
- `coltype` – DBAPI coltype argument received in cursor.description.

`should_evaluate_none = False`
If True, the Python constant `None` is considered to be handled explicitly by this type.

The ORM uses this flag to indicate that a positive value of `None` is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows types which have special behavior for Python None, such as a JSON type, to indicate that they'd like to handle the None value explicitly.

To set this flag on an existing type, use the `TypeEngine.evaluates_none()` method.

See also:

`TypeEngine.evaluates_none()`

New in version 1.1.

`with_variant`(*type_*, *dialect_name*)
Produce a new type object that will utilize the given type when applied to the dialect of the given name.

e.g.:

```
from sqlalchemy.types import String
from sqlalchemy.dialects import mysql

s = String()

s = s.with_variant(mysql.VARCHAR(collation='foo'), 'mysql')
```

The construction of `TypeEngine.with_variant()` is always from the "fallback" type to that which is dialect specific. The returned type is an instance of `Variant`, which itself provides a `Variant.with_variant()` that can be called repeatedly.

Parameters

- **type_** – a `TypeEngine` that will be selected as a variant from the originating type, when a dialect of the given name is in use.
- **dialect_name** – base name of the dialect which uses this type. (i.e. `'postgresql'`, `'mysql'`, etc.)

New in version 0.7.2.

`class sqlalchemy.types.Concatenable`
A mixin that marks a type as supporting 'concatenation', typically strings.

`class sqlalchemy.types.Indexable`
A mixin that marks a type as supporting indexing operations, such as array or JSON structures.

New in version 1.1.0.

`class sqlalchemy.types.NullType`
An unknown type.

`NullType` is used as a default type for those cases where a type cannot be determined, including:

- During table reflection, when the type of a column is not recognized by the `Dialect`
- When constructing SQL expressions using plain Python objects of unknown types (e.g. `somecolumn == my_special_object`)
- When a new `Column` is created, and the given type is passed as `None` or is not passed at all.

The `NullType` can be used within SQL expression invocation without issue, it just has no behavior either at the expression construction level or at the bind-parameter/result processing level. `NullType` will result in a `CompileError` if the compiler is asked to render the type itself, such as if it is used in a `cast()` operation or within a schema creation operation such as that invoked by `MetaData.create_all()` or the `CreateTable` construct.

`class sqlalchemy.types.Variant(`*base*, *mapping*`)`
A wrapping type that selects among a variety of implementations based on dialect in use.

The `Variant` type is typically constructed using the `TypeEngine.with_variant()` method.

New in version 0.7.2.

See also:

`TypeEngine.with_variant()` for an example of use.

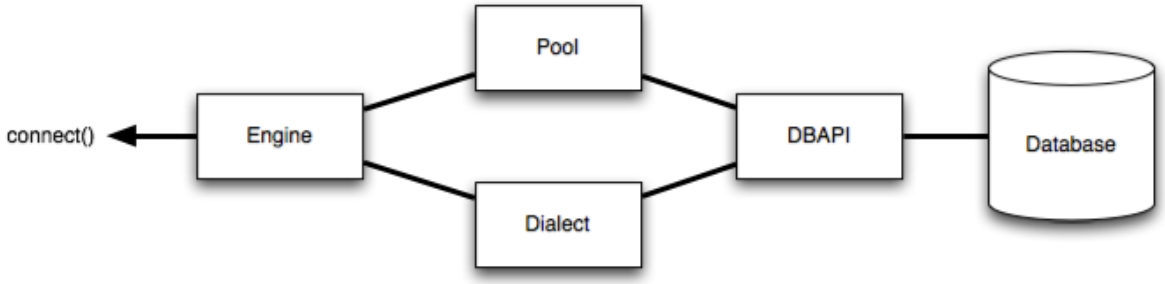
Members with_variant, ___init___

3.5 Engine and Connection Use

3.5.1 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 creates a `Dialect` object tailored towards PostgreSQL, as well as a `Pool` object which will establish a DBAPI connection at `localhost:5432` when a connection request is first received. Note that the `Engine` and its underlying `Pool` do **not** establish the first actual DBAPI connection until the `Engine.connect()` method is called, or an operation which is dependent on this method such as `Engine.execute()` is invoked. In this way, `Engine` and `Pool` can be said to have a *lazy initialization* behavior.

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, connections_toplevel, will detail the usage API of the `Engine` and similar, typically for non-ORM applications.

Supported Databases

SQLAlchemy includes many `Dialect` implementations for various backends. Dialects for the most common databases are included with SQLAlchemy; a handful of others require an additional install of a separate dialect.

See the section dialect_toplevel for information on the various backends available.

Database Urls

The `create_engine()` function produces an `Engine` object based on a URL. These URLs follow RFC-1738, and usually can include username, password, hostname, database name as well as optional keyword arguments for additional configuration. In some cases a file path is accepted, and in others a "data source name" replaces the "host" and "database" portions. The typical form of a database URL is:

```
dialect+driver://username:password@host:port/database
```

Dialect names include the identifying name of the SQLAlchemy dialect, a name such as `sqlite`, `mysql`, `postgresql`, `oracle`, or `mssql`. 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.

Examples for common connection styles follow below. For a full index of detailed information on all included dialects as well as links to third-party dialects, see dialect_toplevel.

PostgreSQL

The PostgreSQL dialect uses psycopg2 as the default DBAPI. pg8000 is also available as a pure-Python substitute:

```
# default
engine = create_engine('postgresql://scott:tiger@localhost/mydatabase')

# psycopg2
engine = create_engine('postgresql+psycopg2://scott:tiger@localhost/mydatabase')

# pg8000
engine = create_engine('postgresql+pg8000://scott:tiger@localhost/mydatabase')
```

More notes on connecting to PostgreSQL at *PostgreSQL*.

MySQL

The MySQL dialect uses mysql-python as the default DBAPI. There are many MySQL DBAPIs available, including MySQL-connector-python and OurSQL:

```
# default
engine = create_engine('mysql://scott:tiger@localhost/foo')

# mysql-python
engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

# MySQL-connector-python
engine = create_engine('mysql+mysqlconnector://scott:tiger@localhost/foo')

# OurSQL
engine = create_engine('mysql+oursql://scott:tiger@localhost/foo')
```

More notes on connecting to MySQL at *MySQL*.

Oracle

The Oracle dialect uses cx_oracle as the default DBAPI:

```
engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('oracle+cx_oracle://scott:tiger@tnsname')
```

More notes on connecting to Oracle at *Oracle*.

Microsoft SQL Server

The SQL Server dialect uses pyodbc as the default DBAPI. pymssql is also available:

```
# pyodbc
engine = create_engine('mssql+pyodbc://scott:tiger@mydsn')

# pymssql
engine = create_engine('mssql+pymssql://scott:tiger@hostname:port/dbname')
```

More notes on connecting to SQL Server at *Microsoft SQL Server*.

SQLite

SQLite connects to file-based databases, using the Python built-in module `sqlite3` by default.

As SQLite connects to local files, the URL format is slightly different. The "file" portion of the URL is the filename of the database. For a relative file path, this requires three slashes:

```
# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')
```

And for an absolute file path, the three slashes are followed by the absolute path:

```
#Unix/Mac - 4 initial slashes in total
engine = create_engine('sqlite:////absolute/path/to/foo.db')
#Windows
engine = create_engine('sqlite:///C:\\path\\to\\foo.db')
#Windows alternative using raw string
engine = create_engine(r'sqlite:///C:\path\to\foo.db')
```

To use a SQLite `:memory:` database, specify an empty URL:

```
engine = create_engine('sqlite://')
```

More notes on connecting to SQLite at *SQLite*.

Others

See dialect_toplevel, the top-level page for all additional dialect documentation.

Engine Creation API

`sqlalchemy.create_engine`(**args*, ***kwargs*)

Create a new `Engine` instance.

The standard calling form is to send the URL as the first positional argument, usually a string that indicates database dialect and connection arguments:

```
engine = create_engine("postgresql://scott:tiger@localhost/test")
```

Additional keyword arguments may then follow it which establish various options on the resulting `Engine` and its underlying `Dialect` and `Pool` constructs:

```
engine = create_engine("mysql://scott:tiger@hostname/dbname",
                            encoding='latin1', echo=True)
```

The string form of the URL is `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.

Once established, the newly resulting `Engine` will request a connection from the underlying `Pool` once `Engine.connect()` is called, or a method which depends on it such as `Engine.execute()` is invoked. The `Pool` in turn will establish the first actual DBAPI connection when this request

is received. The `create_engine()` call itself does **not** establish any actual DBAPI connections directly.

See also:

Engine Configuration

Dialects

connections_toplevel

Parameters

- **`case_sensitive=True`** – if False, result column names will match in a case-insensitive fashion, that is, `row['SomeColumn']`.

  Changed in version 0.8: By default, result row names match case-sensitively. In version 0.7 and prior, all matches were case-insensitive.

- **`connect_args`** – a dictionary of options which will be passed directly to the DBAPI's `connect()` method as additional keyword arguments. See the example at custom_dbapi_args.

- **`convert_unicode=False`** – if set to True, sets the default behavior of `convert_unicode` on the `String` type to `True`, regardless of a setting of `False` on an individual `String` type, thus causing all `String` -based columns to accommodate Python `unicode` objects. This flag is useful as an engine-wide setting when using a DBAPI that does not natively support Python `unicode` objects and raises an error when one is received (such as pyodbc with FreeTDS).

  See `String` for further details on what this flag indicates.

- **`creator`** – a callable which returns a DBAPI connection. This creation function will be passed to the underlying connection pool and will be used to create all new database connections. Usage of this function causes connection parameters specified in the URL argument to be bypassed.

- **`echo=False`** – if True, the Engine will log all statements as well as a repr() of their parameter lists to the engines logger, which defaults to sys.stdout. The `echo` attribute of `Engine` can be modified at any time to turn logging on and off. If set to the string `"debug"`, result rows will be printed to the standard output as well. This flag ultimately controls a Python logger; see dbengine_logging for information on how to configure logging directly.

- **`echo_pool=False`** – if True, the connection pool will log all check-outs/checkins to the logging stream, which defaults to sys.stdout. This flag ultimately controls a Python logger; see dbengine_logging for information on how to configure logging directly.

- **`empty_in_strategy`** – The SQL compilation strategy to use when rendering an IN or NOT IN expression for `ColumnOperators.in_()` where the right-hand side is an empty set. This is a string value that may be one of `static`, `dynamic`, or `dynamic_warn`. The `static` strategy is the default, and an IN comparison to an empty set will generate a simple false expression "1 != 1". The `dynamic` strategy behaves like that of SQLAlchemy 1.1 and earlier, emitting a false expression of the form "expr != expr", which has the effect of evaluting to NULL in the case of a null expression. `dynamic_warn` is the same as `dynamic`, however also emits a warning when an empty set is encountered; this because the "dynamic" comparison is typically poorly performing on most databases.

  New in version 1.2: Added the `empty_in_strategy` setting and additionally defaulted the behavior for empty-set IN comparisons to a static boolean expression.

- `encoding` – Defaults to `utf-8`. This is the string encoding used by SQLAlchemy for string encode/decode operations which occur within SQLAlchemy, **outside of the DBAPI.** Most modern DBAPIs feature some degree of direct support for Python `unicode` objects, what you see in Python 2 as a string of the form `u'some string'`. For those scenarios where the DBAPI is detected as not supporting a Python `unicode` object, this encoding is used to determine the source/destination encoding. It is **not used** for those cases where the DBAPI handles unicode directly.

  To properly configure a system to accommodate Python `unicode` objects, the DBAPI should be configured to handle unicode to the greatest degree as is appropriate - see the notes on unicode pertaining to the specific target database in use at dialect_toplevel.

  Areas where string encoding may need to be accommodated outside of the DBAPI include zero or more of:

  – the values passed to bound parameters, corresponding to the `Unicode` type or the `String` type when `convert_unicode` is `True`;

  – the values returned in result set columns corresponding to the `Unicode` type or the `String` type when `convert_unicode` is `True`;

  – the string SQL statement passed to the DBAPI's `cursor.execute()` method;

  – the string names of the keys in the bound parameter dictionary passed to the DBAPI's `cursor.execute()` as well as `cursor.setinputsizes()` methods;

  – the string column names retrieved from the DBAPI's `cursor.description` attribute.

  When using Python 3, the DBAPI is required to support *all* of the above values as Python `unicode` objects, which in Python 3 are just known as `str`. In Python 2, the DBAPI does not specify unicode behavior at all, so SQLAlchemy must make decisions for each of the above values on a per-DBAPI basis - implementations are completely inconsistent in their behavior.

- `execution_options` – Dictionary execution options which will be applied to all connections. See `execution_options()`

- `implicit_returning=True` – When `True`, a RETURNING- compatible construct, if available, will be used to fetch newly generated primary key values when a single row INSERT statement is emitted with no existing returning() clause. This applies to those backends which support RETURNING or a compatible construct, including PostgreSQL, Firebird, Oracle, Microsoft SQL Server. Set this to `False` to disable the automatic usage of RETURNING.

- `isolation_level` – this string parameter is interpreted by various dialects in order to affect the transaction isolation level of the database connection. The parameter essentially accepts some subset of these string arguments: `"SERIALIZABLE"`, `"REPEATABLE_READ"`, `"READ_COMMITTED"`, `"READ_UNCOMMITTED"` and `"AUTOCOMMIT"`. Behavior here varies per backend, and individual dialects should be consulted directly.

  Note that the isolation level can also be set on a per-`Connection` basis as well, using the `Connection.execution_options.isolation_level` feature.

  See also:

  `Connection.default_isolation_level` - view default level

  `Connection.execution_options.isolation_level` - set per `Connection` isolation level

SQLite Transaction Isolation

PostgreSQL Transaction Isolation

MySQL Transaction Isolation

session_transaction_isolation - for the ORM

- **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.
- **paramstyle=None** – The paramstyle to use when rendering bound parameters. This style defaults to the one recommended by the DBAPI itself, which is retrieved from the .paramstyle attribute of the DBAPI. However, most DBAPIs accept more than one paramstyle, and in particular it may be desirable to change a "named" paramstyle into a "positional" one, or vice versa. When this attribute is passed, it should be one of the values "qmark", "numeric", "named", "format" or "pyformat", and should correspond to a parameter style known to be supported by the DBAPI in use.
- **pool=None** – an already-constructed instance of Pool, such as a QueuePool instance. If non-None, this pool will be used directly as the underlying connection pool for the engine, bypassing whatever connection parameters are present in the URL argument. For information on constructing connection pools manually, see pooling_toplevel.
- **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_pre_ping** – boolean, if True will enable the connection pool "pre-ping" feature that tests connections for liveness upon each checkout.

  New in version 1.2.

  See also:

  pool_disconnects_pessimistic

- **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).

  See also:

  pool_setting_recycle

- **pool_reset_on_return='rollback'** – set the "reset on return" behavior of the pool, which is whether rollback(), commit(), or nothing is called upon connections being returned to the pool. See the docstring for reset_on_return at Pool.

  New in version 0.7.6.

- **pool_timeout=30** – number of seconds to wait before giving up on getting a connection from the pool. This is only used with QueuePool.

- **plugins** – string list of plugin names to load. See CreateEnginePlugin for background.

  New in version 1.2.3.

- **strategy='plain'** – selects alternate engine implementations. Currently available are:

  – the threadlocal strategy, which is described in threadlocal_strategy;

  – the mock strategy, which dispatches all statement execution to a function passed as the argument executor. See example in the FAQ.

- **executor=None** – a function taking arguments (sql, *multiparams, **params), to which the mock strategy will dispatch all statement execution. Used only by strategy='mock'.

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.

The keys of interest to engine_from_config() should be prefixed, e.g. sqlalchemy.url, sqlalchemy.echo, etc. The 'prefix' argument indicates the prefix to be searched for. Each matching key (after the prefix is stripped) is treated as though it were the corresponding keyword argument to a create_engine() call.

The only required key is (assuming the default prefix) sqlalchemy.url, which provides the database URL.

A select set of keyword arguments will be "coerced" to their expected type based on string values. The set of arguments is extensible per-dialect using the engine_config_types accessor.

Parameters

- **configuration** – A dictionary (typically produced from a config file, but this is not a requirement). Items whose keys start with the value of 'prefix' will have that prefix stripped, and will then be passed to create_engine.

- **prefix** – Prefix to match and then strip from keys in 'configuration'.

- **kwargs** – Each keyword argument to engine_from_config() itself overrides the corresponding item taken from the 'configuration' dictionary. Keyword arguments should *not* be prefixed.

sqlalchemy.engine.url.**make_url**(*name_or_url*)
Given a string or unicode instance, produce a new URL instance.

The given string is parsed according to the RFC 1738 spec. If an existing URL object is passed, just returns the object.

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 `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.

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.

Pooling

The `Engine` will ask the connection pool for a connection when the `connect()` or `execute()` methods are called. The default connection pool, `QueuePool`, 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.

Note: `QueuePool` is not used by default for SQLite engines. See *SQLite* for details on SQLite connection pool usage.

For more information on connection pooling, see pooling_toplevel.

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':17,
↪'argument2':'bar'})
```

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)
```

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.WARN` 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()`.

Note: The SQLAlchemy `Engine` conserves Python function call overhead by only emitting log statements when the current logging level is detected as `logging.INFO` or `logging.DEBUG`. It only checks this level when a new connection is procured from the connection pool. Therefore when changing the logging configuration for an already-running application, any `Connection` that's currently active, or more commonly a `Session` object that's active in a transaction, won't log any SQL according to the new configuration until a new `Connection` is procured (in the case of `Session`, this is after the current transaction ends and a new one begins).

3.5.2 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.

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 `Engine.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 released 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 sqlexpression_toplevel.

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 unitofwork_transaction 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 invoke Transaction.rollback() or Transaction.commit():

```
connection = engine.connect()
trans = connection.begin()
try:
    r1 = connection.execute(table1.select())
    connection.execute(table1.insert(), col1=7, col2='this is some data')
    trans.commit()
except:
    trans.rollback()
    raise
```

The above block can be created more succinctly using context managers, either given an Engine:

```
# runs a transaction
with engine.begin() as connection:
    r1 = connection.execute(table1.select())
    connection.execute(table1.insert(), col1=7, col2='this is some data')
```

Or from the `Connection`, in which case the `Transaction` object is available as well:

```
with connection.begin() as trans:
    r1 = connection.execute(table1.select())
    connection.execute(table1.insert(), col1=7, col2='this is some data')
```

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's␣
↪transaction
    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.

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`? While some DBAPI implementations provide various special "non-transactional" modes, the core behavior of DBAPI per PEP-0249 is that a *transaction is always in progress*, providing only `rollback()` and `commit()` methods but no `begin()`. SQLAlchemy assumes this is the case for any given DBAPI.

Given this requirement, SQLAlchemy implements its own "autocommit" feature 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 the presence of the `autocommit=True` execution option on the statement. If the statement is a text-only statement and the flag is not set, a regular expression is used to detect INSERT, UPDATE, DELETE, as well as a variety of other commands for a particular backend:

```
conn = engine.connect()
conn.execute("INSERT INTO users VALUES (1, 'john')")  # autocommits
```

The "autocommit" feature is only in effect when no `Transaction` has otherwise been declared. This means the feature is not generally used with the ORM, as the `Session` object by default always maintains an ongoing `Transaction`.

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))
```

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`:

```
result = engine.execute("select username from users")
for row in result:
    print("username:", row['username'])
```

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:

```
from sqlalchemy import MetaData, Table, Column, Integer

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 makes usage of the `execute()` method on the expression itself. This method is provided as part of the `Executable` class, which refers to a SQL statement that is sufficient for being invoked against the database. The method makes usage of the assumption that either an `Engine` or `Connection` has been **bound** to the expression object. By "bound" we mean that the special attribute `MetaData.bind` has been used to associate a series of `Table` objects and all SQL constructs derived from them with a specific engine:

```
engine = create_engine('sqlite:///file.db')
meta.bind = engine
result = users_table.select().execute()
for row in result:
    # ....
result.close()
```

Above, we associate an `Engine` with a `MetaData` object using the special attribute `MetaData.bind`. The `select()` construct produced from the `Table` object has a method `execute()`, which will search for an `Engine` that's "bound" to the `Table`.

Overall, the usage of "bound metadata" has three general effects:

- SQL statement objects gain an `Executable.execute()` method which automatically locates a "bind" with which to execute themselves.
- The ORM `Session` object supports using "bound metadata" in order to establish which `Engine` should be used to invoke SQL statements on behalf of a particular mapped class, though the `Session` also features its own explicit system of establishing complex `Engine`/ mapped class configurations.
- The `MetaData.create_all()`, `MetaData.drop_all()`, `Table.create()`, `Table.drop()`, and "autoload" features all make usage of the bound `Engine` automatically without the need to pass it explicitly.

Note: The concepts of "bound metadata" and "implicit execution" are not emphasized in modern SQLAlchemy. While they offer some convenience, they are no longer required by any API and are never necessary.

In applications where multiple `Engine` objects are present, each one logically associated with a certain set of tables (i.e. *vertical sharding*), the "bound metadata" technique can be used so that individual `Table` can refer to the appropriate `Engine` automatically; in particular this is supported within the ORM via the `Session` object as a means to associate `Table` objects with an appropriate `Engine`, as an alternative to using the bind arguments accepted directly by the `Session`.

However, the "implicit execution" technique is not at all appropriate for use with the ORM, as it bypasses the transactional context maintained by the `Session`.

Overall, in the *vast majority* of cases, "bound metadata" and "implicit execution" are **not useful**. While "bound metadata" has a marginal level of usefulness with regards to ORM configuration, "implicit execution" is a very old usage pattern that in most cases is more confusing than it is helpful, and its usage is discouraged. Both patterns seem to encourage the overuse of expedient "short cuts" in application design which lead to problems later on.

Modern SQLAlchemy usage, especially the ORM, places a heavy stress on working within the context of a transaction at all times; the "implicit execution" concept makes the job of associating statement execution with a particular transaction much more difficult. The `Executable.execute()` method on a particular SQL statement usually implies that the execution is not part of any particular transaction,

which is usually not the desired effect.

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.

Translation of Schema Names

To support multi-tenancy applications that distribute common sets of tables into multiple schemas, the `Connection.execution_options.schema_translate_map` execution option may be used to repurpose a set of `Table` objects to render under different schema names without any changes.

Given a table:

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

The "schema" of this `Table` as defined by the `Table.schema` attribute is `None`. The `Connection.execution_options.schema_translate_map` can specify that all `Table` objects with a schema of `None` would instead render the schema as `user_schema_one`:

```
connection = engine.connect().execution_options(
    schema_translate_map={None: "user_schema_one"})

result = connection.execute(user_table.select())
```

The above code will invoke SQL on the database of the form:

```
SELECT user_schema_one.user.id, user_schema_one.user.name FROM
user_schema.user
```

That is, the schema name is substituted with our translated name. The map can specify any number of target->destination schemas:

```
connection = engine.connect().execution_options(
    schema_translate_map={
        None: "user_schema_one",      # no schema name -> "user_schema_one"
        "special": "special_schema", # schema="special" becomes "special_schema"
        "public": None               # Table objects with schema="public" will render with no␣
↪schema
    })
```

The `Connection.execution_options.schema_translate_map` parameter affects all DDL and SQL constructs generated from the SQL expression language, as derived from the `Table` or `Sequence` objects. It does **not** impact literal string SQL used via the `expression.text()` construct nor via plain strings passed to `Connection.execute()`.

The feature takes effect **only** in those cases where the name of the schema is derived directly from that of a `Table` or `Sequence`; it does not impact methods where a string schema name is passed directly. By this pattern, it takes effect within the "can create" / "can drop" checks performed by methods such as `MetaData.create_all()` or `MetaData.drop_all()` are called, and it takes effect when using table reflection given a `Table` object. However it does **not** affect the operations present on the `Inspector` object, as the schema name is passed to these methods explicitly.

New in version 1.1.

Engine Disposal

The `Engine` refers to a connection pool, which means under normal circumstances, there are open database connections present while the `Engine` object is still resident in memory. When an `Engine` is garbage collected, its connection pool is no longer referred to by that `Engine`, and assuming none of its connections are still checked out, the pool and its connections will also be garbage collected, which has the effect of closing out the actual database connections as well. But otherwise, the `Engine` will hold onto open database connections assuming it uses the normally default pool implementation of `QueuePool`.

The `Engine` is intended to normally be a permanent fixture established up-front and maintained throughout the lifespan of an application. It is **not** intended to be created and disposed on a per-connection basis; it is instead a registry that maintains both a pool of connections as well as configurational information about the database and DBAPI in use, as well as some degree of internal caching of per-database resources.

However, there are many cases where it is desirable that all connection resources referred to by the `Engine` be completely closed out. It's generally not a good idea to rely on Python garbage collection for this to occur for these cases; instead, the `Engine` can be explicitly disposed using the `Engine.dispose()` method. This disposes of the engine's underlying connection pool and replaces it with a new one that's empty. Provided that the `Engine` is discarded at this point and no longer used, all **checked-in** connections which it refers to will also be fully closed.

Valid use cases for calling `Engine.dispose()` include:

- When a program wants to release any remaining checked-in connections held by the connection pool and expects to no longer be connected to that database at all for any future operations.
- When a program uses multiprocessing or `fork()`, and an `Engine` object is copied to the child process, `Engine.dispose()` should be called so that the engine creates brand new database connections local to that fork. Database connections generally do **not** travel across process boundaries.
- Within test suites or multitenancy scenarios where many ad-hoc, short-lived `Engine` objects may be created and disposed.

Connections that are **checked out** are **not** discarded when the engine is disposed or garbage collected, as these connections are still strongly referenced elsewhere by the application. However, after `Engine.dispose()` is called, those connections are no longer associated with that `Engine`; when they are closed, they will be returned to their now-orphaned connection pool which will ultimately be garbage collected, once all connections which refer to it are also no longer referenced anywhere. Since this process is not easy to control, it is strongly recommended that `Engine.dispose()` is called only after all checked out connections are checked in or otherwise de-associated from their pool.

An alternative for applications that are negatively impacted by the `Engine` object's use of connection pooling is to disable pooling entirely. This typically incurs only a modest performance impact upon the use of new connections, and means that when a connection is checked in, it is entirely closed out and is not held in memory. See pool_switching for guidelines on how to disable pooling.

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`.

Note: The "threadlocal" feature is generally discouraged. It's designed for a particular pattern of usage which is generally considered as a legacy pattern. 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.

Working with Raw DBAPI Connections

There are some cases where SQLAlchemy does not provide a genericized way at accessing some DBAPI functions, such as calling stored procedures as well as dealing with multiple result sets. In these cases, it's just as expedient to deal with the raw DBAPI connection directly.

The most common way to access the raw DBAPI connection is to get it from an already present `Connection` object directly. It is present using the `Connection.connection` attribute:

```
connection = engine.connect()
dbapi_conn = connection.connection
```

The DBAPI connection here is actually a "proxied" in terms of the originating connection pool, however this is an implementation detail that in most cases can be ignored. As this DBAPI connection is still

contained within the scope of an owning `Connection` object, it is best to make use of the `Connection` object for most features such as transaction control as well as calling the `Connection.close()` method; if these operations are performed on the DBAPI connection directly, the owning `Connection` will not be aware of these changes in state.

To overcome the limitations imposed by the DBAPI connection that is maintained by an owning `Connection`, a DBAPI connection is also available without the need to procure a `Connection` first, using the `Engine.raw_connection()` method of `Engine`:

```
dbapi_conn = engine.raw_connection()
```

This DBAPI connection is again a "proxied" form as was the case before. The purpose of this proxying is now apparent, as when we call the `.close()` method of this connection, the DBAPI connection is typically not actually closed, but instead released back to the engine's connection pool:

```
dbapi_conn.close()
```

While SQLAlchemy may in the future add built-in patterns for more DBAPI use cases, there are diminishing returns as these cases tend to be rarely needed and they also vary highly dependent on the type of DBAPI in use, so in any case the direct DBAPI calling pattern is always there for those cases where it is needed.

Some recipes for DBAPI connection use follow.

Calling Stored Procedures

For stored procedures with special syntactical or parameter concerns, DBAPI-level callproc may be used:

```
connection = engine.raw_connection()
try:
    cursor = connection.cursor()
    cursor.callproc("my_procedure", ['x', 'y', 'z'])
    results = list(cursor.fetchall())
    cursor.close()
    connection.commit()
finally:
    connection.close()
```

Multiple Result Sets

Multiple result set support is available from a raw DBAPI cursor using the nextset method:

```
connection = engine.raw_connection()
try:
    cursor = connection.cursor()
    cursor.execute("select * from table1; select * from table2")
    results_one = cursor.fetchall()
    cursor.nextset()
    results_two = cursor.fetchall()
    cursor.close()
finally:
    connection.close()
```

Registering New Dialects

The `create_engine()` function call locates the given dialect using setuptools entrypoints. These entry points can be established for third party dialects within the setup.py script. For example, to create a new dialect "foodialect://", the steps are as follows:

1. Create a package called `foodialect`.
2. The package should have a module containing the dialect class, which is typically a subclass of `sqlalchemy.engine.default.DefaultDialect`. In this example let's say it's called `FooDialect` and its module is accessed via `foodialect.dialect`.
3. The entry point can be established in setup.py as follows:

```
entry_points="""
[sqlalchemy.dialects]
foodialect = foodialect.dialect:FooDialect
"""
```

If the dialect is providing support for a particular DBAPI on top of an existing SQLAlchemy-supported database, the name can be given including a database-qualification. For example, if `FooDialect` were in fact a MySQL dialect, the entry point could be established like this:

```
entry_points="""
[sqlalchemy.dialects]
mysql.foodialect = foodialect.dialect:FooDialect
"""
```

The above entrypoint would then be accessed as `create_engine("mysql+foodialect://")`.

Registering Dialects In-Process

SQLAlchemy also allows a dialect to be registered within the current process, bypassing the need for separate installation. Use the `register()` function as follows:

```
from sqlalchemy.dialects import registry
registry.register("mysql.foodialect", "myapp.dialect", "MyMySQLDialect")
```

The above will respond to `create_engine("mysql+foodialect://")` and load the `MyMySQLDialect` class from the `myapp.dialect` module.

New in version 0.8.

Connection / Engine API

class sqlalchemy.engine.**Connection**(*engine*, *connection=None*, *close_with_result=False*, *_branch_from=None*, *_execution_options=None*, *_dispatch=None*, *_has_events=None*)

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.

begin()

Begin a transaction and return a transaction handle.

The returned object is an instance of `Transaction`. This object represents the "scope" of the transaction, which completes when either the `Transaction.rollback()` or `Transaction.commit()` method is called.

Nested calls to `begin()` on the same `Connection` will return new `Transaction` objects that represent an emulated transaction within the scope of the enclosing transaction, that is:

```
trans = conn.begin()   # outermost transaction
trans2 = conn.begin()  # "nested"
trans2.commit()        # does nothing
trans.commit()         # actually commits
```

Calls to `Transaction.commit()` only have an effect when invoked via the outermost `Transaction` object, though the `Transaction.rollback()` method of any of the `Transaction` objects will roll back the transaction.

See also:

`Connection.begin_nested()` - use a SAVEPOINT

`Connection.begin_twophase()` - use a two phase /XID transaction

`Engine.begin()` - context manager available from `Engine`.

`begin_nested()`
Begin a nested transaction and return a transaction handle.

The returned object is an instance of `NestedTransaction`.

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.

See also `Connection.begin()`, `Connection.begin_twophase()`.

`begin_twophase`(*xid=None*)
Begin a two-phase or XA transaction and return a transaction handle.

The returned object is an instance of `TwoPhaseTransaction`, which in addition to the methods provided by `Transaction`, also provides a `prepare()` method.

Parameters `xid` – the two phase transaction id. If not supplied, a random id will be generated.

See also `Connection.begin()`, `Connection.begin_twophase()`.

`close()`
Close this `Connection`.

This results in a release of the underlying database resources, that is, the DBAPI connection referenced internally. The DBAPI connection is typically restored back to the connection-holding `Pool` referenced by the `Engine` that produced this `Connection`. Any transactional state present on the DBAPI connection is also unconditionally released via the DBAPI connection's `rollback()` method, regardless of any `Transaction` object that may be outstanding with regards to this `Connection`.

After `close()` is called, the `Connection` is permanently in a closed state, and will allow no further operations.

`closed`
Return True if this connection is closed.

`connect()`
Returns a branched version of this `Connection`.

The `Connection.close()` method on the returned `Connection` can be called and this `Connection` will remain open.

This method provides usage symmetry with `Engine.connect()`, including for usage with context managers.

`connection`
The underlying DB-API connection managed by this Connection.

See also:

dbapi_connections

`contextual_connect`(***kwargs*)
Returns a branched version of this `Connection`.

The `Connection.close()` method on the returned `Connection` can be called and this `Connection` will remain open.

This method provides usage symmetry with `Engine.contextual_connect()`, including for usage with context managers.

`default_isolation_level`
The default isolation level assigned to this `Connection`.

This is the isolation level setting that the `Connection` has when first procured via the `Engine.connect()` method. This level stays in place until the `Connection.execution_options.isolation_level` is used to change the setting on a per-`Connection` basis.

Unlike `Connection.get_isolation_level()`, this attribute is set ahead of time from the first connection procured by the dialect, so SQL query is not invoked when this accessor is called.

New in version 0.9.9.

See also:

`Connection.get_isolation_level()` - view current level

`create_engine.isolation_level` - set per `Engine` isolation level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`detach()`
Detach the underlying DB-API connection from its connection pool.

E.g.:

```
with engine.connect() as conn:
    conn.detach()
    conn.execute("SET search_path TO schema1, schema2")

    # work with connection

# connection is fully closed (since we used "with:", can
# also call .close())
```

This `Connection` instance will remain usable. When closed (or exited from a context manager context as above), the DB-API connection will be literally closed and not returned to its originating pool.

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).

`execute`(*object*, **multiparams*, ***params*)
Executes a SQL statement construct and returns a `ResultProxy`.

Parameters

- `object` – The statement to be executed. May be one of:
  - a plain 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

- ***multiparams/**params** – represent bound parameter values to be used in the execution. Typically, the format is either a collection of one or more dictionaries passed to *multiparams:

```
conn.execute(
    table.insert(),
    {"id":1, "value":"v1"},
    {"id":2, "value":"v2"}
)
```

...or individual key/values interpreted by **params:

```
conn.execute(
    table.insert(), id=1, value="v1"
)
```

In the case that a plain SQL string is passed, and the underlying DBAPI accepts positional bind parameters, a collection of tuples or individual values in *multiparams may be passed:

```
conn.execute(
    "INSERT INTO table (id, value) VALUES (?, ?)",
    (1, "v1"), (2, "v2")
)

conn.execute(
    "INSERT INTO table (id, value) VALUES (?, ?)",
    1, "v1"
)
```

Note above, the usage of a question mark "?" or other symbol is contingent upon the "paramstyle" accepted by the DBAPI in use, which may be any of "qmark", "named", "pyformat", "format", "numeric". See pep-249 for details on paramstyle.

To execute a textual SQL statement which uses bound parameters in a DBAPI-agnostic way, use the `text()` construct.

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's usually a good idea to ensure that the copies will be discarded immediately, which is implicit if used as in:

```
result = connection.execution_options(stream_results=True).\
                    execute(stmt)
```

Note that any key/value can be passed to `Connection.execution_options()`, and it will be stored in the `_execution_options` dictionary of the `Connection`. It is suitable for usage by end-user schemes to communicate with event listeners, for example.

The keywords that are currently recognized by SQLAlchemy itself include all those listed under `Executable.execution_options()`, as well as others that are specific to `Connection`.

Parameters

- `autocommit` – Available on: Connection, statement. 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), and an explicit transaction is not in progress.

- `compiled_cache` – Available on: Connection. A dictionary where `Compiled` objects will be cached when the `Connection` compiles a clause expression into a `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.

  Note that the ORM makes use of its own "compiled" caches for some operations, including flush operations. The caching used by the ORM internally supersedes a cache dictionary specified here.

- `isolation_level` – Available on: `Connection`. Set the transaction isolation level for the lifespan of this `Connection` object (*not* the underlying DBAPI connection, for which the level is reset to its original setting upon termination of this `Connection` object).

  Valid values include those string values accepted by the `create_engine.isolation_level` parameter passed to `create_engine()`. These levels are semi-database specific; see individual dialect documentation for valid levels.

  Note that this option necessarily affects the underlying DBAPI connection for the lifespan of the originating `Connection`, and is not per-execution. This setting is not removed until the underlying DBAPI connection is returned to the connection pool, i.e. the `Connection.close()` method is called.

  Warning: The `isolation_level` execution option should **not** be used when a transaction is already established, that is, the `Connection.begin()` method or similar has been called. A database cannot change the isolation level on a transaction in progress, and different DBAPIs and/or SQLAlchemy dialects may implicitly roll back or commit the transaction, or not affect the connection at all.

  Changed in version 0.9.9: A warning is emitted when the `isolation_level` execution option is used after a transaction has been started with `Connection.begin()` or similar.

  Note: The `isolation_level` execution option is implicitly reset if the `Connection` is invalidated, e.g. via the `Connection.invalidate()` method, or if a disconnection error occurs. The new connection produced after the invalidation will not have the isolation level re-applied to it

automatically.

See also:

`create_engine.isolation_level` - set per `Engine` isolation level

`Connection.get_isolation_level()` - view current level

SQLite Transaction Isolation

PostgreSQL Transaction Isolation

MySQL Transaction Isolation

SQL Server Transaction Isolation

session_transaction_isolation - for the ORM

- **`no_parameters`** – When `True`, if the final parameter list or dictionary is totally empty, will invoke the statement on the cursor as `cursor.execute(statement)`, not passing the parameter collection at all. Some DBAPIs such as psycopg2 and mysql-python consider percent signs as significant only when parameters are present; this option allows code to generate SQL containing percent signs (and possibly other characters) that is neutral regarding whether it's executed by the DBAPI or piped into a script that's later invoked by command line tools.

  New in version 0.7.6.

- **`stream_results`** – Available on: Connection, statement. 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, mysqldb and pymysql dialects.

- **`schema_translate_map`** – Available on: Connection, Engine. A dictionary mapping schema names to schema names, that will be applied to the `Table.schema` element of each `Table` encountered when SQL or DDL expression elements are compiled into strings; the resulting schema name will be converted based on presence in the map of the original name.

  New in version 1.1.

  See also:

  schema_translating

`get_isolation_level()`

Return the current isolation level assigned to this `Connection`.

This will typically be the default isolation level as determined by the dialect, unless if the `Connection.execution_options.isolation_level` feature has been used to alter the isolation level on a per-`Connection` basis.

This attribute will typically perform a live SQL operation in order to procure the current isolation level, so the value returned is the actual level on the underlying DBAPI connection regardless of how this state was set. Compare to the `Connection.default_isolation_level` accessor which returns the dialect-level setting without performing a SQL query.

New in version 0.9.9.

See also:

`Connection.default_isolation_level` - view default level

`create_engine.isolation_level` - set per `Engine` isolation level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`in_transaction()`
Return True if a transaction is in progress.

`info`
Info dictionary associated with the underlying DBAPI connection referred to by this `Connection`, allowing user-defined data to be associated with the connection.

The data here will follow along with the DBAPI connection including after it is returned to the connection pool and used again in subsequent instances of `Connection`.

`invalidate(`*exception=None*`)`
Invalidate the underlying DBAPI connection associated with this `Connection`.

The underlying DBAPI 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 use (where "use" typically means using the `Connection.execute()` method or similar), this `Connection` will attempt to procure a new DBAPI connection using the services of the `Pool` as a source of connectivity (e.g. a "reconnection").

If a transaction was in progress (e.g. the `Connection.begin()` method has been called) when `Connection.invalidate()` method is called, at the DBAPI level all state associated with this transaction is lost, as the DBAPI connection is closed. The `Connection` will not allow a reconnection to proceed until the `Transaction` object is ended, by calling the `Transaction.rollback()` method; until that point, any attempt at continuing to use the `Connection` will raise an `InvalidRequestError`. This is to prevent applications from accidentally continuing an ongoing transactional operations despite the fact that the transaction has been lost due to an invalidation.

The `Connection.invalidate()` method, just like auto-invalidation, will at the connection pool level invoke the `PoolEvents.invalidate()` event.

See also:

pool_connection_invalidation

`invalidated`
Return True if this connection was invalidated.

`run_callable(`*callable_*, **args*, ***kwargs*`)`
Given a callable object or function, execute it, passing a `Connection` as the first argument.

The given *args and **kwargs are passed subsequent to the `Connection` argument.

This function, along with `Engine.run_callable()`, allows a function to be run with a `Connection` or `Engine` object without the need to know which one is being dealt with.

`scalar(`*object*, **multiparams*, ***params*`)`
Executes and returns the first column of the first row.

The underlying result/cursor is closed after execution.

`schema_for_object = <sqlalchemy.sql.schema._SchemaTranslateMap object>`
Return the ".schema" attribute for an object.

Used for `Table`, `Sequence` and similar objects, and takes into account the `Connection.execution_options.schema_translate_map` parameter.

New in version 1.1.

See also:

schema_translating

`transaction(`*callable_*, **args*, ***kwargs*`)`
Execute the given function within a transaction boundary.

The function is passed this `Connection` as the first argument, followed by the given *args and **kwargs, e.g.:

```
def do_something(conn, x, y):
    conn.execute("some statement", {'x':x, 'y':y})

conn.transaction(do_something, 5, 10)
```

The operations inside the function are all invoked within the context of a single `Transaction`. Upon success, the transaction is committed. If an exception is raised, the transaction is rolled back before propagating the exception.

Note: The `transaction()` method is superseded by the usage of the Python `with:` statement, which can be used with `Connection.begin()`:

```
with conn.begin():
    conn.execute("some statement", {'x':5, 'y':10})
```

As well as with `Engine.begin()`:

```
with engine.begin() as conn:
    conn.execute("some statement", {'x':5, 'y':10})
```

See also:

`Engine.begin()` - engine-level transactional context

`Engine.transaction()` - engine-level version of `Connection.transaction()`

class sqlalchemy.engine.Connectable

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.

connect(***kwargs*)

Return a `Connection` object.

Depending on context, this may be `self` if this object is already an instance of `Connection`, or a newly procured `Connection` if this object is an instance of `Engine`.

contextual_connect()

Return a `Connection` object which may be part of an ongoing context.

Depending on context, this may be `self` if this object is already an instance of `Connection`, or a newly procured `Connection` if this object is an instance of `Engine`.

create(*entity*, ***kwargs*)

Emit CREATE statements for the given schema entity.

Deprecated since version 0.7: Use the create() method on the given schema object directly, i.e. `Table.create()`, `Index.create()`, `MetaData.create_all()`

drop(*entity*, ***kwargs*)

Emit DROP statements for the given schema entity.

Deprecated since version 0.7: Use the drop() method on the given schema object directly, i.e. `Table.drop()`, `Index.drop()`, `MetaData.drop_all()`

execute(*object*, **multiparams*, ***params*)

Executes the given construct and returns a `ResultProxy`.

scalar(*object*, **multiparams*, ***params*)

Executes and returns the first column of the first row.

The underlying cursor is closed after execution.

class sqlalchemy.engine.CreateEnginePlugin(*url*, *kwargs*)

A set of hooks intended to augment the construction of an Engine object based on entrypoint names in a URL.

The purpose of CreateEnginePlugin is to allow third-party systems to apply engine, pool and dialect level event listeners without the need for the target application to be modified; instead, the plugin names can be added to the database URL. Target applications for CreateEnginePlugin include:

- connection and SQL performance tools, e.g. which use events to track number of checkouts and/or time spent with statements
- connectivity plugins such as proxies

Plugins are registered using entry points in a similar way as that of dialects:

```
entry_points={
    'sqlalchemy.plugins': [
        'myplugin = myapp.plugins:MyPlugin'
    ]
```

A plugin that uses the above names would be invoked from a database URL as in:

```
from sqlalchemy import create_engine

engine = create_engine(
  "mysql+pymysql://scott:tiger@localhost/test?plugin=myplugin")
```

Alternatively, the create_engine.plugins" argument may be passed as a list to :func:.create_engine‘:

```
engine = create_engine(
  "mysql+pymysql://scott:tiger@localhost/test",
  plugins=["myplugin"])
```

New in version 1.2.3: plugin names can also be specified to create_engine() as a list

The plugin argument supports multiple instances, so that a URL may specify multiple plugins; they are loaded in the order stated in the URL:

```
engine = create_engine(
  "mysql+pymysql://scott:tiger@localhost/"
  "test?plugin=plugin_one&plugin=plugin_twp&plugin=plugin_three")
```

A plugin can receive additional arguments from the URL string as well as from the keyword arguments passed to create_engine(). The URL object and the keyword dictionary are passed to the constructor so that these arguments can be extracted from the url's URL.query collection as well as from the dictionary:

```
class MyPlugin(CreateEnginePlugin):
    def __init__(self, url, kwargs):
        self.my_argument_one = url.query.pop('my_argument_one')
        self.my_argument_two = url.query.pop('my_argument_two')
        self.my_argument_three = kwargs.pop('my_argument_three', None)
```

Arguments like those illustrated above would be consumed from the following:

```
from sqlalchemy import create_engine

engine = create_engine(
  "mysql+pymysql://scott:tiger@localhost/"
  "test?plugin=myplugin&my_argument_one=foo&my_argument_two=bar",
  my_argument_three='bat')
```

The URL and dictionary are used for subsequent setup of the engine as they are, so the plugin

can modify their arguments in-place. Arguments that are only understood by the plugin should be popped or otherwise removed so that they aren't interpreted as erroneous arguments afterwards.

When the engine creation process completes and produces the `Engine` object, it is again passed to the plugin via the `CreateEnginePlugin.engine_created()` hook. In this hook, additional changes can be made to the engine, most typically involving setup of events (e.g. those defined in core_event_toplevel).

New in version 1.1.

engine_created(*engine*)
Receive the `Engine` object when it is fully constructed.

The plugin may make additional changes to the engine, such as registering engine or connection pool events.

handle_dialect_kwargs(*dialect_cls*, *dialect_args*)
parse and modify dialect kwargs

handle_pool_kwargs(*pool_cls*, *pool_args*)
parse and modify pool kwargs

class sqlalchemy.engine.Engine(*pool*, *dialect*, *url*, *logging_name=None*, *echo=None*, *proxy=None*, *execution_options=None*)
Connects a `Pool` and `Dialect` together to provide a source of database connectivity and behavior.

An `Engine` object is instantiated publicly using the `create_engine()` function.

See also:

Engine Configuration

connections_toplevel

begin(*close_with_result=False*)
Return a context manager delivering a `Connection` with a `Transaction` established.

E.g.:

```
with engine.begin() as conn:
    conn.execute("insert into table (x, y, z) values (1, 2, 3)")
    conn.execute("my_special_procedure(5)")
```

Upon successful operation, the `Transaction` is committed. If an error is raised, the `Transaction` is rolled back.

The `close_with_result` flag is normally `False`, and indicates that the `Connection` will be closed when the operation is complete. When set to `True`, it indicates the `Connection` is in "single use" mode, where the `ResultProxy` returned by the first call to `Connection.execute()` will close the `Connection` when that `ResultProxy` has exhausted all result rows.

New in version 0.7.6.

See also:

`Engine.connect()` - procure a `Connection` from an `Engine`.

`Connection.begin()` - start a `Transaction` for a particular `Connection`.

connect(***kwargs*)
Return a new `Connection` object.

The `Connection` object is a facade that uses a DBAPI connection internally in order to communicate with the database. This connection is procured from the connection-holding `Pool` referenced by this `Engine`. When the `close()` method of the `Connection` object is called, the underlying DBAPI connection is then returned to the connection pool, where it may be used again in a subsequent call to `connect()`.

`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.

`dispose`()
Dispose of the connection pool used by this `Engine`.

This has the effect of fully closing all **currently checked in** database connections. Connections that are still checked out will **not** be closed, however they will no longer be associated with this `Engine`, so when they are closed individually, eventually the `Pool` which they are associated with will be garbage collected and they will be closed out fully, if not already closed on checkin.

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, so as long as the `Engine` isn't used again, no new connections will be made.

See also:

engine_disposal

`driver`
Driver name of the `Dialect` in use by this `Engine`.

`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.

`execution_options`(***opt*)
Return a new `Engine` that will provide `Connection` objects with the given execution options.

The returned `Engine` remains related to the original `Engine` in that it shares the same connection pool and other state:

- The `Pool` used by the new `Engine` is the same instance. The `Engine.dispose()` method will replace the connection pool instance for the parent engine as well as this one.
- Event listeners are "cascaded" - meaning, the new `Engine` inherits the events of the parent, and new events can be associated with the new `Engine` individually.
- The logging configuration and logging_name is copied from the parent `Engine`.

The intent of the `Engine.execution_options()` method is to implement "sharding" schemes where multiple `Engine` objects refer to the same connection pool, but are differentiated by options that would be consumed by a custom event:

```
primary_engine = create_engine("mysql://")
shard1 = primary_engine.execution_options(shard_id="shard1")
shard2 = primary_engine.execution_options(shard_id="shard2")
```

Above, the `shard1` engine serves as a factory for `Connection` objects that will contain the execution option `shard_id=shard1`, and `shard2` will produce `Connection` objects that contain the execution option `shard_id=shard2`.

An event handler can consume the above execution option to perform a schema switch or other operation, given a connection. Below we emit a MySQL `use` statement to switch databases, at the same time keeping track of which database we've established using the `Connection.info` dictionary, which gives us a persistent storage space that follows the DBAPI connection:

```
from sqlalchemy import event
from sqlalchemy.engine import Engine

shards = {"default": "base", shard_1: "db1", "shard_2": "db2"}

@event.listens_for(Engine, "before_cursor_execute")
def _switch_shard(conn, cursor, stmt,
        params, context, executemany):
    shard_id = conn._execution_options.get('shard_id', "default")
    current_shard = conn.info.get("current_shard", None)

    if current_shard != shard_id:
        cursor.execute("use %s" % shards[shard_id])
        conn.info["current_shard"] = shard_id
```

New in version 0.8.

See also:

`Connection.execution_options()` - update execution options on a `Connection` object.

`Engine.update_execution_options()` - update the execution options for a given `Engine` in place.

has_table(*table_name*, *schema=None*)
Return True if the given backend has a table of the given name.

See also:

metadata_reflection_inspector - detailed schema inspection using the `Inspector` interface.

`quoted_name` - used to pass quoting information along with a schema identifier.

name
String name of the `Dialect` in use by this `Engine`.

raw_connection(*_connection=None*)
Return a "raw" DBAPI connection from the connection pool.

The returned object is a proxied version of the DBAPI connection object used by the underlying driver in use. The object will have all the same behavior as the real DBAPI connection, except that its `close()` method will result in the connection being returned to the pool, rather than being closed for real.

This method provides direct DBAPI connection access for special situations when the API provided by `Connection` is not needed. When a `Connection` object is already present, the DBAPI connection is available using the `Connection.connection` accessor.

See also:

dbapi_connections

run_callable(*callable_*, **args*, ***kwargs*)
Given a callable object or function, execute it, passing a `Connection` as the first argument.

The given *args and **kwargs are passed subsequent to the `Connection` argument.

This function, along with `Connection.run_callable()`, allows a function to be run with a `Connection` or `Engine` object without the need to know which one is being dealt with.

schema_for_object = <sqlalchemy.sql.schema._SchemaTranslateMap object>
Return the ".schema" attribute for an object.

Used for `Table`, `Sequence` and similar objects, and takes into account the `Connection.execution_options.schema_translate_map` parameter.

New in version 1.1.

See also:

schema_translating

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`.

transaction(*callable_*, **args*, ***kwargs*)
Execute the given function within a transaction boundary.

The function is passed a `Connection` newly procured from `Engine.contextual_connect()` as the first argument, followed by the given *args and **kwargs.

e.g.:

```
def do_something(conn, x, y):
    conn.execute("some statement", {'x':x, 'y':y})

engine.transaction(do_something, 5, 10)
```

The operations inside the function are all invoked within the context of a single `Transaction`. Upon success, the transaction is committed. If an exception is raised, the transaction is rolled back before propagating the exception.

Note: The `transaction()` method is superseded by the usage of the Python `with:` statement, which can be used with `Engine.begin()`:

```
with engine.begin() as conn:
    conn.execute("some statement", {'x':5, 'y':10})
```

See also:

`Engine.begin()` - engine-level transactional context

`Connection.transaction()` - connection-level version of `Engine.transaction()`

update_execution_options(***opt*)
Update the default execution_options dictionary of this `Engine`.

The given keys/values in **opt are added to the default execution options that will be used for all connections. The initial contents of this dictionary can be sent via the `execution_options` parameter to `create_engine()`.

See also:

`Connection.execution_options()`

`Engine.execution_options()`

class sqlalchemy.engine.ExceptionContext

Encapsulate information about an error condition in progress.

This object exists solely to be passed to the ConnectionEvents.handle_error() event, supporting an interface that can be extended without backwards-incompatibility.

New in version 0.9.7.

chained_exception = None

The exception that was returned by the previous handler in the exception chain, if any.

If present, this exception will be the one ultimately raised by SQLAlchemy unless a subsequent handler replaces it.

May be None.

connection = None

The Connection in use during the exception.

This member is present, except in the case of a failure when first connecting.

See also:

ExceptionContext.engine

cursor = None

The DBAPI cursor object.

May be None.

engine = None

The Engine in use during the exception.

This member should always be present, even in the case of a failure when first connecting.

New in version 1.0.0.

execution_context = None

The ExecutionContext corresponding to the execution operation in progress.

This is present for statement execution operations, but not for operations such as transaction begin/end. It also is not present when the exception was raised before the ExecutionContext could be constructed.

Note that the ExceptionContext.statement and ExceptionContext.parameters members may represent a different value than that of the ExecutionContext, potentially in the case where a ConnectionEvents.before_cursor_execute() event or similar modified the statement/parameters to be sent.

May be None.

invalidate_pool_on_disconnect = True

Represent whether all connections in the pool should be invalidated when a "disconnect" condition is in effect.

Setting this flag to False within the scope of the ConnectionEvents.handle_error() event will have the effect such that the full collection of connections in the pool will not be invalidated during a disconnect; only the current connection that is the subject of the error will actually be invalidated.

The purpose of this flag is for custom disconnect-handling schemes where the invalidation of other connections in the pool is to be performed based on other conditions, or even on a per-connection basis.

New in version 1.0.3.

is_disconnect = None

Represent whether the exception as occurred represents a "disconnect" condition.

This flag will always be True or False within the scope of the `ConnectionEvents.handle_error()` handler.

SQLAlchemy will defer to this flag in order to determine whether or not the connection should be invalidated subsequently. That is, by assigning to this flag, a "disconnect" event which then results in a connection and pool invalidation can be invoked or prevented by changing this flag.

original_exception = None
The exception object which was caught.

This member is always present.

parameters = None
Parameter collection that was emitted directly to the DBAPI.

May be None.

sqlalchemy_exception = None
The `sqlalchemy.exc.StatementError` which wraps the original, and will be raised if exception handling is not circumvented by the event.

May be None, as not all exception types are wrapped by SQLAlchemy. For DBAPI-level exceptions that subclass the dbapi's Error class, this field will always be present.

statement = None
String SQL statement that was emitted directly to the DBAPI.

May be None.

class sqlalchemy.engine.NestedTransaction(*connection*, *parent*)
Represent a 'nested', or SAVEPOINT transaction.

A new `NestedTransaction` object may be procured using the `Connection.begin_nested()` method.

The interface is the same as that of `Transaction`.

class sqlalchemy.engine.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.

_cursor_description()
May be overridden by subclasses.

_process_row
alias of `RowProxy`

_soft_close()
Soft close this `ResultProxy`.

This releases all DBAPI cursor resources, but leaves the ResultProxy "open" from a semantic perspective, meaning the fetchXXX() methods will continue to return empty results.

This method is called automatically when:

- all result rows are exhausted using the fetchXXX() methods.
- cursor.description is None.

This method is **not public**, but is documented in order to clarify the "autoclose" process used.

New in version 1.0.0.

See also:

`ResultProxy.close()`

`close()`
Close this ResultProxy.

This closes out the underlying DBAPI cursor corresonding to the statement execution, if one is still present. Note that the DBAPI cursor is automatically released when the `ResultProxy` exhausts all available rows. `ResultProxy.close()` is generally an optional method except in the case when discarding a `ResultProxy` that still has additional rows pending for fetch.

In the case of a result that is the product of connectionless execution, the underlying `Connection` object is also closed, which releases DBAPI connection resources.

After this method is called, it is no longer valid to call upon the fetch methods, which will raise a `ResourceClosedError` on subsequent use.

Changed in version 1.0.0: - the `ResultProxy.close()` method has been separated out from the process that releases the underlying DBAPI cursor resource. The "auto close" feature of the `Connection` now performs a so-called "soft close", which releases the underlying DBAPI cursor, but allows the `ResultProxy` to still behave as an open-but-exhausted result set; the actual `ResultProxy.close()` method is never called. It is still safe to discard a `ResultProxy` that has been fully exhausted without calling this method.

See also:

connections_toplevel

`ResultProxy._soft_close()`

`fetchall()`
Fetch all rows, just like DB-API `cursor.fetchall()`.

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

Subsequent calls to `ResultProxy.fetchall()` will return an empty list. After the `ResultProxy.close()` method is called, the method will raise `ResourceClosedError`.

Changed in version 1.0.0: - Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the `ResultProxy.close()` method.

`fetchmany(`*size=None*`)`
Fetch many rows, just like DB-API `cursor.fetchmany(size=cursor.arraysize)`.

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

Calls to `ResultProxy.fetchmany()` after all rows have been exhausted will return an empty list. After the `ResultProxy.close()` method is called, the method will raise `ResourceClosedError`.

Changed in version 1.0.0: - Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the `ResultProxy.close()` method.

`fetchone()`
Fetch one row, just like DB-API `cursor.fetchone()`.

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

Calls to `ResultProxy.fetchone()` after all rows have been exhausted will return `None`. After the `ResultProxy.close()` method is called, the method will raise `ResourceClosedError`.

Changed in version 1.0.0: - Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the `ResultProxy.close()` method.

`first()`
Fetch the first row and then close the result set unconditionally.

Returns None if no row is present.

After calling this method, the object is fully closed, e.g. the `ResultProxy.close()` method will have been called.

`inserted_primary_key`
Return the primary key for the row just inserted.

The return value is a list of scalar values corresponding to the list of primary key columns in the target table.

This only applies to single row `insert()` constructs which did not explicitly specify `Insert.returning()`.

Note that primary key columns which specify a server_default clause, or otherwise do not qualify as "autoincrement" columns (see the notes at `Column`), and were generated using the database-side default, will appear in this list as `None` unless the backend supports "returning" and the insert statement executed with the "implicit returning" enabled.

Raises `InvalidRequestError` if the executed statement is not a compiled expression construct or is not an insert() construct.

`is_insert`
True if this `ResultProxy` is the result of a executing an expression language compiled `expression.insert()` construct.

When True, this implies that the `inserted_primary_key` attribute is accessible, assuming the statement did not include a user defined "returning" construct.

`keys()`
Return the current set of string keys for rows.

`last_inserted_params()`
Return the collection of inserted parameters from this execution.

Raises `InvalidRequestError` if the executed statement is not a compiled expression construct or is not an insert() construct.

`last_updated_params()`
Return the collection of updated parameters from this execution.

Raises `InvalidRequestError` if the executed statement is not a compiled expression construct or is not an update() construct.

`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 when using insert() expression constructs; the `inserted_primary_key` attribute provides a tuple of primary key values for a newly inserted row, regardless of database backend.

`next()`
Implement the next() protocol.

New in version 1.2.

`postfetch_cols()`
Return `postfetch_cols()` from the underlying `ExecutionContext`.

See `ExecutionContext` for details.

Raises `InvalidRequestError` if the executed statement is not a compiled expression construct or is not an insert() or update() construct.

`prefetch_cols()`
Return `prefetch_cols()` from the underlying `ExecutionContext`.

See `ExecutionContext` for details.

Raises `InvalidRequestError` if the executed statement is not a compiled expression construct or is not an insert() or update() construct.

`returned_defaults`
Return the values of default columns that were fetched using the `ValuesBase.return_defaults()` feature.

The value is an instance of `RowProxy`, or `None` if `ValuesBase.return_defaults()` was not used or if the backend does not support RETURNING.

New in version 0.9.0.

See also:

`ValuesBase.return_defaults()`

`returns_rows`
True if this `ResultProxy` returns rows.

I.e. if it is legal to call the methods `fetchone()`, `fetchmany()` `fetchall()`.

`rowcount`
Return the 'rowcount' for this result.

The 'rowcount' reports the number of rows *matched* by the WHERE criterion of an UPDATE or DELETE statement.

Note: Notes regarding `ResultProxy.rowcount`:

- This attribute returns the number of rows *matched*, which is not necessarily the same as the number of rows that were actually *modified* - an UPDATE statement, for example, may have no net change on a given row if the SET values given are the same as those present in the row already. Such a row would be matched but not modified. On backends that feature both styles, such as MySQL, rowcount is configured by default to return the match count in all cases.
- `ResultProxy.rowcount` is *only* useful in conjunction with an UPDATE or DELETE statement. Contrary to what the Python DBAPI says, it does *not* return the number of rows available from the results of a SELECT statement as DBAPIs cannot support this functionality when rows are unbuffered.
- `ResultProxy.rowcount` may not be fully implemented by all dialects. In particular, most DBAPIs do not support an aggregate rowcount result from an executemany call. The `ResultProxy.supports_sane_rowcount()` and `ResultProxy.supports_sane_multi_rowcount()` methods will report from the dialect if each usage is known to be supported.
- Statements that use RETURNING may not return a correct rowcount.

---

scalar()
Fetch the first column of the first row, and close the result set.

Returns None if no row is present.

After calling this method, the object is fully closed, e.g. the `ResultProxy.close()` method will have been called.

supports_sane_multi_rowcount()
Return `supports_sane_multi_rowcount` from the dialect.

See `ResultProxy.rowcount` for background.

supports_sane_rowcount()
Return `supports_sane_rowcount` from the dialect.

See `ResultProxy.rowcount` for background.

class sqlalchemy.engine.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.

class sqlalchemy.engine.Transaction(*connection*, *parent*)
Represent a database transaction in progress.

The `Transaction` object is procured by calling the `begin()` method of `Connection`:

```
from sqlalchemy import create_engine
engine = create_engine("postgresql://scott:tiger@localhost/test")
connection = engine.connect()
trans = connection.begin()
connection.execute("insert into x (a, b) values (1, 2)")
trans.commit()
```

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:

```
with connection.begin():
    connection.execute("insert into x (a, b) values (1, 2)")
```

The Transaction object is **not** threadsafe.

See also: `Connection.begin()`, `Connection.begin_twophase()`, `Connection.begin_nested()`.

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.

class sqlalchemy.engine.TwoPhaseTransaction(*connection*, *xid*)
Represent a two-phase transaction.

A new TwoPhaseTransaction object may be procured using the Connection.begin_twophase() method.

The interface is the same as that of Transaction with the addition of the prepare() method.

prepare()
Prepare this TwoPhaseTransaction.

After a PREPARE, the transaction can be committed.

3.5.3 Connection Pooling

A connection pool is a standard technique used to maintain long running connections in memory for efficient re-use, as well as to provide management for the total number of connections an application might use simultaneously.

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.

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 only to learn how to 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, the SingletonThreadPool or NullPool are selected by the dialect to provide greater compatibility with SQLite's threading and locking 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.

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)
```

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 the `PoolEvents` event hooks, so that the usual URL argument to `create_engine()` is still usable. `creator` is there as a last resort for when a DBAPI has some form of `connect` that is not at all supported by SQLAlchemy.

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, it is 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).

The `close()` step also performs the important step of calling the `rollback()` method of the DBAPI connection. This is so that any existing transaction on the connection is removed, not only ensuring that no existing state remains on next usage, but also so that table and row locks are released as well as that any isolated data snapshots are removed. This behavior can be disabled using the `reset_on_return` option of `Pool`.

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)
```

Pool Events

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 `PoolEvents` for details.

Dealing with Disconnects

The connection pool has the ability to refresh individual connections as well as its entire set of connections, setting the previously pooled connections as "invalid". A common use case is allow the connection pool to gracefully recover when the database server has been restarted, and all previously established connections are no longer functional. There are two approaches to this.

Disconnect Handling - Pessimistic

The pessimistic approach refers to emitting a test statement on the SQL connection at the start of each connection pool checkout, to test that the database connection is still viable. Typically, this is a simple statement like "SELECT 1", but may also make use of some DBAPI-specific method to test the connection for liveness.

The approach adds a small bit of overhead to the connection checkout process, however is otherwise the most simple and reliable approach to completely eliminating database errors due to stale pooled connections. The calling application does not need to be concerned about organizing operations to be able to recover from stale connections checked out from the pool.

It is critical to note that the pre-ping approach **does not accommodate for connections dropped in the middle of transactions or other SQL operations**. If the database becomes unavailable while a transaction is in progress, the transaction will be lost and the database error will be raised. While the `Connection` object will detect a "disconnect" situation and recycle the connection as well as invalidate the rest of the connection pool when this condition occurs, the individual operation where the exception was raised will be lost, and it's up to the application to either abandon the operation, or retry the whole transaction again.

Pessimistic testing of connections upon checkout is achievable by using the `Pool.pre_ping` argument, available from `create_engine()` via the `create_engine.pool_pre_ping` argument:

```
engine = create_engine("mysql+pymysql://user:pw@host/db", pool_pre_ping=True)
```

The "pre ping" feature will normally emit SQL equivalent to "SELECT 1" each time a connection is checked out from the pool; if an error is raised that is detected as a "disconnect" situation, the connection will be immediately recycled, and all other pooled connections older than the current time are invalidated, so that the next time they are checked out, they will also be recycled before use.

If the database is still not available when "pre ping" runs, then the initial connect will fail and the error for failure to connect will be propagated normally. In the uncommon situation that the database is available for connections, but is not able to respond to a "ping", the "pre_ping" will try up to three times before giving up, propagating the database error last received.

Note: the "SELECT 1" emitted by "pre-ping" is invoked within the scope of the connection pool / dialect, using a very short codepath for minimal Python latency. As such, this statement is **not logged in the SQL echo output**, and will not show up in SQLAlchemy's engine logging.

New in version 1.2: Added "pre-ping" capability to the `Pool` class.

Custom / Legacy Pessimistic Ping

Before `create_engine.pool_pre_ping` was added, the "pre-ping" approach historically has been performed manually using the `ConnectionEvents.engine_connect()` engine event. The most common recipe for this is below, for reference purposes in case an application is already using such a recipe, or special behaviors are needed:

```
from sqlalchemy import exc
from sqlalchemy import event
from sqlalchemy import select

some_engine = create_engine(...)

@event.listens_for(some_engine, "engine_connect")
def ping_connection(connection, branch):
    if branch:
        # "branch" refers to a sub-connection of a connection,
        # we don't want to bother pinging on these.
        return

    # turn off "close with result".  This flag is only used with
    # "connectionless" execution, otherwise will be False in any case
    save_should_close_with_result = connection.should_close_with_result
    connection.should_close_with_result = False

    try:
        # run a SELECT 1.   use a core select() so that
        # the SELECT of a scalar value without a table is
        # appropriately formatted for the backend
        connection.scalar(select([1]))
    except exc.DBAPIError as err:
        # catch SQLAlchemy's DBAPIError, which is a wrapper
        # for the DBAPI's exception.  It includes a .connection_invalidated
        # attribute which specifies if this connection is a "disconnect"
        # condition, which is based on inspection of the original exception
        # by the dialect in use.
        if err.connection_invalidated:
            # run the same SELECT again - the connection will re-validate
            # itself and establish a new connection.  The disconnect detection
            # here also causes the whole connection pool to be invalidated
            # so that all stale connections are discarded.
            connection.scalar(select([1]))
        else:
            raise
    finally:
        # restore "close with result"
        connection.should_close_with_result = save_should_close_with_result
```

The above recipe has the advantage that we are making use of SQLAlchemy's facilities for detecting those DBAPI exceptions that are known to indicate a "disconnect" situation, as well as the `Engine` object's ability to correctly invalidate the current connection pool when this condition occurs and allowing the current `Connection` to re-validate onto a new DBAPI connection.

Disconnect Handling - Optimistic

When pessimistic handling is not employed, as well as when the database is shutdown and/or restarted in the middle of a connection's period of use within a transaction, the other approach to dealing with stale / closed connections is to let SQLAlchemy handle disconnects as they occur, at which point all connections in the pool are invalidated, meaning they are assumed to be stale and will be refreshed upon next checkout. This behavior assumes the `Pool` is used in conjunction with a `Engine`. The `Engine` has logic which can detect disconnection events and refresh the pool automatically.

When the `Connection` attempts to use a DBAPI connection, and an exception is raised that corresponds to a "disconnect" event, the connection is invalidated. The `Connection` then calls the `Pool.recreate()` method, effectively invalidating all connections not currently checked out so that they are replaced with new ones upon next checkout. This flow is illustrated by the code example below:

```
from sqlalchemy import create_engine, exc
e = create_engine(...)
c = e.connect()

try:
    # suppose the database has been restarted.
    c.execute("SELECT * FROM table")
    c.close()
except exc.DBAPIError, e:
    # an exception is raised, Connection is invalidated.
    if e.connection_invalidated:
        print("Connection was invalidated!")

# after the invalidate event, a new connection
# starts with a new Pool
c = e.connect()
c.execute("SELECT * FROM table")
```

The above example illustrates that no special intervention is needed to refresh the pool, which continues normally after a disconnection event is detected. However, one database exception is raised, per each connection that is in use while the database unavailability event occurred. In a typical web application using an ORM Session, the above condition would correspond to a single request failing with a 500 error, then the web application continuing normally beyond that. Hence the approach is "optimistic" in that frequent database restarts are not anticipated.

Setting Pool Recycle

An additional setting that can augment the "optimistic" approach is to set the pool recycle parameter. This parameter prevents the pool from using a particular connection that has passed a certain age, and is appropriate for database backends such as MySQL that automatically close connections that have been stale after a particular period of time:

```
from sqlalchemy import create_engine
e = create_engine("mysql://scott:tiger@localhost/test", pool_recycle=3600)
```

Above, any DBAPI connection that has been open for more than one hour will be invalidated and replaced, upon next checkout. Note that the invalidation **only** occurs during checkout - not on any connections that are held in a checked out state. `pool_recycle` is a function of the `Pool` itself, independent of whether or not an `Engine` is in use.

More on Invalidation

The `Pool` provides "connection invalidation" services which allow both explicit invalidation of a connection as well as automatic invalidation in response to conditions that are determined to render a

connection unusable.

"Invalidation" means that a particular DBAPI connection is removed from the pool and discarded. The `.close()` method is called on this connection if it is not clear that the connection itself might not be closed, however if this method fails, the exception is logged but the operation still proceeds.

When using a `Engine`, the `Connection.invalidate()` method is the usual entrypoint to explicit invalidation. Other conditions by which a DBAPI connection might be invalidated include:

- a DBAPI exception such as `OperationalError`, raised when a method like `connection.execute()` is called, is detected as indicating a so-called "disconnect" condition. As the Python DBAPI provides no standard system for determining the nature of an exception, all SQLAlchemy dialects include a system called `is_disconnect()` which will examine the contents of an exception object, including the string message and any potential error codes included with it, in order to determine if this exception indicates that the connection is no longer usable. If this is the case, the `_ConnectionFairy.invalidate()` method is called and the DBAPI connection is then discarded.
- When the connection is returned to the pool, and calling the `connection.rollback()` or `connection.commit()` methods, as dictated by the pool's "reset on return" behavior, throws an exception. A final attempt at calling `.close()` on the connection will be made, and it is then discarded.
- When a listener implementing `PoolEvents.checkout()` raises the `DisconnectionError` exception, indicating that the connection won't be usable and a new connection attempt needs to be made.

All invalidations which occur will invoke the `PoolEvents.invalidate()` event.

Using Connection Pools with Multiprocessing

It's critical that when using a connection pool, and by extension when using an `Engine` created via `create_engine()`, that the pooled connections **are not shared to a forked process**. TCP connections are represented as file descriptors, which usually work across process boundaries, meaning this will cause concurrent access to the file descriptor on behalf of two or more entirely independent Python interpreter states.

There are two approaches to dealing with this.

The first is, either create a new `Engine` within the child process, or upon an existing `Engine`, call `Engine.dispose()` before the child process uses any connections. This will remove all existing connections from the pool so that it makes all new ones. Below is a simple version using `multiprocessing.Process`, but this idea should be adapted to the style of forking in use:

```
eng = create_engine("...")

def run_in_process():
  eng.dispose()

  with eng.connect() as conn:
      conn.execute("...")

p = Process(target=run_in_process)
```

The next approach is to instrument the `Pool` itself with events so that connections are automatically invalidated in the subprocess. This is a little more magical but probably more foolproof:

```
from sqlalchemy import event
from sqlalchemy import exc
import os

eng = create_engine("...")

@event.listens_for(engine, "connect")
def connect(dbapi_connection, connection_record):
```

```
    connection_record.info['pid'] = os.getpid()

@event.listens_for(engine, "checkout")
def checkout(dbapi_connection, connection_record, connection_proxy):
    pid = os.getpid()
    if connection_record.info['pid'] != pid:
        connection_record.connection = connection_proxy.connection = None
        raise exc.DisconnectionError(
                "Connection record belongs to pid %s, "
                "attempting to check out in pid %s" %
                (connection_record.info['pid'], pid)
        )
```

Above, we use an approach similar to that described in pool_disconnects_pessimistic to treat a DBAPI connection that originated in a different parent process as an "invalid" connection, coercing the pool to recycle the connection record to make a new connection.

API Documentation - Available Pool Implementations

class sqlalchemy.pool.**Pool**(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*, *events=None*, *dialect=None*, *pre_ping=False*, *_dispatch=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*, *events=None*, *dialect=None*, *pre_ping=False*, *_dispatch=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 `Pool.unique_connection()` method is provided to return a consistently unique connection to bypass this behavior when the flag is set.

  Warning: The `Pool.use_threadlocal` flag **does not affect the behavior** of `Engine.connect()`. `Engine.connect()` makes use of the `Pool.unique_connection()` method which **does not use**

thread local context. To produce a `Connection` which refers to the `Pool.connect()` method, use `Engine.contextual_connect()`.

Note that other SQLAlchemy connectivity systems such as `Engine.execute()` as well as the orm `Session` make use of `Engine.contextual_connect()` internally, so these functions are compatible with the `Pool.use_threadlocal` setting.

See also:

threadlocal_strategy - contains detail on the "threadlocal" engine strategy, which provides a more comprehensive approach to "threadlocal" connectivity for the specific use case of using `Engine` and `Connection` objects directly.

- **`reset_on_return`** – Determine steps to take on connections as they are returned to the pool. reset_on_return can have any of these values:
  - `"rollback"` - call rollback() on the connection, to release locks and transaction resources. This is the default value. The vast majority of use cases should leave this value set.
  - `True` - same as 'rollback', this is here for backwards compatibility.
  - `"commit"` - call commit() on the connection, to release locks and transaction resources. A commit here may be desirable for databases that cache query plans if a commit is emitted, such as Microsoft SQL Server. However, this value is more dangerous than 'rollback' because any data changes present on the transaction are committed unconditionally.
  - `None` - don't do anything on the connection. This setting should only be made on a database that has no transaction support at all, namely MySQL MyISAM. By not doing anything, performance can be improved. This setting should **never be selected** for a database that supports transactions, as it will lead to deadlocks and stale state.
  - `"none"` - same as `None`

    New in version 0.9.10.
  - `False` - same as None, this is here for backwards compatibility.

  Changed in version 0.7.6: `Pool.reset_on_return` accepts `"rollback"` and `"commit"` arguments.
- **`events`** – a list of 2-tuples, each of the form `(callable, target)` which will be passed to `event.listen()` upon construction. Provided here so that event listeners can be assigned via `create_engine()` before dialect-level listeners are applied.
- **`listeners`** – Deprecated. 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. This has been superseded by `listen()`.
- **`dialect`** – a `Dialect` that will handle the job of calling rollback(), close(), or commit() on DBAPI connections. If omitted, a built-in "stub" dialect is used. Applications that make use of `create_engine()` should not use this parameter as it is handled by the engine creation strategy.

  New in version 1.1: - `dialect` is now a public parameter to the `Pool`.

- **pre_ping** – if True, the pool will emit a "ping" (typically "SELECT 1", but is dialect-specific) on the connection upon checkout, to test if the connection is alive or not. If not, the connection is transparently re-connected and upon success, all other pooled connections established prior to that timestamp are invalidated. Requires that a dialect is passed as well to interpret the disconnection error.

  New in version 1.2.

`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, as it only affects connections that are idle in the pool.

See also the `Pool.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 conjunction with `dispose()` to close out an entire `Pool` and create a new one in its place.

`unique_connection()`
Produce a DBAPI connection that is not referenced by any thread-local context.

This method is equivalent to `Pool.connect()` when the `Pool.use_threadlocal` flag is not set to True. When `Pool.use_threadlocal` is True, the `Pool.unique_connection()` method provides a means of bypassing the threadlocal context.

`class sqlalchemy.pool.QueuePool(`*creator*, *pool_size=5*, *max_overflow=10*, *timeout=30*, ***kw*`)`
A `Pool` that imposes a limit on the number of open connections.

`QueuePool` is the default pooling implementation used for all `Engine` objects, unless the SQLite dialect is in use.

`__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, same as that of `Pool.creator`.
- **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.
- `**kw` – Other keyword arguments including `Pool.recycle`, `Pool.echo`, `Pool.reset_on_return` and others are passed to the `Pool` constructor.

`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.

`unique_connection()`
Produce a DBAPI connection that is not referenced by any thread-local context.

This method is equivalent to `Pool.connect()` when the `Pool.use_threadlocal` flag is not set to True. When `Pool.use_threadlocal` is True, the `Pool.unique_connection()` method provides a means of bypassing the threadlocal context.

`class sqlalchemy.pool.SingletonThreadPool(`*creator*, *pool_size=5*, ***kw*`)`
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.

Warning: the `SingletonThreadPool` will call `.close()` on arbitrary connections that exist beyond the size setting of `pool_size`, e.g. if more unique **thread identities** than what `pool_size` states are used. This cleanup is non-deterministic and not sensitive to whether or not the connections linked to those thread identities are currently in use.

`SingletonThreadPool` may be improved in a future release, however in its current status it is generally used only for test scenarios using a SQLite `:memory:` database and is not recommended for production use.

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.
`SingletonThreadPool` is used by the SQLite dialect automatically when a memory-based database is used. See *SQLite*.

`__init__(`*creator*, *pool_size=5*, ***kw*`)`

`class sqlalchemy.pool.AssertionPool(`**args*, ***kw*`)`
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.

Changed in version 0.7: `AssertionPool` also logs a traceback of where the original connection was checked out, and reports this in the assertion error raised.

`class sqlalchemy.pool.NullPool(`*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*, *events=None*, *dialect=None*, *pre_ping=False*, *_dispatch=None*`)`
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.

Changed in version 0.7: `NullPool` is used by the SQlite dialect automatically when a file-based database is used. See *SQLite*.

`class sqlalchemy.pool.StaticPool(`*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*, *events=None*, *dialect=None*, *pre_ping=False*, *_dispatch=None*`)`

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.

`class sqlalchemy.pool._ConnectionFairy(`*dbapi_connection*, *connection_record*, *echo*`)`

Proxies a DBAPI connection and provides return-on-dereference support.

This is an internal object used by the `Pool` implementation to provide context management to a DBAPI connection delivered by that `Pool`.

The name "fairy" is inspired by the fact that the `_ConnectionFairy` object's lifespan is transitory, as it lasts only for the length of a specific DBAPI connection being checked out from the pool, and additionally that as a transparent proxy, it is mostly invisible.

See also:

`_ConnectionRecord`

`_connection_record = None`

A reference to the `_ConnectionRecord` object associated with the DBAPI connection.

This is currently an internal accessor which is subject to change.

`connection = None`

A reference to the actual DBAPI connection being tracked.

`cursor(`**args*, ***kwargs*`)`

Return a new DBAPI cursor for the underlying connection.

This method is a proxy for the `connection.cursor()` DBAPI method.

`detach()`

Separate this connection from its Pool.

This means that the connection will no longer be returned to the pool when closed, and will instead be literally closed. The containing ConnectionRecord is separated from the DB-API connection, and will create a new connection when next used.

Note that any overall connection limiting constraints imposed by a Pool implementation may be violated after a detach, as the detached connection is removed from the pool's knowledge and control.

`info`

Info dictionary associated with the underlying DBAPI connection referred to by this `ConnectionFairy`, allowing user-defined data to be associated with the connection.

The data here will follow along with the DBAPI connection including after it is returned to the connection pool and used again in subsequent instances of `_ConnectionFairy`. It is shared with the `_ConnectionRecord.info` and `Connection.info` accessors.

The dictionary associated with a particular DBAPI connection is discarded when the connection itself is discarded.

`invalidate(`*e=None*, *soft=False*`)`

Mark this connection as invalidated.

This method can be called directly, and is also called as a result of the `Connection.invalidate()` method. When invoked, the DBAPI connection is immediately closed and

discarded from further use by the pool. The invalidation mechanism proceeds via the `_ConnectionRecord.invalidate()` internal method.

Parameters

- **e** – an exception object indicating a reason for the invalidation.
- **soft** – if True, the connection isn't closed; instead, this connection will be recycled on next checkout.

  New in version 1.0.3.

See also:

pool_connection_invalidation

`is_valid`
Return True if this `_ConnectionFairy` still refers to an active DBAPI connection.

`record_info`
Info dictionary associated with the `_ConnectionRecord container referred to by this :class:`.ConnectionFairy`.

Unlike the `_ConnectionFairy.info` dictionary, the lifespan of this dictionary is persistent across connections that are disconnected and/or invalidated within the lifespan of a `_ConnectionRecord`.

New in version 1.1.

`class sqlalchemy.pool._ConnectionRecord`(*pool*, *connect=True*)
Internal object which maintains an individual DBAPI connection referenced by a `Pool`.

The `_ConnectionRecord` object always exists for any particular DBAPI connection whether or not that DBAPI connection has been "checked out". This is in contrast to the `_ConnectionFairy` which is only a public facade to the DBAPI connection while it is checked out.

A `_ConnectionRecord` may exist for a span longer than that of a single DBAPI connection. For example, if the `_ConnectionRecord.invalidate()` method is called, the DBAPI connection associated with this `_ConnectionRecord` will be discarded, but the `_ConnectionRecord` may be used again, in which case a new DBAPI connection is produced when the `Pool` next uses this record.

The `_ConnectionRecord` is delivered along with connection pool events, including `PoolEvents.connect()` and `PoolEvents.checkout()`, however `_ConnectionRecord` still remains an internal object whose API and internals may change.

See also:

`_ConnectionFairy`

`connection = None`
A reference to the actual DBAPI connection being tracked.

May be `None` if this `_ConnectionRecord` has been marked as invalidated; a new DBAPI connection may replace it if the owning pool calls upon this `_ConnectionRecord` to reconnect.

`info`
The `.info` dictionary associated with the DBAPI connection.

This dictionary is shared among the `_ConnectionFairy.info` and `Connection.info` accessors.

Note: The lifespan of this dictionary is linked to the DBAPI connection itself, meaning that it is **discarded** each time the DBAPI connection is closed and/or invalidated. The `_ConnectionRecord.record_info` dictionary remains persistent throughout the lifespan of the `_ConnectionRecord` container.

`invalidate`(*e=None*, *soft=False*)
Invalidate the DBAPI connection held by this `_ConnectionRecord`.

This method is called for all connection invalidations, including when the `_ConnectionFairy.invalidate()` or `Connection.invalidate()` methods are called, as well as when any so-called "automatic invalidation" condition occurs.

Parameters

- **e** – an exception object indicating a reason for the invalidation.
- **soft** – if True, the connection isn't closed; instead, this connection will be recycled on next checkout.

  New in version 1.0.3.

See also:

pool_connection_invalidation

`record_info`
An "info' dictionary associated with the connection record itself.

Unlike the `_ConnectionRecord.info` dictionary, which is linked to the lifespan of the DBAPI connection, this dictionary is linked to the lifespan of the `_ConnectionRecord` container itself and will remain persisent throughout the life of the `_ConnectionRecord`.

New in version 1.1.

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.

3.5.4 Core Events

This section describes the event interfaces provided in SQLAlchemy Core. For an introduction to the event listening API, see event_toplevel. ORM events are described in orm_event_toplevel.

class sqlalchemy.event.base.**Events**
Define event listening functions for a particular target type.

Connection Pool Events

class sqlalchemy.events.**PoolEvents**
Available events for Pool.

The methods here define the name of an event as well as the names of members that are passed to listener functions.

e.g.:

```
from sqlalchemy import event

def my_on_checkout(dbapi_conn, connection_rec, connection_proxy):
    "handle an on checkout event"

event.listen(Pool, 'checkout', my_on_checkout)
```

In addition to accepting the Pool class and Pool instances, PoolEvents also accepts Engine objects and the Engine class as targets, which will be resolved to the .pool attribute of the given engine or the Pool class:

```
engine = create_engine("postgresql://scott:tiger@localhost/test")

# will associate with engine.pool
event.listen(engine, 'checkout', my_on_checkout)
```

checkin(*dbapi_connection*, *connection_record*)
Called when a connection returns to the pool.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'checkin')
```

```
def receive_checkin(dbapi_connection, connection_record):
    "listen for the 'checkin' event"

    # ... (event handling logic) ...
```

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.)

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the `_ConnectionRecord` managing the DBAPI connection.

`checkout`(*dbapi_connection*, *connection_record*, *connection_proxy*)

Called when a connection is retrieved from the Pool.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'checkout')
def receive_checkout(dbapi_connection, connection_record, connection_proxy):
    "listen for the 'checkout' event"

    # ... (event handling logic) ...
```

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the `_ConnectionRecord` managing the DBAPI connection.
- **connection_proxy** – the `_ConnectionFairy` object which will proxy the public interface of the DBAPI connection for the lifespan of the checkout.

If you raise a `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.

See also:

`ConnectionEvents.engine_connect()` - a similar event which occurs upon creation of a new `Connection`.

`close`(*dbapi_connection*, *connection_record*)

Called when a DBAPI connection is closed.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'close')
def receive_close(dbapi_connection, connection_record):
    "listen for the 'close' event"

    # ... (event handling logic) ...
```

The event is emitted before the close occurs.

The close of a connection can fail; typically this is because the connection is already closed. If the close operation fails, the connection is discarded.

The `close()` event corresponds to a connection that's still associated with the pool. To intercept close events for detached connections use `close_detached()`.

New in version 1.1.

`close_detached`(*dbapi_connection*)
Called when a detached DBAPI connection is closed.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'close_detached')
def receive_close_detached(dbapi_connection):
    "listen for the 'close_detached' event"

    # ... (event handling logic) ...
```

The event is emitted before the close occurs.

The close of a connection can fail; typically this is because the connection is already closed. If the close operation fails, the connection is discarded.

New in version 1.1.

`connect`(*dbapi_connection*, *connection_record*)
Called at the moment a particular DBAPI connection is first created for a given `Pool`.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'connect')
def receive_connect(dbapi_connection, connection_record):
    "listen for the 'connect' event"

    # ... (event handling logic) ...
```

This event allows one to capture the point directly after which the DBAPI module-level `.connect()` method has been used in order to produce a new DBAPI connection.

Parameters

- **`dbapi_connection`** – a DBAPI connection.
- **`connection_record`** – the `_ConnectionRecord` managing the DBAPI connection.

`detach`(*dbapi_connection*, *connection_record*)
Called when a DBAPI connection is "detached" from a pool.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'detach')
def receive_detach(dbapi_connection, connection_record):
    "listen for the 'detach' event"

    # ... (event handling logic) ...
```

This event is emitted after the detach occurs. The connection is no longer associated with the given connection record.

New in version 1.1.

first_connect(*dbapi_connection*, *connection_record*)

Called exactly once for the first time a DBAPI connection is checked out from a particular `Pool`.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'first_connect')
def receive_first_connect(dbapi_connection, connection_record):
    "listen for the 'first_connect' event"

    # ... (event handling logic) ...
```

The rationale for `PoolEvents.first_connect()` is to determine information about a particular series of database connections based on the settings used for all connections. Since a particular `Pool` refers to a single "creator" function (which in terms of a `Engine` refers to the URL and connection options used), it is typically valid to make observations about a single connection that can be safely assumed to be valid about all subsequent connections, such as the database version, the server and client encoding settings, collation settings, and many others.

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the `_ConnectionRecord` managing the DBAPI connection.

invalidate(*dbapi_connection*, *connection_record*, *exception*)

Called when a DBAPI connection is to be "invalidated".

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'invalidate')
def receive_invalidate(dbapi_connection, connection_record, exception):
    "listen for the 'invalidate' event"

    # ... (event handling logic) ...
```

This event is called any time the `_ConnectionRecord.invalidate()` method is invoked, either from API usage or via "auto-invalidation", without the `soft` flag.

The event occurs before a final attempt to call `.close()` on the connection occurs.

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the `_ConnectionRecord` managing the DBAPI connection.
- **exception** – the exception object corresponding to the reason for this invalidation, if any. May be `None`.

New in version 0.9.2: Added support for connection invalidation listening.

See also:

pool_connection_invalidation

`reset`(*dbapi_connection*, *connection_record*)
Called before the "reset" action occurs for a pooled connection.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'reset')
def receive_reset(dbapi_connection, connection_record):
    "listen for the 'reset' event"

    # ... (event handling logic) ...
```

This event represents when the `rollback()` method is called on the DBAPI connection before it is returned to the pool. The behavior of "reset" can be controlled, including disabled, using the `reset_on_return` pool argument.

The `PoolEvents.reset()` event is usually followed by the `PoolEvents.checkin()` event is called, except in those cases where the connection is discarded immediately after reset.

Parameters

- **`dbapi_connection`** – a DBAPI connection.
- **`connection_record`** – the `_ConnectionRecord` managing the DBAPI connection.

New in version 0.8.

See also:

`ConnectionEvents.rollback()`

`ConnectionEvents.commit()`

`soft_invalidate`(*dbapi_connection*, *connection_record*, *exception*)
Called when a DBAPI connection is to be "soft invalidated".

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'soft_invalidate')
def receive_soft_invalidate(dbapi_connection, connection_record, exception):
    "listen for the 'soft_invalidate' event"

    # ... (event handling logic) ...
```

This event is called any time the `_ConnectionRecord.invalidate()` method is invoked with the `soft` flag.

Soft invalidation refers to when the connection record that tracks this connection will force a reconnect after the current connection is checked in. It does not actively close the dbapi_connection at the point at which it is called.

New in version 1.0.3.

SQL Execution and Connection Events

class sqlalchemy.events.ConnectionEvents

Available events for Connectable, which includes Connection and Engine.

The methods here define the name of an event as well as the names of members that are passed to listener functions.

An event listener can be associated with any Connectable class or instance, such as an Engine, e.g.:

```
from sqlalchemy import event, create_engine

def before_cursor_execute(conn, cursor, statement, parameters, context,
                                                executemany):
    log.info("Received statement: %s", statement)

engine = create_engine('postgresql://scott:tiger@localhost/test')
event.listen(engine, "before_cursor_execute", before_cursor_execute)
```

or with a specific Connection:

```
with engine.begin() as conn:
    @event.listens_for(conn, 'before_cursor_execute')
    def before_cursor_execute(conn, cursor, statement, parameters,
                                    context, executemany):
        log.info("Received statement: %s", statement)
```

When the methods are called with a *statement* parameter, such as in after_cursor_execute(), before_cursor_execute() and dbapi_error(), the statement is the exact SQL string that was prepared for transmission to the DBAPI cursor in the connection's Dialect.

The before_execute() and before_cursor_execute() events can also be established with the retval=True flag, which allows modification of the statement and parameters to be sent to the database. The before_cursor_execute() event is particularly useful here to add ad-hoc string transformations, such as comments, to all executions:

```
from sqlalchemy.engine import Engine
from sqlalchemy import event

@event.listens_for(Engine, "before_cursor_execute", retval=True)
def comment_sql_calls(conn, cursor, statement, parameters,
                                    context, executemany):
    statement = statement + " -- some comment"
    return statement, parameters
```

Note: ConnectionEvents can be established on any combination of Engine, Connection, as well as instances of each of those classes. Events across all four scopes will fire off for a given instance of Connection. However, for performance reasons, the Connection object determines at instantiation time whether or not its parent Engine has event listeners established. Event listeners added to the Engine class or to an instance of Engine *after* the instantiation of a dependent Connection instance will usually *not* be available on that Connection instance. The newly added listeners will instead take effect for Connection instances created subsequent to those event listeners being established on the parent Engine class or instance.

Parameters **retval=False** – Applies to the before_execute() and before_cursor_execute() events only. When True, the user-defined event function must have a return value, which is a tuple of parameters that replace the given statement and parameters. See those methods for a description of specific return arguments.

Changed in version 0.8: `ConnectionEvents` can now be associated with any `Connectable` including `Connection`, in addition to the existing support for `Engine`.

after_cursor_execute(*conn*, *cursor*, *statement*, *parameters*, *context*, *executemany*)
Intercept low-level cursor execute() events after execution.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'after_cursor_execute')
def receive_after_cursor_execute(conn, cursor, statement, parameters, context,␣
↪executemany):
    "listen for the 'after_cursor_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'after_cursor_execute', named=True)
def receive_after_cursor_execute(**kw):
    "listen for the 'after_cursor_execute' event"
    conn = kw['conn']
    cursor = kw['cursor']

    # ... (event handling logic) ...
```

Parameters

- **conn** – `Connection` object
- **cursor** – DBAPI cursor object. Will have results pending if the statement was a SELECT, but these should not be consumed as they will be needed by the `ResultProxy`.
- **statement** – string SQL statement, as passed to the DBAPI
- **parameters** – Dictionary, tuple, or list of parameters being passed to the `execute()` or `executemany()` method of the DBAPI `cursor`. In some cases may be `None`.
- **context** – `ExecutionContext` object in use. May be `None`.
- **executemany** – boolean, if `True`, this is an `executemany()` call, if `False`, this is an `execute()` call.

after_execute(*conn*, *clauseelement*, *multiparams*, *params*, *result*)
Intercept high level execute() events after execute.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'after_execute')
def receive_after_execute(conn, clauseelement, multiparams, params, result):
    "listen for the 'after_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'after_execute', named=True)
def receive_after_execute(**kw):
    "listen for the 'after_execute' event"
    conn = kw['conn']
```

```
    clauseelement = kw['clauseelement']

    # ... (event handling logic) ...
```

Parameters

- **conn** – Connection object
- **clauseelement** – SQL expression construct, Compiled instance, or string statement passed to Connection.execute().
- **multiparams** – Multiple parameter sets, a list of dictionaries.
- **params** – Single parameter set, a single dictionary.
- **result** – ResultProxy generated by the execution.

before_cursor_execute(*conn*, *cursor*, *statement*, *parameters*, *context*, *executemany*)

Intercept low-level cursor execute() events before execution, receiving the string SQL statement and DBAPI-specific parameter list to be invoked against a cursor.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'before_cursor_execute')
def receive_before_cursor_execute(conn, cursor, statement, parameters, context,␣
↪executemany):
    "listen for the 'before_cursor_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'before_cursor_execute', named=True)
def receive_before_cursor_execute(**kw):
    "listen for the 'before_cursor_execute' event"
    conn = kw['conn']
    cursor = kw['cursor']

    # ... (event handling logic) ...
```

This event is a good choice for logging as well as late modifications to the SQL string. It's less ideal for parameter modifications except for those which are specific to a target backend.

This event can be optionally established with the retval=True flag. The statement and parameters arguments should be returned as a two-tuple in this case:

```
@event.listens_for(Engine, "before_cursor_execute", retval=True)
def before_cursor_execute(conn, cursor, statement,
                parameters, context, executemany):
    # do something with statement, parameters
    return statement, parameters
```

See the example at ConnectionEvents.

Parameters

- **conn** – Connection object
- **cursor** – DBAPI cursor object
- **statement** – string SQL statement, as to be passed to the DBAPI

- **parameters** – Dictionary, tuple, or list of parameters being passed to the `execute()` or `executemany()` method of the DBAPI `cursor`. In some cases may be `None`.
- **context** – `ExecutionContext` object in use. May be `None`.
- **executemany** – boolean, if `True`, this is an `executemany()` call, if `False`, this is an `execute()` call.

See also:

`before_execute()`

`after_cursor_execute()`

before_execute(*conn*, *clauseelement*, *multiparams*, *params*)

Intercept high level execute() events, receiving uncompiled SQL constructs and other objects prior to rendering into SQL.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'before_execute')
def receive_before_execute(conn, clauseelement, multiparams, params):
    "listen for the 'before_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'before_execute', named=True)
def receive_before_execute(**kw):
    "listen for the 'before_execute' event"
    conn = kw['conn']
    clauseelement = kw['clauseelement']

    # ... (event handling logic) ...
```

This event is good for debugging SQL compilation issues as well as early manipulation of the parameters being sent to the database, as the parameter lists will be in a consistent format here.

This event can be optionally established with the `retval=True` flag. The `clauseelement`, `multiparams`, and `params` arguments should be returned as a three-tuple in this case:

```
@event.listens_for(Engine, "before_execute", retval=True)
def before_execute(conn, clauseelement, multiparams, params):
    # do something with clauseelement, multiparams, params
    return clauseelement, multiparams, params
```

Parameters

- **conn** – `Connection` object
- **clauseelement** – SQL expression construct, `Compiled` instance, or string statement passed to `Connection.execute()`.
- **multiparams** – Multiple parameter sets, a list of dictionaries.
- **params** – Single parameter set, a single dictionary.

See also:

`before_cursor_execute()`

begin(*conn*)
Intercept begin() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'begin')
def receive_begin(conn):
    "listen for the 'begin' event"

    # ... (event handling logic) ...
```

Parameters **conn** – Connection object

begin_twophase(*conn*, *xid*)
Intercept begin_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'begin_twophase')
def receive_begin_twophase(conn, xid):
    "listen for the 'begin_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – Connection object
- **xid** – two-phase XID identifier

commit(*conn*)
Intercept commit() events, as initiated by a Transaction.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'commit')
def receive_commit(conn):
    "listen for the 'commit' event"

    # ... (event handling logic) ...
```

Note that the Pool may also "auto-commit" a DBAPI connection upon checkin, if the reset_on_return flag is set to the value 'commit'. To intercept this commit, use the PoolEvents.reset() hook.

Parameters **conn** – Connection object

commit_twophase(*conn*, *xid*, *is_prepared*)
Intercept commit_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
```

```
@event.listens_for(SomeEngine, 'commit_twophase')
def receive_commit_twophase(conn, xid, is_prepared):
    "listen for the 'commit_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – Connection object
- **xid** – two-phase XID identifier
- **is_prepared** – boolean, indicates if TwoPhaseTransaction.prepare() was called.

dbapi_error(*conn*, *cursor*, *statement*, *parameters*, *context*, *exception*)

Intercept a raw DBAPI error.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'dbapi_error')
def receive_dbapi_error(conn, cursor, statement, parameters, context, exception):
    "listen for the 'dbapi_error' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'dbapi_error', named=True)
def receive_dbapi_error(**kw):
    "listen for the 'dbapi_error' event"
    conn = kw['conn']
    cursor = kw['cursor']

    # ... (event handling logic) ...
```

This event is called with the DBAPI exception instance received from the DBAPI itself, *before* SQLAlchemy wraps the exception with it's own exception wrappers, and before any other operations are performed on the DBAPI cursor; the existing transaction remains in effect as well as any state on the cursor.

The use case here is to inject low-level exception handling into an Engine, typically for logging and debugging purposes.

Warning: Code should **not** modify any state or throw any exceptions here as this will interfere with SQLAlchemy's cleanup and error handling routines. For exception modification, please refer to the new ConnectionEvents.handle_error() event.

Subsequent to this hook, SQLAlchemy may attempt any number of operations on the connection/cursor, including closing the cursor, rolling back of the transaction in the case of connectionless execution, and disposing of the entire connection pool if a "disconnect" was detected. The exception is then wrapped in a SQLAlchemy DBAPI exception wrapper and re-thrown.

Parameters

- **conn** – Connection object
- **cursor** – DBAPI cursor object

- **statement** – string SQL statement, as passed to the DBAPI
- **parameters** – Dictionary, tuple, or list of parameters being passed to the `execute()` or `executemany()` method of the DBAPI `cursor`. In some cases may be `None`.
- **context** – `ExecutionContext` object in use. May be `None`.
- **exception** – The **unwrapped** exception emitted directly from the DBAPI. The class here is specific to the DBAPI module in use.

Deprecated since version 0.9.7: - replaced by `ConnectionEvents.handle_error()`

engine_connect(*conn*, *branch*)

Intercept the creation of a new `Connection`.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'engine_connect')
def receive_engine_connect(conn, branch):
    "listen for the 'engine_connect' event"

    # ... (event handling logic) ...
```

This event is called typically as the direct result of calling the `Engine.connect()` method.

It differs from the `PoolEvents.connect()` method, which refers to the actual connection to a database at the DBAPI level; a DBAPI connection may be pooled and reused for many operations. In contrast, this event refers only to the production of a higher level `Connection` wrapper around such a DBAPI connection.

It also differs from the `PoolEvents.checkout()` event in that it is specific to the `Connection` object, not the DBAPI connection that `PoolEvents.checkout()` deals with, although this DBAPI connection is available here via the `Connection.connection` attribute. But note there can in fact be multiple `PoolEvents.checkout()` events within the lifespan of a single `Connection` object, if that `Connection` is invalidated and re-established. There can also be multiple `Connection` objects generated for the same already-checked-out DBAPI connection, in the case that a "branch" of a `Connection` is produced.

Parameters

- **conn** – `Connection` object.
- **branch** – if True, this is a "branch" of an existing `Connection`. A branch is generated within the course of a statement execution to invoke supplemental statements, most typically to pre-execute a SELECT of a default value for the purposes of an INSERT statement.

New in version 0.9.0.

See also:

pool_disconnects_pessimistic - illustrates how to use `ConnectionEvents.engine_connect()` to transparently ensure pooled connections are connected to the database.

`PoolEvents.checkout()` the lower-level pool checkout event for an individual DBAPI connection

`ConnectionEvents.set_connection_execution_options()` - a copy of a `Connection` is also made when the `Connection.execution_options()` method is called.

engine_disposed(*engine*)

Intercept when the `Engine.dispose()` method is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'engine_disposed')
def receive_engine_disposed(engine):
    "listen for the 'engine_disposed' event"

    # ... (event handling logic) ...
```

The `Engine.dispose()` method instructs the engine to "dispose" of it's connection pool (e.g. `Pool`), and replaces it with a new one. Disposing of the old pool has the effect that existing checked-in connections are closed. The new pool does not establish any new connections until it is first used.

This event can be used to indicate that resources related to the `Engine` should also be cleaned up, keeping in mind that the `Engine` can still be used for new requests in which case it re-acquires connection resources.

New in version 1.0.5.

handle_error(*exception_context*)

Intercept all exceptions processed by the `Connection`.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'handle_error')
def receive_handle_error(exception_context):
    "listen for the 'handle_error' event"

    # ... (event handling logic) ...
```

This includes all exceptions emitted by the DBAPI as well as within SQLAlchemy's statement invocation process, including encoding errors and other statement validation errors. Other areas in which the event is invoked include transaction begin and end, result row fetching, cursor creation.

Note that `handle_error()` may support new kinds of exceptions and new calling scenarios at *any time*. Code which uses this event must expect new calling patterns to be present in minor releases.

To support the wide variety of members that correspond to an exception, as well as to allow extensibility of the event without backwards incompatibility, the sole argument received is an instance of `ExceptionContext`. This object contains data members representing detail about the exception.

Use cases supported by this hook include:

- read-only, low-level exception handling for logging and debugging purposes
- exception re-writing
- Establishing or disabling whether a connection or the owning connection pool is invalidated or expired in response to a specific exception.

The hook is called while the cursor from the failed operation (if any) is still open and accessible. Special cleanup operations can be called on this cursor; SQLAlchemy will attempt to close this cursor subsequent to this hook being invoked. If the connection is in "autocommit" mode, the transaction also remains open within the scope of this hook; the rollback of the per-statement transaction also occurs after the hook is called.

For the common case of detecting a "disconnect" situation which is not currently handled by the SQLAlchemy dialect, the `ExceptionContext.is_disconnect` flag can be set to True which will cause the exception to be considered as a disconnect situation, which typically results in the connection pool being invalidated:

```
@event.listens_for(Engine, "handle_error")
def handle_exception(context):
    if isinstance(context.original_exception, pyodbc.Error):
        for code in (
            '08S01', '01002', '08003',
            '08007', '08S02', '08001', 'HYT00', 'HY010'):

            if code in str(context.original_exception):
                context.is_disconnect = True
```

A handler function has two options for replacing the SQLAlchemy-constructed exception into one that is user defined. It can either raise this new exception directly, in which case all further event listeners are bypassed and the exception will be raised, after appropriate cleanup as taken place:

```
@event.listens_for(Engine, "handle_error")
def handle_exception(context):
    if isinstance(context.original_exception,
        psycopg2.OperationalError) and \
        "failed" in str(context.original_exception):
        raise MySpecialException("failed operation")
```

Warning: Because the `ConnectionEvents.handle_error()` event specifically provides for exceptions to be re-thrown as the ultimate exception raised by the failed statement, **stack traces will be misleading** if the user-defined event handler itself fails and throws an unexpected exception; the stack trace may not illustrate the actual code line that failed! It is advised to code carefully here and use logging and/or inline debugging if unexpected exceptions are occurring.

Alternatively, a "chained" style of event handling can be used, by configuring the handler with the `retval=True` modifier and returning the new exception instance from the function. In this case, event handling will continue onto the next handler. The "chained" exception is available using `ExceptionContext.chained_exception`:

```
@event.listens_for(Engine, "handle_error", retval=True)
def handle_exception(context):
    if context.chained_exception is not None and \
        "special" in context.chained_exception.message:
        return MySpecialException("failed",
            cause=context.chained_exception)
```

Handlers that return `None` may be used within the chain; when a handler returns `None`, the previous exception instance, if any, is maintained as the current exception that is passed onto the next handler.

When a custom exception is raised or returned, SQLAlchemy raises this new exception as-is, it is not wrapped by any SQLAlchemy object. If the exception is not a subclass of `sqlalchemy.exc.StatementError`, certain features may not be available; currently this includes the ORM's feature of adding a detail hint about "autoflush" to exceptions raised within the autoflush process.

Parameters `context` – an `ExceptionContext` object. See this class for details on all available members.

New in version 0.9.7: Added the `ConnectionEvents.handle_error()` hook.

Changed in version 1.1: The `handle_error()` event will now receive all exceptions that inherit from `BaseException`, including `SystemExit` and `KeyboardInterrupt`. The setting for `ExceptionContext.is_disconnect` is `True` in this case and the default for `ExceptionContext.invalidate_pool_on_disconnect` is `False`.

Changed in version 1.0.0: The `handle_error()` event is now invoked when an `Engine` fails during the initial call to `Engine.connect()`, as well as when a `Connection` object encounters an error during a reconnect operation.

Changed in version 1.0.0: The `handle_error()` event is not fired off when a dialect makes use of the `skip_user_error_events` execution option. This is used by dialects which intend to catch SQLAlchemy-specific exceptions within specific operations, such as when the MySQL dialect detects a table not present within the `has_table()` dialect method. Prior to 1.0.0, code which implements `handle_error()` needs to ensure that exceptions thrown in these scenarios are re-raised without modification.

`prepare_twophase`(*conn*, *xid*)

Intercept prepare_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'prepare_twophase')
def receive_prepare_twophase(conn, xid):
    "listen for the 'prepare_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – `Connection` object
- **xid** – two-phase XID identifier

`release_savepoint`(*conn*, *name*, *context*)

Intercept release_savepoint() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'release_savepoint')
def receive_release_savepoint(conn, name, context):
    "listen for the 'release_savepoint' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – `Connection` object
- **name** – specified name used for the savepoint.
- **context** – `ExecutionContext` in use. May be `None`.

`rollback`(*conn*)

Intercept rollback() events, as initiated by a `Transaction`.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'rollback')
def receive_rollback(conn):
    "listen for the 'rollback' event"

    # ... (event handling logic) ...
```

Note that the `Pool` also "auto-rolls back" a DBAPI connection upon checkin, if the `reset_on_return` flag is set to its default value of `'rollback'`. To intercept this rollback, use the `PoolEvents.reset()` hook.

Parameters `conn` – `Connection` object

See also:

`PoolEvents.reset()`

`rollback_savepoint`(*conn*, *name*, *context*)
Intercept rollback_savepoint() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'rollback_savepoint')
def receive_rollback_savepoint(conn, name, context):
    "listen for the 'rollback_savepoint' event"

    # ... (event handling logic) ...
```

Parameters

- `conn` – `Connection` object
- `name` – specified name used for the savepoint.
- `context` – `ExecutionContext` in use. May be `None`.

`rollback_twophase`(*conn*, *xid*, *is_prepared*)
Intercept rollback_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'rollback_twophase')
def receive_rollback_twophase(conn, xid, is_prepared):
    "listen for the 'rollback_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- `conn` – `Connection` object
- `xid` – two-phase XID identifier
- `is_prepared` – boolean, indicates if `TwoPhaseTransaction.prepare()` was called.

savepoint(*conn*, *name*)

Intercept savepoint() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'savepoint')
def receive_savepoint(conn, name):
    "listen for the 'savepoint' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – Connection object
- **name** – specified name used for the savepoint.

set_connection_execution_options(*conn*, *opts*)

Intercept when the Connection.execution_options() method is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'set_connection_execution_options')
def receive_set_connection_execution_options(conn, opts):
    "listen for the 'set_connection_execution_options' event"

    # ... (event handling logic) ...
```

This method is called after the new Connection has been produced, with the newly updated execution options collection, but before the Dialect has acted upon any of those new options.

Note that this method is not called when a new Connection is produced which is inheriting execution options from its parent Engine; to intercept this condition, use the ConnectionEvents.engine_connect() event.

Parameters

- **conn** – The newly copied Connection object
- **opts** – dictionary of options that were passed to the Connection.execution_options() method.

New in version 0.9.0.

See also:

ConnectionEvents.set_engine_execution_options() - event which is called when Engine.execution_options() is called.

set_engine_execution_options(*engine*, *opts*)

Intercept when the Engine.execution_options() method is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'set_engine_execution_options')
def receive_set_engine_execution_options(engine, opts):
    "listen for the 'set_engine_execution_options' event"
```

```
    # ... (event handling logic) ...
```

The `Engine.execution_options()` method produces a shallow copy of the `Engine` which stores the new options. That new `Engine` is passed here. A particular application of this method is to add a `ConnectionEvents.engine_connect()` event handler to the given `Engine` which will perform some per- `Connection` task specific to these execution options.

Parameters

- **conn** – The newly copied `Engine` object
- **opts** – dictionary of options that were passed to the `Connection.execution_options()` method.

New in version 0.9.0.

See also:

`ConnectionEvents.set_connection_execution_options()` - event which is called when `Connection.execution_options()` is called.

`class sqlalchemy.events.DialectEvents`

event interface for execution-replacement functions.

These events allow direct instrumentation and replacement of key dialect functions which interact with the DBAPI.

Note: `DialectEvents` hooks should be considered **semi-public** and experimental. These hooks are not for general use and are only for those situations where intricate re-statement of DBAPI mechanics must be injected onto an existing dialect. For general-use statement-interception events, please use the `ConnectionEvents` interface.

See also:

`ConnectionEvents.before_cursor_execute()`

`ConnectionEvents.before_execute()`

`ConnectionEvents.after_cursor_execute()`

`ConnectionEvents.after_execute()`

New in version 0.9.4.

`do_connect`(*dialect*, *conn_rec*, *cargs*, *cparams*)

Receive connection arguments before a connection is made.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'do_connect')
def receive_do_connect(dialect, conn_rec, cargs, cparams):
    "listen for the 'do_connect' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'do_connect', named=True)
def receive_do_connect(**kw):
    "listen for the 'do_connect' event"
    dialect = kw['dialect']
    conn_rec = kw['conn_rec']
```

```
    # ... (event handling logic) ...
```

Return a DBAPI connection to halt further events from invoking; the returned connection will be used.

Alternatively, the event can manipulate the cargs and/or cparams collections; cargs will always be a Python list that can be mutated in-place and cparams a Python dictionary. Return None to allow control to pass to the next event handler and ultimately to allow the dialect to connect normally, given the updated arguments.

New in version 1.0.3.

do_execute(*cursor*, *statement*, *parameters*, *context*)
Receive a cursor to have execute() called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'do_execute')
def receive_do_execute(cursor, statement, parameters, context):
    "listen for the 'do_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'do_execute', named=True)
def receive_do_execute(**kw):
    "listen for the 'do_execute' event"
    cursor = kw['cursor']
    statement = kw['statement']

    # ... (event handling logic) ...
```

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

do_execute_no_params(*cursor*, *statement*, *context*)
Receive a cursor to have execute() with no parameters called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'do_execute_no_params')
def receive_do_execute_no_params(cursor, statement, context):
    "listen for the 'do_execute_no_params' event"

    # ... (event handling logic) ...
```

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

do_executemany(*cursor*, *statement*, *parameters*, *context*)
Receive a cursor to have executemany() called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
```

```
@event.listens_for(SomeEngine, 'do_executemany')
def receive_do_executemany(cursor, statement, parameters, context):
    "listen for the 'do_executemany' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'do_executemany', named=True)
def receive_do_executemany(**kw):
    "listen for the 'do_executemany' event"
    cursor = kw['cursor']
    statement = kw['statement']

    # ... (event handling logic) ...
```

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

Schema Events

class sqlalchemy.events.**DDLEvents**

Define event listeners for schema objects, that is, `SchemaItem` and other `SchemaEventTarget` subclasses, including `MetaData`, `Table`, `Column`.

`MetaData` and `Table` support events specifically regarding when CREATE and DROP DDL is emitted to the database.

Attachment events are also provided to customize behavior whenever a child schema element is associated with a parent, such as, when a `Column` is associated with its `Table`, when a `ForeignKeyConstraint` is associated with a `Table`, etc.

Example using the `after_create` event:

```
from sqlalchemy import event
from sqlalchemy import Table, Column, Metadata, Integer

m = MetaData()
some_table = Table('some_table', m, Column('data', Integer))

def after_create(target, connection, **kw):
    connection.execute("ALTER TABLE %s SET name=foo_%s" %
                            (target.name, target.name))

event.listen(some_table, "after_create", after_create)
```

DDL events integrate closely with the `DDL` class and the `DDLElement` hierarchy of DDL clause constructs, which are themselves appropriate as listener callables:

```
from sqlalchemy import DDL
event.listen(
    some_table,
    "after_create",
    DDL("ALTER TABLE %(table)s SET name=foo_%(table)s")
)
```

The methods here define the name of an event as well as the names of members that are passed to listener functions.

For all `DDLEvent` events, the `propagate=True` keyword argument will ensure that a given event handler is propagated to copies of the object, which are made when using the `Table.tometadata()` method:

```
from sqlalchemy import DDL
event.listen(
    some_table,
    "after_create",
    DDL("ALTER TABLE %(table)s SET name=foo_%(table)s"),
    propagate=True
)

new_table = some_table.tometadata(new_metadata)
```

The above DDL object will also be associated with the Table object represented by new_table.

See also:

event_toplevel

DDLElement

DDL

schema_ddl_sequences

after_create(*target*, *connection*, ***kw*)

Called after CREATE statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'after_create')
def receive_after_create(target, connection, **kw):
    "listen for the 'after_create' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the MetaData or Table object which is the target of the event.
- **connection** – the Connection where the CREATE statement or statements have been emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

event.listen() also accepts the propagate=True modifier for this event; when True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when Table.tometadata() is used.

after_drop(*target*, *connection*, ***kw*)

Called after DROP statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'after_drop')
def receive_after_drop(target, connection, **kw):
    "listen for the 'after_drop' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the MetaData or Table object which is the target of the event.
- **connection** – the Connection where the DROP statement or statements have been emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

event.listen() also accepts the propagate=True modifier for this event; when True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when Table.tometadata() is used.

after_parent_attach(*target*, *parent*)
Called after a SchemaItem is associated with a parent SchemaItem.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'after_parent_attach')
def receive_after_parent_attach(target, parent):
    "listen for the 'after_parent_attach' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the target object
- **parent** – the parent to which the target is being attached.

event.listen() also accepts the propagate=True modifier for this event; when True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when Table.tometadata() is used.

before_create(*target*, *connection*, ***kw*)
Called before CREATE statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'before_create')
def receive_before_create(target, connection, **kw):
    "listen for the 'before_create' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the MetaData or Table object which is the target of the event.
- **connection** – the Connection where the CREATE statement or statements will be emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

`event.listen()` also accepts the `propagate=True` modifier for this event; when True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when `Table.tometadata()` is used.

`before_drop`(*target*, *connection*, ***kw*)
Called before DROP statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'before_drop')
def receive_before_drop(target, connection, **kw):
    "listen for the 'before_drop' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the `MetaData` or `Table` object which is the target of the event.
- **connection** – the `Connection` where the DROP statement or statements will be emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

`event.listen()` also accepts the `propagate=True` modifier for this event; when True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when `Table.tometadata()` is used.

`before_parent_attach`(*target*, *parent*)
Called before a `SchemaItem` is associated with a parent `SchemaItem`.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'before_parent_attach')
def receive_before_parent_attach(target, parent):
    "listen for the 'before_parent_attach' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the target object
- **parent** – the parent to which the target is being attached.

`event.listen()` also accepts the `propagate=True` modifier for this event; when True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when `Table.tometadata()` is used.

`column_reflect`(*inspector*, *table*, *column_info*)
Called for each unit of 'column info' retrieved when a `Table` is being reflected.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'column_reflect')
def receive_column_reflect(inspector, table, column_info):
    "listen for the 'column_reflect' event"

    # ... (event handling logic) ...
```

The dictionary of column information as returned by the dialect is passed, and can be modified. The dictionary is that returned in each element of the list returned by `reflection.Inspector.get_columns()`:

- `name` - the column's name
- `type` - the type of this column, which should be an instance of `TypeEngine`
- `nullable` - boolean flag if the column is NULL or NOT NULL
- `default` - the column's server default value. This is normally specified as a plain string SQL expression, however the event can pass a `FetchedValue`, `DefaultClause`, or `sql.expression.text()` object as well.

  Changed in version 1.1.6: The `DDLEvents.column_reflect()` event allows a non string `FetchedValue`, `sql.expression.text()`, or derived object to be specified as the value of `default` in the column dictionary.
- `attrs` - dict containing optional column attributes

The event is called before any action is taken against this dictionary, and the contents can be modified. The `Column` specific arguments `info`, `key`, and `quote` can also be added to the dictionary and will be passed to the constructor of `Column`.

Note that this event is only meaningful if either associated with the `Table` class across the board, e.g.:

```
from sqlalchemy.schema import Table
from sqlalchemy import event

def listen_for_reflect(inspector, table, column_info):
    "receive a column_reflect event"
    # ...

event.listen(
        Table,
        'column_reflect',
        listen_for_reflect)
```

...or with a specific `Table` instance using the `listeners` argument:

```
def listen_for_reflect(inspector, table, column_info):
    "receive a column_reflect event"
    # ...

t = Table(
    'sometable',
    autoload=True,
    listeners=[
        ('column_reflect', listen_for_reflect)
    ])
```

This because the reflection process initiated by `autoload=True` completes within the scope of the constructor for `Table`.

event.listen() also accepts the propagate=True modifier for this event; when True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when Table.tometadata() is used.

class sqlalchemy.events.SchemaEventTarget
Base class for elements that are the targets of DDLEvents events.

This includes SchemaItem as well as SchemaType.

3.6 Core API Basics

3.6.1 Events

SQLAlchemy includes an event API which publishes a wide variety of hooks into the internals of both SQLAlchemy Core and ORM.

New in version 0.7: The system supersedes the previous system of "extension", "proxy", and "listener" classes.

Event Registration

Subscribing to an event occurs through a single API point, the listen() function, or alternatively the listens_for() decorator. These functions accept a target, a string identifier which identifies the event to be intercepted, and a user-defined listening function. Additional positional and keyword arguments to these two functions may be supported by specific types of events, which may specify alternate interfaces for the given event function, or provide instructions regarding secondary event targets based on the given target.

The name of an event and the argument signature of a corresponding listener function is derived from a class bound specification method, which exists bound to a marker class that's described in the documentation. For example, the documentation for PoolEvents.connect() indicates that the event name is "connect" and that a user-defined listener function should receive two positional arguments:

```
from sqlalchemy.event import listen
from sqlalchemy.pool import Pool

def my_on_connect(dbapi_con, connection_record):
    print("New DBAPI connection:", dbapi_con)

listen(Pool, 'connect', my_on_connect)
```

To listen with the listens_for() decorator looks like:

```
from sqlalchemy.event import listens_for
from sqlalchemy.pool import Pool

@listens_for(Pool, "connect")
def my_on_connect(dbapi_con, connection_record):
    print("New DBAPI connection:", dbapi_con)
```

Named Argument Styles

There are some varieties of argument styles which can be accepted by listener functions. Taking the example of PoolEvents.connect(), this function is documented as receiving dbapi_connection and connection_record arguments. We can opt to receive these arguments by name, by establishing a listener function that accepts **keyword arguments, by passing named=True to either listen() or listens_for():

```
from sqlalchemy.event import listens_for
from sqlalchemy.pool import Pool

@listens_for(Pool, "connect", named=True)
def my_on_connect(**kw):
    print("New DBAPI connection:", kw['dbapi_connection'])
```

When using named argument passing, the names listed in the function argument specification will be used as keys in the dictionary.

Named style passes all arguments by name regardless of the function signature, so specific arguments may be listed as well, in any order, as long as the names match up:

```
from sqlalchemy.event import listens_for
from sqlalchemy.pool import Pool

@listens_for(Pool, "connect", named=True)
def my_on_connect(dbapi_connection, **kw):
    print("New DBAPI connection:", dbapi_connection)
    print("Connection record:", kw['connection_record'])
```

Above, the presence of `**kw` tells `listens_for()` that arguments should be passed to the function by name, rather than positionally.

New in version 0.9.0: Added optional `named` argument dispatch to event calling.

Targets

The `listen()` function is very flexible regarding targets. It generally accepts classes, instances of those classes, and related classes or objects from which the appropriate target can be derived. For example, the above mentioned `"connect"` event accepts `Engine` classes and objects as well as `Pool` classes and objects:

```
from sqlalchemy.event import listen
from sqlalchemy.pool import Pool, QueuePool
from sqlalchemy import create_engine
from sqlalchemy.engine import Engine
import psycopg2

def connect():
    return psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')

my_pool = QueuePool(connect)
my_engine = create_engine('postgresql://ed@localhost/test')

# associate listener with all instances of Pool
listen(Pool, 'connect', my_on_connect)

# associate listener with all instances of Pool
# via the Engine class
listen(Engine, 'connect', my_on_connect)

# associate listener with my_pool
listen(my_pool, 'connect', my_on_connect)

# associate listener with my_engine.pool
listen(my_engine, 'connect', my_on_connect)
```

Modifiers

Some listeners allow modifiers to be passed to `listen()`. These modifiers sometimes provide alternate calling signatures for listeners. Such as with ORM events, some event listeners can have a return value which modifies the subsequent handling. By default, no listener ever requires a return value, but by passing `retval=True` this value can be supported:

```
def validate_phone(target, value, oldvalue, initiator):
    """Strip non-numeric characters from a phone number"""

    return re.sub(r'\D', '', value)

# setup listener on UserContact.phone attribute, instructing
# it to use the return value
listen(UserContact.phone, 'set', validate_phone, retval=True)
```

Event Reference

Both SQLAlchemy Core and SQLAlchemy ORM feature a wide variety of event hooks:

- **Core Events** - these are described in core_event_toplevel and include event hooks specific to connection pool lifecycle, SQL statement execution, transaction lifecycle, and schema creation and teardown.
- **ORM Events** - these are described in orm_event_toplevel, and include event hooks specific to class and attribute instrumentation, object initialization hooks, attribute on-change hooks, session state, flush, and commit hooks, mapper initialization, object/result population, and per-instance persistence hooks.

API Reference

sqlalchemy.event.**listen**(*target*, *identifier*, *fn*, **args*, ***kw*)

Register a listener function for the given target.

e.g.:

```
from sqlalchemy import event
from sqlalchemy.schema import UniqueConstraint

def unique_constraint_name(const, table):
    const.name = "uq_%s_%s" % (
        table.name,
        list(const.columns)[0].name
    )
event.listen(
        UniqueConstraint,
        "after_parent_attach",
        unique_constraint_name)
```

A given function can also be invoked for only the first invocation of the event using the `once` argument:

```
def on_config():
    do_config()

event.listen(Mapper, "before_configure", on_config, once=True)
```

New in version 0.9.4: Added `once=True` to `event.listen()` and `event.listens_for()`.

Note: The `listen()` function cannot be called at the same time that the target event is being run. This has implications for thread safety, and also means an event cannot be added from inside the listener function for itself. The list of events to be run are present inside of a mutable collection that can't be changed during iteration.

Event registration and removal is not intended to be a "high velocity" operation; it is a configurational operation. For systems that need to quickly associate and deassociate with events at high scale, use a mutable structure that is handled from inside of a single listener.

Changed in version 1.0.0: - a `collections.deque()` object is now used as the container for the list of events, which explicitly disallows collection mutation while the collection is being iterated.

See also:

`listens_for()`

`remove()`

`sqlalchemy.event.listens_for(target, identifier, *args, **kw)`

Decorate a function as a listener for the given target + identifier.

e.g.:

```
from sqlalchemy import event
from sqlalchemy.schema import UniqueConstraint

@event.listens_for(UniqueConstraint, "after_parent_attach")
def unique_constraint_name(const, table):
    const.name = "uq_%s_%s" % (
        table.name,
        list(const.columns)[0].name
    )
```

A given function can also be invoked for only the first invocation of the event using the `once` argument:

```
@event.listens_for(Mapper, "before_configure", once=True)
def on_config():
    do_config()
```

New in version 0.9.4: Added `once=True` to `event.listen()` and `event.listens_for()`.

See also:

`listen()` - general description of event listening

`sqlalchemy.event.remove(target, identifier, fn)`

Remove an event listener.

The arguments here should match exactly those which were sent to `listen()`; all the event registration which proceeded as a result of this call will be reverted by calling `remove()` with the same arguments.

e.g.:

```
# if a function was registered like this...
@event.listens_for(SomeMappedClass, "before_insert", propagate=True)
def my_listener_function(*arg):
    pass

# ... it's removed like this
event.remove(SomeMappedClass, "before_insert", my_listener_function)
```

Above, the listener function associated with `SomeMappedClass` was also propagated to subclasses of `SomeMappedClass`; the `remove()` function will revert all of these operations.

New in version 0.9.0.

Note: The `remove()` function cannot be called at the same time that the target event is being run. This has implications for thread safety, and also means an event cannot be removed from inside the listener function for itself. The list of events to be run are present inside of a mutable collection that can't be changed during iteration.

Event registration and removal is not intended to be a "high velocity" operation; it is a configurational operation. For systems that need to quickly associate and deassociate with events at high scale, use a mutable structure that is handled from inside of a single listener.

Changed in version 1.0.0: - a `collections.deque()` object is now used as the container for the list of events, which explicitly disallows collection mutation while the collection is being iterated.

See also:

`listen()`

`sqlalchemy.event.contains`(*target*, *identifier*, *fn*)
Return True if the given target/ident/fn is set up to listen.

New in version 0.9.0.

3.6.2 Runtime Inspection API

The inspection module provides the `inspect()` function, which delivers runtime information about a wide variety of SQLAlchemy objects, both within the Core as well as the ORM.

The `inspect()` function is the entry point to SQLAlchemy's public API for viewing the configuration and construction of in-memory objects. Depending on the type of object passed to `inspect()`, the return value will either be a related object which provides a known interface, or in many cases it will return the object itself.

The rationale for `inspect()` is twofold. One is that it replaces the need to be aware of a large variety of "information getting" functions in SQLAlchemy, such as `Inspector.from_engine()`, `orm.attributes.instance_state()`, `orm.class_mapper()`, and others. The other is that the return value of `inspect()` is guaranteed to obey a documented API, thus allowing third party tools which build on top of SQLAlchemy configurations to be constructed in a forwards-compatible way.

New in version 0.8: The `inspect()` system is introduced as of version 0.8.

`sqlalchemy.inspection.inspect`(*subject*, *raiseerr=True*)
Produce an inspection object for the given target.

The returned value in some cases may be the same object as the one given, such as if a `Mapper` object is passed. In other cases, it will be an instance of the registered inspection type for the given object, such as if an `engine.Engine` is passed, an `Inspector` object is returned.

Parameters

- **`subject`** – the subject to be inspected.
- **`raiseerr`** – When `True`, if the given subject does not correspond to a known SQLAlchemy inspected type, `sqlalchemy.exc.NoInspectionAvailable` is raised. If `False`, `None` is returned.

Available Inspection Targets

Below is a listing of many of the most common inspection targets.

- `Connectable` (i.e. `Engine`, `Connection`) - returns an `Inspector` object.

- `ClauseElement` - all SQL expression components, including `Table`, `Column`, serve as their own inspection objects, meaning any of these objects passed to `inspect()` return themselves.
- `object` - an object given will be checked by the ORM for a mapping - if so, an `InstanceState` is returned representing the mapped state of the object. The `InstanceState` also provides access to per attribute state via the `AttributeState` interface as well as the per-flush "history" of any attribute via the `History` object.
- `type` (i.e. a class) - a class given will be checked by the ORM for a mapping - if so, a `Mapper` for that class is returned.
- mapped attribute - passing a mapped attribute to `inspect()`, such as `inspect(MyClass.some_attribute)`, returns a `QueryableAttribute` object, which is the descriptor associated with a mapped class. This descriptor refers to a `MapperProperty`, which is usually an instance of `ColumnProperty` or `RelationshipProperty`, via its `QueryableAttribute.property` attribute.
- `AliasedClass` - returns an `AliasedInsp` object.

3.6.3 Deprecated Event Interfaces

This section describes the class-based core event interface introduced in SQLAlchemy 0.5. The ORM analogue is described at dep_interfaces_orm_toplevel.

Deprecated since version 0.7: The new event system described in event_toplevel replaces the extension/proxy/listener system, providing a consistent interface to all events without the need for subclassing.

Execution, Connection and Cursor Events

class sqlalchemy.interfaces.ConnectionProxy

Allows interception of statement execution by Connections.

Note: `ConnectionProxy` is deprecated. Please refer to `ConnectionEvents`.

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, executemany):
        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.

Connection Pool Events

class sqlalchemy.interfaces.**PoolListener**
Hooks into the lifecycle of connections in a `Pool`.

Note: `PoolListener` is deprecated. Please refer to `PoolEvents`.

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` exe-

cution. 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

3.6.4 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.AmbiguousForeignKeysError`(**arg*, ***kw*)
Raised when more than one foreign key matching can be located between two selectables during a join.

exception sqlalchemy.exc.ArgumentError(*arg, **kw)
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, msg=None, code=None)
Raised by topological sorts when a circular dependency is detected.

There are two scenarios where this error occurs:

- In a Session flush operation, if two objects are mutually dependent on each other, they can not be inserted or deleted via INSERT or DELETE statements alone; an UPDATE will be needed to post-associate or pre-deassociate one of the foreign key constrained values. The `post_update` flag described at post_update can resolve this cycle.
- In a `MetaData.sorted_tables` operation, two `ForeignKey` or `ForeignKeyConstraint` objects mutually refer to each other. Apply the `use_alter=True` flag to one or both, see use_alter.

exception sqlalchemy.exc.CompileError(*arg, **kw)
Raised when an error occurs during SQL compilation

exception sqlalchemy.exc.DBAPIError(statement, params, orig, connection_invalidated=False, code=None)
Raised when the execution of a database operation fails.

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.

`DBAPIError` features `statement` and `params` attributes which supply context regarding the specifics of the statement which had an issue, for the typical case when the error was raised within the context of emitting a SQL statement.

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, code=None)
Wraps a DB-API DataError.

exception sqlalchemy.exc.DatabaseError(statement, params, orig, connection_invalidated=False, code=None)
Wraps a DB-API DatabaseError.

exception sqlalchemy.exc.DisconnectionError(*arg, **kw)
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 the `PoolEvents.checkout()` event so that the host pool forces a retry; the exception will be caught three times in a row before the pool gives up and raises `InvalidRequestError` regarding the connection attempt.

class sqlalchemy.exc.DontWrapMixin
A mixin class which, when applied to a user-defined Exception class, will not be wrapped inside of `StatementError` if the error is emitted within the process of executing a statement.

E.g.:

```
from sqlalchemy.exc import DontWrapMixin

class MyCustomException(Exception, DontWrapMixin):
    pass

class MySpecialType(TypeDecorator):
    impl = String
```

```
def process_bind_param(self, value, dialect):
    if value == 'invalid':
        raise MyCustomException("invalid!")
```

exception sqlalchemy.exc.**IdentifierError**(**arg*, ***kw*)
Raised when a schema name is beyond the max character limit

exception sqlalchemy.exc.**IntegrityError**(*statement*, *params*, *orig*, *connection_invalidated=False*, *code=None*)
Wraps a DB-API IntegrityError.

exception sqlalchemy.exc.**InterfaceError**(*statement*, *params*, *orig*, *connection_invalidated=False*, *code=None*)
Wraps a DB-API InterfaceError.

exception sqlalchemy.exc.**InternalError**(*statement*, *params*, *orig*, *connection_invalidated=False*, *code=None*)
Wraps a DB-API InternalError.

exception sqlalchemy.exc.**InvalidRequestError**(**arg*, ***kw*)
SQLAlchemy was asked to do something it can't do.

This error generally corresponds to runtime state errors.

exception sqlalchemy.exc.**InvalidatePoolError**(**arg*, ***kw*)
Raised when the connection pool should invalidate all stale connections.

A subclass of `DisconnectionError` that indicates that the disconnect situation encountered on the connection probably means the entire pool should be invalidated, as the database has been restarted.

This exception will be handled otherwise the same way as `DisconnectionError`, allowing three attempts to reconnect before giving up.

New in version 1.2.

exception sqlalchemy.exc.**NoForeignKeysError**(**arg*, ***kw*)
Raised when no foreign keys can be located between two selectables during a join.

exception sqlalchemy.exc.**NoInspectionAvailable**(**arg*, ***kw*)
A subject passed to `sqlalchemy.inspection.inspect()` produced no context for inspection.

exception sqlalchemy.exc.**NoReferenceError**(**arg*, ***kw*)
Raised by `ForeignKey` to indicate a reference cannot be resolved.

exception sqlalchemy.exc.**NoReferencedColumnError**(*message*, *tname*, *cname*)
Raised by `ForeignKey` when the referred `Column` cannot be located.

exception sqlalchemy.exc.**NoReferencedTableError**(*message*, *tname*)
Raised by `ForeignKey` when the referred `Table` cannot be located.

exception sqlalchemy.exc.**NoSuchColumnError**
A nonexistent column is requested from a `RowProxy`.

exception sqlalchemy.exc.**NoSuchModuleError**(**arg*, ***kw*)
Raised when a dynamically-loaded module (usually a database dialect) of a particular name cannot be located.

exception sqlalchemy.exc.**NoSuchTableError**(**arg*, ***kw*)
Table does not exist or is not visible to a connection.

exception sqlalchemy.exc.**NotSupportedError**(*statement*, *params*, *orig*, *connection_invalidated=False*, *code=None*)
Wraps a DB-API NotSupportedError.

exception sqlalchemy.exc.**ObjectNotExecutableError**(*target*)
Raised when an object is passed to .execute() that can't be executed as SQL.

New in version 1.1.

exception sqlalchemy.exc.OperationalError(*statement*, *params*, *orig*, *connection_invalidated=False*, *code=None*)
Wraps a DB-API OperationalError.

exception sqlalchemy.exc.ProgrammingError(*statement*, *params*, *orig*, *connection_invalidated=False*, *code=None*)
Wraps a DB-API ProgrammingError.

exception sqlalchemy.exc.ResourceClosedError(**arg*, ***kw*)
An operation was requested from a connection, cursor, or other object that's in a closed state.

exception sqlalchemy.exc.SADeprecationWarning
Issued once per usage of a deprecated API.

exception sqlalchemy.exc.SAPendingDeprecationWarning
Issued once per usage of a deprecated API.

exception sqlalchemy.exc.SAWarning
Issued at runtime.

exception sqlalchemy.exc.SQLAlchemyError(**arg*, ***kw*)
Generic error class.

exception sqlalchemy.exc.StatementError(*message*, *statement*, *params*, *orig*, *code=None*)
An error occurred during execution of a SQL statement.

`StatementError` wraps the exception raised during execution, and features `statement` and `params` attributes which supply context regarding the specifics of the statement which had an issue.

The wrapped exception object is available in the `orig` attribute.

orig = None
The DBAPI exception object.

params = None
The parameter list being used when this exception occurred.

statement = None
The string SQL statement being invoked when this exception occurred.

exception sqlalchemy.exc.TimeoutError(**arg*, ***kw*)
Raised when a connection pool times out on getting a connection.

exception sqlalchemy.exc.UnboundExecutionError(**arg*, ***kw*)
SQL was attempted without a database connection to execute it on.

exception sqlalchemy.exc.UnreflectableTableError(**arg*, ***kw*)
Table exists but can't be reflectted for some reason.

New in version 1.2.

exception sqlalchemy.exc.UnsupportedCompilationError(*compiler*, *element_type*)
Raised when an operation is not supported by the given compiler.

New in version 0.8.3.

3.6.5 Core Internals

Some key internal constructs are listed here.

class sqlalchemy.engine.interfaces.Compiled(*dialect*, *statement*, *bind=None*, *schema_translate_map=None*, *compile_kwargs={}*)
Represent a compiled SQL or DDL expression.

The `__str__` method of the `Compiled` object should produce the actual text of the statement. `Compiled` objects are specific to their underlying database dialect, and also may or may not be specific to the columns referenced within a particular set of bind parameters. In no case should the `Compiled` object be dependent on the actual values of those bind parameters, even though it may reference those values as defaults.

compile()
Produce the internal string representation of this element.

Deprecated since version 0.7: `Compiled` objects now compile within the constructor.

construct_params(*params=None*)
Return the bind params for this compiled object.

Parameters **params** – a dict of string/object pairs whose values will override bind values compiled in to the statement.

execute(**multiparams*, ***params*)
Execute this compiled object.

execution_options = {}
Execution options propagated from the statement. In some cases, sub-elements of the statement can modify these.

params
Return the bind params for this compiled object.

scalar(**multiparams*, ***params*)
Execute this compiled object and return the result's scalar value.

sql_compiler
Return a Compiled that is capable of processing SQL expressions.

If this compiler is one, it would likely just return 'self'.

class sqlalchemy.sql.compiler.DDLCompiler(*dialect*, *statement*, *bind=None*, *schema_translate_map=None*, *compile_kwargs={}*)

compile()
Produce the internal string representation of this element.

Deprecated since version 0.7: `Compiled` objects now compile within the constructor.

define_constraint_remote_table(*constraint*, *table*, *preparer*)
Format the remote table clause of a CREATE CONSTRAINT clause.

execute(**multiparams*, ***params*)
Execute this compiled object.

params
Return the bind params for this compiled object.

scalar(**multiparams*, ***params*)
Execute this compiled object and return the result's scalar value.

class sqlalchemy.engine.default.DefaultDialect(*convert_unicode=False*, *encoding='utf-8'*, *paramstyle=None*, *dbapi=None*, *implicit_returning=None*, *supports_right_nested_joins=None*, *case_sensitive=True*, *supports_native_boolean=None*, *empty_in_strategy='static'*, *label_length=None*, ***kwargs*)
Default implementation of Dialect

construct_arguments = None

Optional set of argument specifiers for various SQLAlchemy constructs, typically schema items.

To implement, establish as a series of tuples, as in:

```
construct_arguments = [
    (schema.Index, {
        "using": False,
        "where": None,
        "ops": None
    })
]
```

If the above construct is established on the PostgreSQL dialect, the `Index` construct will now accept the keyword arguments `postgresql_using`, `postgresql_where`, nad `postgresql_ops`. Any other argument specified to the constructor of `Index` which is prefixed with `postgresql_` will raise `ArgumentError`.

A dialect which does not include a `construct_arguments` member will not participate in the argument validation system. For such a dialect, any argument name is accepted by all participating constructs, within the namespace of arguments prefixed with that dialect name. The rationale here is so that third-party dialects that haven't yet implemented this feature continue to function in the old way.

New in version 0.9.2.

See also:

`DialectKWArgs` - implementing base class which consumes `DefaultDialect.construct_arguments`

create_xid()

Create a random two-phase transaction ID.

This id will be passed to do_begin_twophase(), do_rollback_twophase(), do_commit_twophase(). Its format is unspecified.

dbapi_exception_translation_map = {}

mapping used in the extremely unusual case that a DBAPI's published exceptions don't actually have the ___name___ that they are linked towards.

New in version 1.0.5.

denormalize_name(*name*)

convert the given name to a case insensitive identifier for the backend if it is an all-lowercase name.

this method is only used if the dialect defines requires_name_normalize=True.

do_begin_twophase(*connection*, *xid*)

Begin a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid

do_commit_twophase(*connection*, *xid*, *is_prepared=True*, *recover=False*)

Commit a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid

- **is_prepared** – whether or not TwoPhaseTransaction.prepare() was called.
- **recover** – if the recover flag was passed.

do_prepare_twophase(*connection*, *xid*)
Prepare a two phase transaction on the given connection.

Parameters

- **connection** – a Connection.
- **xid** – xid

do_recover_twophase(*connection*)
Recover list of uncommitted prepared two phase transaction identifiers on the given connection.

Parameters **connection** – a Connection.

do_rollback_twophase(*connection*, *xid*, *is_prepared=True*, *recover=False*)
Rollback a two phase transaction on the given connection.

Parameters

- **connection** – a Connection.
- **xid** – xid
- **is_prepared** – whether or not TwoPhaseTransaction.prepare() was called.
- **recover** – if the recover flag was passed.

engine_created(*engine*)
A convenience hook called before returning the final Engine.

If the dialect returned a different class from the get_dialect_cls() method, then the hook is called on both classes, first on the dialect class returned by the get_dialect_cls() method and then on the class on which the method was called.

The hook should be used by dialects and/or wrappers to apply special events to the engine or its components. In particular, it allows a dialect-wrapping class to apply dialect-level events.

New in version 1.0.3.

execute_sequence_format
alias of tuple

get_check_constraints(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about check constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return check constraint information as a list of dicts with these keys:

name the check constraint's name

sqltext the check constraint's SQL expression

****kw** other options passed to the dialect's get_check_constraints() method.

New in version 1.1.0.

get_columns(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about columns in *table_name*.

Given a Connection, a string *table_name*, and an optional string *schema*, return column information as a list of dictionaries with these keys:

name the column's name

type [sqlalchemy.types#TypeEngine]

nullable boolean

default the column's default value

autoincrement boolean

sequence

a dictionary of the form

{'name' [str, 'start' :int, 'increment': int, 'minvalue': int,] 'maxvalue': int, 'nominvalue': bool, 'nomaxvalue': bool, 'cycle': bool, 'cache': int, 'order': bool}

Additional column attributes may be present.

`get_dialect_cls`(*url*)
Given a URL, return the `Dialect` that will be used.

This is a hook that allows an external plugin to provide functionality around an existing dialect, by allowing the plugin to be loaded from the url based on an entrypoint, and then the plugin returns the actual dialect to be used.

By default this just returns the cls.

New in version 1.0.3.

`get_foreign_keys`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about foreign_keys in *table_name*.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return foreign key information as a list of dicts with these keys:

name the constraint's name

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

`get_indexes`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about indexes in *table_name*.

Given a `Connection`, a string *table_name* and an optional string *schema*, return index information as a list of dictionaries with these keys:

name the index's name

column_names list of column names in order

unique boolean

`get_isolation_level`(*dbapi_conn*)
Given a DBAPI connection, return its isolation level.

When working with a `Connection` object, the corresponding DBAPI connection may be procured using the `Connection.connection` accessor.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

create_engine.isolation_level - set per Engine isolation level

get_pk_constraint(*conn*, *table_name*, *schema=None*, ***kw*)
Compatibility method, adapts the result of get_primary_keys() for those dialects which don't implement get_pk_constraint().

get_primary_keys(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about primary keys in *table_name*.

Deprecated. This method is only called by the default implementation of Dialect.get_pk_constraint(). Dialects should instead implement the Dialect.get_pk_constraint() method directly.

get_table_comment(*connection*, *table_name*, *schema=None*, ***kw*)
Return the "comment" for the table identified by *table_name*.

Given a string *table_name* and an optional string *schema*, return table comment information as a dictionary with this key:

text text of the comment

Raises NotImplementedError for dialects that don't support comments.

New in version 1.2.

get_table_names(*connection*, *schema=None*, ***kw*)
Return a list of table names for *schema*.

get_temp_table_names(*connection*, *schema=None*, ***kw*)
Return a list of temporary table names on the given connection, if supported by the underlying backend.

get_temp_view_names(*connection*, *schema=None*, ***kw*)
Return a list of temporary view names on the given connection, if supported by the underlying backend.

get_unique_constraints(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about unique constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return unique constraint information as a list of dicts with these keys:

name the unique constraint's name

column_names list of column names in order

****kw** other options passed to the dialect's get_unique_constraints() method.

New in version 0.9.0.

get_view_definition(*connection*, *view_name*, *schema=None*, ***kw*)
Return view definition.

Given a Connection, a string *view_name*, and an optional string *schema*, return the view definition.

get_view_names(*connection*, *schema=None*, ***kw*)
Return a list of all view names available in the database.

schema: Optional, retrieve names from a non-default schema.

has_sequence(*connection*, *sequence_name*, *schema=None*)
Check the existence of a particular sequence in the database.

Given a Connection object and a string *sequence_name*, return True if the given sequence exists in the database, False otherwise.

has_table(*connection*, *table_name*, *schema=None*)
Check the existence of a particular table in the database.

Given a `Connection` object and a string *table_name*, return True if the given table (possibly within the specified *schema*) exists in the database, False otherwise.

`normalize_name`(*name*)
convert the given name to lowercase if it is detected as case insensitive.

this method is only used if the dialect defines requires_name_normalize=True.

`on_connect`()
return a callable which sets up a newly created DBAPI connection.

This is used to set dialect-wide per-connection options such as isolation modes, unicode modes, etc.

If a callable is returned, it will be assembled into a pool listener that receives the direct DBAPI connection, with all wrappers removed.

If None is returned, no listener will be generated.

`preparer`
alias of `IdentifierPreparer`

`set_isolation_level`(*dbapi_conn*, *level*)
Given a DBAPI connection, set its isolation level.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

`statement_compiler`
alias of `SQLCompiler`

`type_descriptor`(*typeobj*)
Provide a database-specific `TypeEngine` object, given the generic object which comes from the types module.

This method looks for a dictionary called `colspecs` as a class or instance-level variable, and passes on to `types.adapt_type()`.

`class sqlalchemy.engine.interfaces.Dialect`
Define the behavior of a specific database and DB-API combination.

Any aspect of metadata definition, SQL query generation, execution, result-set handling, or anything else which varies between databases is defined under the general category of the Dialect. The Dialect acts as a factory for other database-specific object implementations including ExecutionContext, Compiled, DefaultGenerator, and TypeEngine.

All Dialects implement the following attributes:
name identifying name for the dialect from a DBAPI-neutral point of view (i.e. 'sqlite')
driver identifying name for the dialect's DBAPI
positional True if the paramstyle for this Dialect is positional.
paramstyle the paramstyle to be used (some DB-APIs support multiple paramstyles).
convert_unicode True if Unicode conversion should be applied to all `str` types.
encoding type of encoding to use for unicode, usually defaults to 'utf-8'.
statement_compiler a `Compiled` class used to compile SQL statements
ddl_compiler a `Compiled` class used to compile DDL statements

server_version_info a tuple containing a version number for the DB backend in use. This value is only available for supporting dialects, and is typically populated during the initial connection to the database.

default_schema_name the name of the default schema. This value is only available for supporting dialects, and is typically populated during the initial connection to the database.

execution_ctx_cls a `ExecutionContext` class used to handle statement execution

execute_sequence_format either the 'tuple' or 'list' type, depending on what cursor.execute() accepts for the second argument (they vary).

preparer a `IdentifierPreparer` class used to quote identifiers.

supports_alter `True` if the database supports `ALTER TABLE`.

max_identifier_length The maximum length of identifier names.

supports_unicode_statements Indicate whether the DB-API can receive SQL statements as Python unicode strings

supports_unicode_binds Indicate whether the DB-API can receive string bind parameters as Python unicode strings

supports_sane_rowcount Indicate whether the dialect properly implements rowcount for `UPDATE` and `DELETE` statements.

supports_sane_multi_rowcount Indicate whether the dialect properly implements rowcount for `UPDATE` and `DELETE` statements when executed via executemany.

preexecute_autoincrement_sequences True if 'implicit' primary key functions must be executed separately in order to get their value. This is currently oriented towards PostgreSQL.

implicit_returning use RETURNING or equivalent during INSERT execution in order to load newly generated primary keys and other column defaults in one execution, which are then available via inserted_primary_key. If an insert statement has returning() specified explicitly, the "implicit" functionality is not used and inserted_primary_key will not be available.

colspecs A dictionary of TypeEngine classes from sqlalchemy.types mapped to subclasses that are specific to the dialect class. This dictionary is class-level only and is not accessed from the dialect instance itself.

supports_default_values Indicates if the construct `INSERT INTO tablename DEFAULT VALUES` is supported

supports_sequences Indicates if the dialect supports CREATE SEQUENCE or similar.

sequences_optional If True, indicates if the "optional" flag on the Sequence() construct should signal to not generate a CREATE SEQUENCE. Applies only to dialects that support sequences. Currently used only to allow PostgreSQL SERIAL to be used on a column that specifies Sequence() for usage on other backends.

supports_native_enum Indicates if the dialect supports a native ENUM construct. This will prevent types.Enum from generating a CHECK constraint when that type is used.

supports_native_boolean Indicates if the dialect supports a native boolean construct. This will prevent types.Boolean from generating a CHECK constraint when that type is used.

dbapi_exception_translation_map A dictionary of names that will contain as values the names of pep-249 exceptions ("IntegrityError", "OperationalError", etc) keyed to alternate class names, to support the case where a DBAPI has exception classes that aren't named as they are referred to (e.g. IntegrityError = MyException). In the vast majority of cases this dictionary is empty.

New in version 1.0.5.

`connect()`

return a callable which sets up a newly created DBAPI connection.

The callable accepts a single argument "conn" which is the DBAPI connection itself. It has no return value.

This is used to set dialect-wide per-connection options such as isolation modes, unicode modes, etc.

If a callable is returned, it will be assembled into a pool listener that receives the direct DBAPI connection, with all wrappers removed.

If None is returned, no listener will be generated.

`create_connect_args(`*url*`)`

Build DB-API compatible connection arguments.

Given a `URL` object, returns a tuple consisting of a **args/**kwargs* suitable to send directly to the dbapi's connect function.

`create_xid()`
Create a two-phase transaction ID.

This id will be passed to do_begin_twophase(), do_rollback_twophase(), do_commit_twophase(). Its format is unspecified.

`denormalize_name(`*name*`)`
convert the given name to a case insensitive identifier for the backend if it is an all-lowercase name.

this method is only used if the dialect defines requires_name_normalize=True.

`do_begin(`*dbapi_connection*`)`
Provide an implementation of `connection.begin()`, given a DB-API connection.

The DBAPI has no dedicated "begin" method and it is expected that transactions are implicit. This hook is provided for those DBAPIs that might need additional help in this area.

Note that `Dialect.do_begin()` is not called unless a `Transaction` object is in use. The `Dialect.do_autocommit()` hook is provided for DBAPIs that need some extra commands emitted after a commit in order to enter the next transaction, when the SQLAlchemy `Connection` is used in its default "autocommit" mode.

Parameters `dbapi_connection` – a DBAPI connection, typically proxied within a `ConnectionFairy`.

`do_begin_twophase(`*connection*, *xid*`)`
Begin a two phase transaction on the given connection.

Parameters

- `connection` – a `Connection`.
- `xid` – xid

`do_close(`*dbapi_connection*`)`
Provide an implementation of `connection.close()`, given a DBAPI connection.

This hook is called by the `Pool` when a connection has been detached from the pool, or is being returned beyond the normal capacity of the pool.

New in version 0.8.

`do_commit(`*dbapi_connection*`)`
Provide an implementation of `connection.commit()`, given a DB-API connection.

Parameters `dbapi_connection` – a DBAPI connection, typically proxied within a `ConnectionFairy`.

`do_commit_twophase(`*connection*, *xid*, *is_prepared=True*, *recover=False*`)`
Commit a two phase transaction on the given connection.

Parameters

- `connection` – a `Connection`.
- `xid` – xid
- `is_prepared` – whether or not `TwoPhaseTransaction.prepare()` was called.
- `recover` – if the recover flag was passed.

`do_execute(`*cursor*, *statement*, *parameters*, *context=None*`)`
Provide an implementation of `cursor.execute(statement, parameters)`.

`do_execute_no_params`(*cursor*, *statement*, *parameters*, *context=None*)
Provide an implementation of `cursor.execute(statement)`.

The parameter collection should not be sent.

`do_executemany`(*cursor*, *statement*, *parameters*, *context=None*)
Provide an implementation of `cursor.executemany(statement, parameters)`.

`do_prepare_twophase`(*connection*, *xid*)
Prepare a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid

`do_recover_twophase`(*connection*)
Recover list of uncommitted prepared two phase transaction identifiers on the given connection.

Parameters **connection** – a `Connection`.

`do_release_savepoint`(*connection*, *name*)
Release the named savepoint on a connection.

Parameters

- **connection** – a `Connection`.
- **name** – savepoint name.

`do_rollback`(*dbapi_connection*)
Provide an implementation of `connection.rollback()`, given a DB-API connection.

Parameters **dbapi_connection** – a DBAPI connection, typically proxied within a `ConnectionFairy`.

`do_rollback_to_savepoint`(*connection*, *name*)
Rollback a connection to the named savepoint.

Parameters

- **connection** – a `Connection`.
- **name** – savepoint name.

`do_rollback_twophase`(*connection*, *xid*, *is_prepared=True*, *recover=False*)
Rollback a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid
- **is_prepared** – whether or not `TwoPhaseTransaction.prepare()` was called.
- **recover** – if the recover flag was passed.

`do_savepoint`(*connection*, *name*)
Create a savepoint with the given name.

Parameters

- **connection** – a `Connection`.
- **name** – savepoint name.

`classmethod engine_created`(*engine*)
A convenience hook called before returning the final `Engine`.

If the dialect returned a different class from the `get_dialect_cls()` method, then the hook is called on both classes, first on the dialect class returned by the `get_dialect_cls()` method and then on the class on which the method was called.

The hook should be used by dialects and/or wrappers to apply special events to the engine or its components. In particular, it allows a dialect-wrapping class to apply dialect-level events.

New in version 1.0.3.

`get_check_constraints`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about check constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return check constraint information as a list of dicts with these keys:

name the check constraint's name

sqltext the check constraint's SQL expression

****kw** other options passed to the dialect's get_check_constraints() method.

New in version 1.1.0.

`get_columns`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about columns in *table_name*.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return column information as a list of dictionaries with these keys:

name the column's name

type [sqlalchemy.types#TypeEngine]

nullable boolean

default the column's default value

autoincrement boolean

sequence

a dictionary of the form

{'name' [str, 'start' :int, 'increment': int, 'minvalue': int,] 'maxvalue': int, 'nominvalue': bool, 'nomaxvalue': bool, 'cycle': bool, 'cache': int, 'order': bool}

Additional column attributes may be present.

`classmethod get_dialect_cls`(*url*)
Given a URL, return the `Dialect` that will be used.

This is a hook that allows an external plugin to provide functionality around an existing dialect, by allowing the plugin to be loaded from the url based on an entrypoint, and then the plugin returns the actual dialect to be used.

By default this just returns the cls.

New in version 1.0.3.

`get_foreign_keys`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about foreign_keys in *table_name*.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return foreign key information as a list of dicts with these keys:

name the constraint's name

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

`get_indexes`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about indexes in *table_name*.

Given a `Connection`, a string *table_name* and an optional string *schema*, return index information as a list of dictionaries with these keys:

name the index's name

column_names list of column names in order

unique boolean

`get_isolation_level`(*dbapi_conn*)
Given a DBAPI connection, return its isolation level.

When working with a `Connection` object, the corresponding DBAPI connection may be procured using the `Connection.connection` accessor.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

`get_pk_constraint`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about the primary key constraint on table_name‘.

Given a `Connection`, 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`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about primary keys in *table_name*.

Deprecated. This method is only called by the default implementation of `Dialect.get_pk_constraint()`. Dialects should instead implement the `Dialect.get_pk_constraint()` method directly.

`get_table_comment`(*connection*, *table_name*, *schema=None*, ***kw*)
Return the “comment” for the table identified by *table_name*.

Given a string *table_name* and an optional string *schema*, return table comment information as a dictionary with this key:

text text of the comment

Raises `NotImplementedError` for dialects that don't support comments.

New in version 1.2.

`get_table_names`(*connection*, *schema=None*, ***kw*)
Return a list of table names for *schema*.

`get_temp_table_names`(*connection*, *schema=None*, ***kw*)
Return a list of temporary table names on the given connection, if supported by the underlying backend.

`get_temp_view_names`(*connection*, *schema=None*, ***kw*)
Return a list of temporary view names on the given connection, if supported by the underlying backend.

`get_unique_constraints`(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about unique constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return unique constraint information as a list of dicts with these keys:

name the unique constraint's name

column_names list of column names in order

****kw** other options passed to the dialect's get_unique_constraints() method.

New in version 0.9.0.

`get_view_definition`(*connection*, *view_name*, *schema=None*, ***kw*)
Return view definition.

Given a `Connection`, a string *view_name*, and an optional string *schema*, return the view definition.

`get_view_names`(*connection*, *schema=None*, ***kw*)
Return a list of all view names available in the database.

schema: Optional, retrieve names from a non-default schema.

`has_sequence`(*connection*, *sequence_name*, *schema=None*)
Check the existence of a particular sequence in the database.

Given a `Connection` object and a string *sequence_name*, return True if the given sequence exists in the database, False otherwise.

`has_table`(*connection*, *table_name*, *schema=None*)
Check the existence of a particular table in the database.

Given a `Connection` object and a string *table_name*, return True if the given table (possibly within the specified *schema*) exists in the database, False otherwise.

`initialize`(*connection*)
Called during strategized creation of the dialect with a connection.

Allows dialects to configure options based on server version info or other properties.

The connection passed here is a SQLAlchemy Connection object, with full capabilities.

The initialize() method of the base dialect should be called via super().

`is_disconnect`(*e*, *connection*, *cursor*)
Return True if the given DB-API error indicates an invalid connection

`normalize_name`(*name*)
convert the given name to lowercase if it is detected as case insensitive.

this method is only used if the dialect defines requires_name_normalize=True.

`reflecttable`(*connection*, *table*, *include_columns*, *exclude_columns*)
Load table description from the database.

Given a `Connection` and a `Table` object, reflect its columns and properties from the database.

The implementation of this method is provided by `DefaultDialect.reflecttable()`, which makes use of `Inspector` to retrieve column information.

Dialects should **not** seek to implement this method, and should instead implement individual schema inspection operations such as `Dialect.get_columns()`, `Dialect.get_pk_constraint()`, etc.

`reset_isolation_level`(*dbapi_conn*)
Given a DBAPI connection, revert its isolation to the default.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

`set_isolation_level`(*dbapi_conn*, *level*)
Given a DBAPI connection, set its isolation level.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

`classmethod type_descriptor`(*typeobj*)
Transform a generic type to a dialect-specific type.

Dialect classes will usually use the `types.adapt_type()` function in the types module to accomplish this.

The returned result is cached *per dialect class* so can contain no dialect-instance state.

`class sqlalchemy.engine.default.DefaultExecutionContext`

`current_parameters = None`
A dictionary of parameters applied to the current row.

This attribute is only available in the context of a user-defined default generation function, e.g. as described at context_default_functions. It consists of a dictionary which includes entries for each column/value pair that is to be part of the INSERT or UPDATE statement. The keys of the dictionary will be the key value of each `Column`, which is usually synonymous with the name.

Note that the `DefaultExecutionContext.current_parameters` attribute does not accommodate for the "multi-values" feature of the `Insert.values()` method. The `DefaultExecutionContext.get_current_parameters()` method should be preferred.

See also:

`DefaultExecutionContext.get_current_parameters()`

context_default_functions

`get_current_parameters`(*isolate_multiinsert_groups=True*)
Return a dictionary of parameters applied to the current row.

This method can only be used in the context of a user-defined default generation function, e.g. as described at context_default_functions. When invoked, a dictionary is returned which includes entries for each column/value pair that is part of the INSERT or UPDATE

statement. The keys of the dictionary will be the key value of each `Column`, which is usually synonymous with the name.

Parameters `isolate_multiinsert_groups=True` – indicates that multi-valued INSERT contructs created using `Insert.values()` should be handled by returning only the subset of parameters that are local to the current column default invocation. When `False`, the raw parameters of the statement are returned including the naming convention used in the case of multi-valued INSERT.

New in version 1.2: added `DefaultExecutionContext.get_current_parameters()` which provides more functionality over the existing `DefaultExecutionContext.current_parameters` attribute.

See also:

`DefaultExecutionContext.current_parameters`

context_default_functions

`get_lastrowid()`
return self.cursor.lastrowid, or equivalent, after an INSERT.

This may involve calling special cursor functions, issuing a new SELECT on the cursor (or a new one), or returning a stored value that was calculated within post_exec().

This function will only be called for dialects which support "implicit" primary key generation, keep preexecute_autoincrement_sequences set to False, and when no explicit id value was bound to the statement.

The function is called once, directly after post_exec() and before the transaction is committed or ResultProxy is generated. If the post_exec() method assigns a value to *self._lastrowid*, the value is used in place of calling get_lastrowid().

Note that this method is *not* equivalent to the `lastrowid` method on `ResultProxy`, which is a direct proxy to the DBAPI `lastrowid` accessor in all cases.

`get_result_processor`(*type_*, *colname*, *coltype*)
Return a 'result processor' for a given type as present in cursor.description.

This has a default implementation that dialects can override for context-sensitive result type handling.

`set_input_sizes`(*translate=None*, *include_types=None*, *exclude_types=None*)
Given a cursor and ClauseParameters, call the appropriate style of `setinputsizes()` on the cursor, using DB-API types from the bind parameter's `TypeEngine` objects.

This method only called by those dialects which require it, currently cx_oracle.

`class sqlalchemy.engine.interfaces.ExecutionContext`
A messenger object for a Dialect that corresponds to a single execution.

ExecutionContext should have these data members:

connection Connection object which can be freely used by default value generators to execute SQL. This Connection should reference the same underlying connection/transactional resources of root_connection.

root_connection Connection object which is the source of this ExecutionContext. This Connection may have close_with_result=True set, in which case it can only be used once.

dialect dialect which created this ExecutionContext.

cursor DB-API cursor procured from the connection,

compiled if passed to constructor, sqlalchemy.engine.base.Compiled object being executed,

statement string version of the statement to be executed. Is either passed to the constructor, or must be created from the sql.Compiled object by the time pre_exec() has completed.

parameters bind parameters passed to the execute() method. For compiled statements, this is a dictionary or list of dictionaries. For textual statements, it should be in a format suitable for

the dialect's paramstyle (i.e. dict or list of dicts for non positional, list or list of lists/tuples for positional).

isinsert True if the statement is an INSERT.

isupdate True if the statement is an UPDATE.

should_autocommit True if the statement is a "committable" statement.

prefetch_cols a list of Column objects for which a client-side default was fired off. Applies to inserts and updates.

postfetch_cols a list of Column objects for which a server-side default or inline SQL expression value was fired off. Applies to inserts and updates.

`create_cursor()`
Return a new cursor generated from this ExecutionContext's connection.

Some dialects may wish to change the behavior of connection.cursor(), such as postgresql which may return a PG "server side" cursor.

`exception = None`
A DBAPI-level exception that was caught when this ExecutionContext attempted to execute a statement.

This attribute is meaningful only within the `ConnectionEvents.dbapi_error()` event.

New in version 0.9.7.

See also:

`ExecutionContext.is_disconnect`

`ConnectionEvents.dbapi_error()`

`get_rowcount()`
Return the DBAPI `cursor.rowcount` value, or in some cases an interpreted value.

See `ResultProxy.rowcount` for details on this.

`handle_dbapi_exception(`*e*`)`
Receive a DBAPI exception which occurred upon execute, result fetch, etc.

`is_disconnect = None`
Boolean flag set to True or False when a DBAPI-level exception is caught when this ExecutionContext attempted to execute a statement.

This attribute is meaningful only within the `ConnectionEvents.dbapi_error()` event.

New in version 0.9.7.

See also:

`ExecutionContext.exception`

`ConnectionEvents.dbapi_error()`

`lastrow_has_defaults()`
Return True if the last INSERT or UPDATE row contained inlined or database-side defaults.

`post_exec()`
Called after the execution of a compiled statement.

If a compiled statement was passed to this ExecutionContext, the *last_insert_ids*, *last_inserted_params*, etc. datamembers should be available after this method completes.

`pre_exec()`
Called before an execution of a compiled statement.

If a compiled statement was passed to this ExecutionContext, the *statement* and *parameters* datamembers must be initialized after this statement is complete.

`result()`
Return a result object corresponding to this ExecutionContext.

Returns a ResultProxy.

should_autocommit_text(*statement*)
Parse the given textual statement and return True if it refers to a "committable" statement

class sqlalchemy.log.Identified

class sqlalchemy.sql.compiler.IdentifierPreparer(*dialect*, *initial_quote=''*, *final_quote=None*, *escape_quote=''*, *quote_case_sensitive_collations=True*, *omit_schema=False*)
Handle quoting and case-folding of identifiers based on options.

format_column(*column*, *use_table=False*, *name=None*, *table_name=None*, *use_schema=False*)
Prepare a quoted column name.

format_schema(*name*, *quote=None*)
Prepare a quoted schema name.

format_table(*table*, *use_schema=True*, *name=None*)
Prepare a quoted table and schema name.

format_table_seq(*table*, *use_schema=True*)
Format table name and schema as a tuple.

quote(*ident*, *force=None*)
Conditionally quote an identifier.

the 'force' flag should be considered deprecated.

quote_identifier(*value*)
Quote an identifier.

Subclasses should override this to provide database-dependent quoting behavior.

quote_schema(*schema*, *force=None*)
Conditionally quote a schema.

Subclasses can override this to provide database-dependent quoting behavior for schema names.

the 'force' flag should be considered deprecated.

unformat_identifiers(*identifiers*)
Unpack 'schema.table.column'-like strings into components.

class sqlalchemy.sql.compiler.SQLCompiler(*dialect*, *statement*, *column_keys=None*, *inline=False*, ***kwargs*)
Default implementation of `Compiled`.

Compiles `ClauseElement` objects into SQL strings.

ansi_bind_rules = False
SQL 92 doesn't allow bind parameters to be used in the columns clause of a SELECT, nor does it allow ambiguous expressions like "? = ?". A compiler subclass can set this flag to False if the target driver/DB enforces this

construct_params(*params=None*, *_group_number=None*, *_check=True*)
return a dictionary of bind parameter keys and values

contains_expanding_parameters = False
True if we've encountered bindparam(..., expanding=True).

These need to be converted before execution time against the string statement.

default_from()
Called when a SELECT statement has no froms, and no FROM clause is to be appended.

Gives Oracle a chance to tack on a `FROM DUAL` to the string output.

delete_extra_from_clause(*update_stmt*, *from_table*, *extra_froms*, *from_hints*, ***kw*)
Provide a hook to override the generation of an DELETE..FROM clause.

This can be used to implement DELETE..USING for example.

MySQL and MSSQL override this.

get_select_precolumns(*select*, ***kw*)
Called when building a `SELECT` statement, position is just before column list.

isdelete = False
class-level defaults which can be set at the instance level to define if this Compiled instance represents INSERT/UPDATE/DELETE

isinsert = False
class-level defaults which can be set at the instance level to define if this Compiled instance represents INSERT/UPDATE/DELETE

isupdate = False
class-level defaults which can be set at the instance level to define if this Compiled instance represents INSERT/UPDATE/DELETE

params
Return the bind param dictionary embedded into this compiled object, for those values that are present.

render_literal_value(*value*, *type_*)
Render the value of a bind parameter as a quoted literal.

This is used for statement sections that do not accept bind parameters on the target driver/database.

This should be implemented by subclasses using the quoting services of the DBAPI.

render_table_with_column_in_update_from = False
set to True classwide to indicate the SET clause in a multi-table UPDATE statement should qualify columns with the table name (i.e. MySQL only)

returning = None
holds the "returning" collection of columns if the statement is CRUD and defines returning columns either implicitly or explicitly

returning_precedes_values = False
set to True classwide to generate RETURNING clauses before the VALUES or WHERE clause (i.e. MSSQL)

update_from_clause(*update_stmt*, *from_table*, *extra_froms*, *from_hints*, ***kw*)
Provide a hook to override the generation of an UPDATE..FROM clause.

MySQL and MSSQL override this.

update_limit_clause(*update_stmt*)
Provide a hook for MySQL to add LIMIT to the UPDATE

update_tables_clause(*update_stmt*, *from_table*, *extra_froms*, ***kw*)
Provide a hook to override the initial table clause in an UPDATE statement.

MySQL overrides this.

CHAPTER

FOUR

DIALECTS

The **dialect** is the system SQLAlchemy uses to communicate with various types of DBAPI implementations and 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.

All dialects require that an appropriate DBAPI driver is installed.

4.1 Included Dialects

4.1.1 Firebird

Firebird 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.

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.

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()
```

fdb

Arguments

The `fdb` dialect is based on the *sqlalchemy.dialects.firebird.kinterbasdb* dialect, however does not accept every argument that Kinterbasdb does.

- `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 `Connection.execution_options()`:

  ```
  conn = engine.connect().execution_options(enable_rowcount=True)
  r = conn.execute(stmt)
  print r.rowcount
  ```

- `retaining` - False by default. Setting this to True will pass the `retaining=True` keyword argument to the `.commit()` and `.rollback()` methods of the DBAPI connection, which can improve performance in some situations, but apparently with significant caveats. Please read the fdb and/or kinterbasdb DBAPI documentation in order to understand the implications of this flag.

  New in version 0.8.2: - `retaining` keyword argument specifying transaction retaining behavior - in 0.8 it defaults to `True` for backwards compatibility.

  Changed in version 0.9.0: - the `retaining` flag defaults to `False`. In 0.8 it defaulted to `True`.

  See also:

  http://pythonhosted.org/fdb/usage-guide.html#retaining-transactions - information on the "retaining" flag.

kinterbasdb

Arguments

The Kinterbasdb backend accepts the `enable_rowcount` and `retaining` arguments accepted by the *sqlalchemy.dialects.firebird.fdb* dialect. In addition, it also accepts the following:

- `type_conv` - select the kind of mapping done on the types: by default SQLAlchemy uses 200 with Unicode, datetime and decimal support. See the linked documents below for further information.
- `concurrency_level` - set the backend policy with regards to threading issues: by default SQLAlchemy uses policy 1. See the linked documents below for further information.

See also:

http://sourceforge.net/projects/kinterbasdb

http://kinterbasdb.sourceforge.net/dist_docs/usage.html#adv_param_conv_dynamic_type_translation

http://kinterbasdb.sourceforge.net/dist_docs/usage.html#special_issue_concurrency

4.1.2 Microsoft SQL Server

Auto Increment Behavior

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

```
from sqlalchemy import Table, MetaData, Column, Integer

m = MetaData()
t = Table('t', m,
        Column('id', Integer, primary_key=True),
        Column('x', Integer))
m.create_all(engine)
```

The above example will generate DDL as:

```
CREATE TABLE t (
    id INTEGER NOT NULL IDENTITY(1,1),
    x INTEGER NULL,
    PRIMARY KEY (id)
)
```

For the case where this default generation of `IDENTITY` is not desired, specify `autoincrement=False` on all integer primary key columns:

```
m = MetaData()
t = Table('t', m,
        Column('id', Integer, primary_key=True, autoincrement=False),
        Column('x', Integer))
m.create_all(engine)
```

Note: An INSERT statement which refers to an explicit value for such a column is prohibited by SQL Server, however SQLAlchemy will detect this and modify the `IDENTITY_INSERT` flag accordingly at statement execution time. As this is not a high performing process, care should be taken to set the `autoincrement` flag appropriately for columns that will not actually require IDENTITY behavior.

Controlling "Start" and "Increment"

Specific control over the parameters of the `IDENTITY` value is supported using the `schema.Sequence` object. While this object normally represents an explicit "sequence" for supporting backends, on SQL Server it is re-purposed to specify behavior regarding the identity column, including support of the "start" and "increment" values:

```
from sqlalchemy import Table, Integer, Sequence, Column

Table('test', metadata,
```

```
      Column('id', Integer,
             Sequence('blah', start=100, increment=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.

INSERT behavior

Handling of the `IDENTITY` column at INSERT time involves two key techniques. The most common is being able to fetch the "last inserted value" for a given `IDENTITY` column, a process which SQLAlchemy performs implicitly in many cases, most importantly within the ORM.

The process for fetching this value has several variants:

- In the vast majority of cases, RETURNING is used in conjunction with INSERT statements on SQL Server in order to get newly generated primary key values:

  ```
  INSERT INTO t (x) OUTPUT inserted.id VALUES (?)
  ```

- When RETURNING is not available or has been disabled via `implicit_returning=False`, either the `scope_identity()` function or the `@@identity` variable is used; behavior varies by backend:
  - when using PyODBC, the phrase `; select scope_identity()` will be appended to the end of the INSERT statement; a second result set will be fetched in order to receive the value. Given a table as:

    ```
    t = Table('t', m, Column('id', Integer, primary_key=True),
            Column('x', Integer),
            implicit_returning=False)
    ```

    an INSERT will look like:

    ```
    INSERT INTO t (x) VALUES (?); select scope_identity()
    ```

  - Other dialects such as pymssql will call upon `SELECT scope_identity() AS lastrowid` subsequent to an INSERT statement. If the flag `use_scope_identity=False` is passed to `create_engine()`, the statement `SELECT @@identity AS lastrowid` is used instead.

A table that contains an `IDENTITY` column will prohibit an INSERT statement that refers to the identity column explicitly. The SQLAlchemy dialect will detect when an INSERT construct, created using a core `insert()` construct (not a plain string SQL), refers to the identity column, and in this case will emit `SET IDENTITY_INSERT ON` prior to the insert statement proceeding, and `SET IDENTITY_INSERT OFF` subsequent to the execution. Given this example:

```
m = MetaData()
t = Table('t', m, Column('id', Integer, primary_key=True),
                Column('x', Integer))
m.create_all(engine)

engine.execute(t.insert(), {'id': 1, 'x':1}, {'id':2, 'x':2})
```

The above column will be created with IDENTITY, however the INSERT statement we emit is specifying explicit values. In the echo output we can see how SQLAlchemy handles this:

```
CREATE TABLE t (
    id INTEGER NOT NULL IDENTITY(1,1),
    x INTEGER NULL,
    PRIMARY KEY (id)
)

COMMIT
SET IDENTITY_INSERT t ON
INSERT INTO t (id, x) VALUES (?, ?)
((1, 1), (2, 2))
SET IDENTITY_INSERT t OFF
COMMIT
```

This is an auxiliary use case suitable for testing and bulk insert scenarios.

MAX on VARCHAR / NVARCHAR

SQL Server supports the special string "MAX" within the `sqltypes.VARCHAR` and `sqltypes.NVARCHAR` datatypes, to indicate "maximum length possible". The dialect currently handles this as a length of "None" in the base type, rather than supplying a dialect-specific version of these types, so that a base type specified such as `VARCHAR(None)` can assume "unlengthed" behavior on more than one backend without using dialect-specific types.

To build a SQL Server VARCHAR or NVARCHAR with MAX length, use None:

```
my_table = Table(
    'my_table', metadata,
    Column('my_data', VARCHAR(None)),
    Column('my_n_data', NVARCHAR(None))
)
```

Collation Support

Character collations are supported by the base string types, specified by the string argument "collation":

```
from sqlalchemy 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
```

New in version 0.8: Character collations are now part of the base string types.

LIMIT/OFFSET Support

MSSQL has no support for the LIMIT or OFFSET keywords. 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.

Transaction Isolation Level

All SQL Server dialects support setting of transaction isolation level both via a dialect-specific parameter `create_engine.isolation_level` accepted by `create_engine()`, as well as the `Connection.execution_options.isolation_level` argument as passed to `Connection.execution_options()`. This feature works by issuing the command `SET TRANSACTION ISOLATION LEVEL <level>` for each new connection.

To set isolation level using `create_engine()`:

```
engine = create_engine(
    "mssql+pyodbc://scott:tiger@ms_2008",
    isolation_level="REPEATABLE READ"
)
```

To set using per-connection execution options:

```
connection = engine.connect()
connection = connection.execution_options(
    isolation_level="READ COMMITTED"
)
```

Valid values for `isolation_level` include:

- `AUTOCOMMIT` - pyodbc / pymssql-specific
- `READ COMMITTED`
- `READ UNCOMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`
- `SNAPSHOT` - specific to SQL Server

New in version 1.1: support for isolation level setting on Microsoft SQL Server.

New in version 1.2: added AUTOCOMMIT isolation level setting

Nullability

MSSQL has support for three levels of column nullability. The default nullability allows nulls and is explicit in the CREATE TABLE construct:

```
name VARCHAR(20) NULL
```

If `nullable=None` is specified then no specification is made. In other words the database's configured default is used. This will render:

```
name VARCHAR(20)
```

If `nullable` is `True` or `False` then the column will be `NULL` or `NOT NULL` respectively.

Date / Time Handling

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.

Large Text/Binary Type Deprecation

Per SQL Server 2012/2014 Documentation, the `NTEXT`, `TEXT` and `IMAGE` datatypes are to be removed from SQL Server in a future release. SQLAlchemy normally relates these types to the `UnicodeText`, `Text` and `LargeBinary` datatypes.

In order to accommodate this change, a new flag `deprecate_large_types` is added to the dialect, which will be automatically set based on detection of the server version in use, if not otherwise set by the user. The behavior of this flag is as follows:

- When this flag is `True`, the `UnicodeText`, `Text` and `LargeBinary` datatypes, when used to render DDL, will render the types `NVARCHAR(max)`, `VARCHAR(max)`, and `VARBINARY(max)`, respectively. This is a new behavior as of the addition of this flag.
- When this flag is `False`, the `UnicodeText`, `Text` and `LargeBinary` datatypes, when used to render DDL, will render the types `NTEXT`, `TEXT`, and `IMAGE`, respectively. This is the long-standing behavior of these types.
- The flag begins with the value `None`, before a database connection is established. If the dialect is used to render DDL without the flag being set, it is interpreted the same as `False`.
- On first connection, the dialect detects if SQL Server version 2012 or greater is in use; if the flag is still at `None`, it sets it to `True` or `False` based on whether 2012 or greater is detected.
- The flag can be set to either `True` or `False` when the dialect is created, typically via `create_engine()`:

```
eng = create_engine("mssql+pymssql://user:pass@host/db",
                    deprecate_large_types=True)
```

- Complete control over whether the "old" or "new" types are rendered is available in all SQLAlchemy versions by using the UPPERCASE type objects instead: `NVARCHAR`, `VARCHAR`, `types.VARBINARY`, `TEXT`, *`mssql.NTEXT`*, *`mssql.IMAGE`* will always remain fixed and always output exactly that type.

New in version 1.0.0.

Multipart Schema Names

SQL Server schemas sometimes require multiple parts to their "schema" qualifier, that is, including the database name and owner name as separate tokens, such as `mydatabase.dbo.some_table`. These multipart names can be set at once using the `Table.schema` argument of `Table`:

```
Table(
    "some_table", metadata,
    Column("q", String(50)),
    schema="mydatabase.dbo"
)
```

When performing operations such as table or component reflection, a schema argument that contains a dot will be split into separate "database" and "owner" components in order to correctly query the SQL Server information schema tables, as these two values are stored separately. Additionally, when rendering the schema name for DDL or SQL, the two components will be quoted separately for case sensitive names and other special characters. Given an argument as below:

```
Table(
    "some_table", metadata,
    Column("q", String(50)),
    schema="MyDataBase.dbo"
)
```

The above schema would be rendered as `[MyDataBase].dbo`, and also in reflection, would be reflected using "dbo" as the owner and "MyDataBase" as the database name.

To control how the schema name is broken into database / owner, specify brackets (which in SQL Server are quoting characters) in the name. Below, the "owner" will be considered as `MyDataBase.dbo` and the "database" will be None:

```
Table(
    "some_table", metadata,
    Column("q", String(50)),
    schema="[MyDataBase.dbo]"
)
```

To individually specify both database and owner name with special characters or embedded dots, use two sets of brackets:

```
Table(
    "some_table", metadata,
    Column("q", String(50)),
    schema="[MyDataBase.Period].[MyOwner.Dot]"
)
```

Changed in version 1.2: the SQL Server dialect now treats brackets as identifier delimeters splitting the schema into separate database and owner tokens, to allow dots within either name itself.

Legacy Schema Mode

Very old versions of the MSSQL dialect introduced the behavior such that a schema-qualified table would be auto-aliased when used in a SELECT statement; given a table:

```
account_table = Table(
    'account', metadata,
    Column('id', Integer, primary_key=True),
    Column('info', String(100)),
    schema="customer_schema"
)
```

this legacy mode of rendering would assume that "customer_schema.account" would not be accepted by all parts of the SQL statement, as illustrated below:

```
>>> eng = create_engine("mssql+pymssql://mydsn", legacy_schema_aliasing=True)
>>> print(account_table.select().compile(eng))
SELECT account_1.id, account_1.info
FROM customer_schema.account AS account_1
```

This mode of behavior is now off by default, as it appears to have served no purpose; however in the case that legacy applications rely upon it, it is available using the `legacy_schema_aliasing` argument to `create_engine()` as illustrated above.

Changed in version 1.1: the `legacy_schema_aliasing` flag introduced in version 1.0.5 to allow disabling of legacy mode for schemas now defaults to False.

Clustered Index Support

The MSSQL dialect supports clustered indexes (and primary keys) via the `mssql_clustered` option. This option is available to `Index`, `UniqueConstraint`. and `PrimaryKeyConstraint`.

To generate a clustered index:

```
Index("my_index", table.c.x, mssql_clustered=True)
```

which renders the index as `CREATE CLUSTERED INDEX my_index ON table (x)`.

To generate a clustered primary key use:

```
Table('my_table', metadata,
      Column('x', ...),
      Column('y', ...),
      PrimaryKeyConstraint("x", "y", mssql_clustered=True))
```

which will render the table, for example, as:

```
CREATE TABLE my_table (x INTEGER NOT NULL, y INTEGER NOT NULL,
                       PRIMARY KEY CLUSTERED (x, y))
```

Similarly, we can generate a clustered unique constraint using:

```
Table('my_table', metadata,
      Column('x', ...),
      Column('y', ...),
      PrimaryKeyConstraint("x"),
      UniqueConstraint("y", mssql_clustered=True),
      )
```

To explicitly request a non-clustered primary key (for example, when a separate clustered index is desired), use:

```
Table('my_table', metadata,
      Column('x', ...),
      Column('y', ...),
      PrimaryKeyConstraint("x", "y", mssql_clustered=False))
```

which will render the table, for example, as:

```
CREATE TABLE my_table (x INTEGER NOT NULL, y INTEGER NOT NULL,
                       PRIMARY KEY NONCLUSTERED (x, y))
```

Changed in version 1.1: the `mssql_clustered` option now defaults to None, rather than False. `mssql_clustered=False` now explicitly renders the NONCLUSTERED clause, whereas None omits the CLUSTERED clause entirely, allowing SQL Server defaults to take effect.

MSSQL-Specific Index Options

In addition to clustering, the MSSQL dialect supports other special options for `Index`.

INCLUDE

The `mssql_include` option renders INCLUDE(colname) for the given string names:

```
Index("my_index", table.c.x, mssql_include=['y'])
```

would render the index as `CREATE INDEX my_index ON table (x) INCLUDE (y)`

New in version 0.8.

Index ordering

Index ordering is available via functional expressions, such as:

```
Index("my_index", table.c.x.desc())
```

would render the index as `CREATE INDEX my_index ON table (x DESC)`

New in version 0.8.

See also:

schema_indexes_functional

Compatibility Levels

MSSQL supports the notion of setting compatibility levels at the database level. This allows, for instance, to run a database that is compatible 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 compatibility 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.

Triggers

SQLAlchemy by default uses OUTPUT INSERTED to get at newly generated primary key values via IDENTITY columns or other server side defaults. MS-SQL does not allow the usage of OUTPUT INSERTED on tables that have triggers. To disable the usage of OUTPUT INSERTED on a per-table basis, specify `implicit_returning=False` for each `Table` which has triggers:

```
Table('mytable', metadata,
    Column('id', Integer, primary_key=True),
    # ...,
    implicit_returning=False
)
```

Declarative form:

```
class MyClass(Base):
    # ...
    __table_args__ = {'implicit_returning':False}
```

This option can also be specified engine-wide using the `implicit_returning=False` argument on `create_engine()`.

Rowcount Support / ORM Versioning

The SQL Server drivers may have limited ability to return the number of rows updated from an UPDATE or DELETE statement.

As of this writing, the PyODBC driver is not able to return a rowcount when OUTPUT INSERTED is used. This impacts the SQLAlchemy ORM's versioning feature in many cases where server-side value generators are in use in that while the versioning operations can succeed, the ORM cannot always check that an UPDATE or DELETE statement matched the number of rows expected, which is how it verifies that the version identifier matched. When this condition occurs, a warning will be emitted but the operation will proceed.

The use of OUTPUT INSERTED can be disabled by setting the `Table.implicit_returning` flag to `False` on a particular `Table`, which in declarative looks like:

```
class MyTable(Base):
    __tablename__ = 'mytable'
    id = Column(Integer, primary_key=True)
    stuff = Column(String(10))
    timestamp = Column(TIMESTAMP(), default=text('DEFAULT'))
    __mapper_args__ = {
```

```
        'version_id_col': timestamp,
        'version_id_generator': False,
    }
    __table_args__ = {
        'implicit_returning': False
    }
```

Enabling Snapshot Isolation

SQL Server has a default transaction isolation mode that locks entire tables, and causes even mildly concurrent applications to have long held locks and frequent deadlocks. Enabling snapshot isolation for the database as a whole is recommended for modern levels of concurrency support. This is accomplished via the following ALTER DATABASE commands executed at the SQL prompt:

```
ALTER DATABASE MyDatabase SET ALLOW_SNAPSHOT_ISOLATION ON

ALTER DATABASE MyDatabase SET READ_COMMITTED_SNAPSHOT ON
```

Background on SQL Server snapshot isolation is available at http://msdn.microsoft.com/en-us/library/ms175095.aspx.

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.**BIT**

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.mssql.**CHAR**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
The SQL CHAR type.

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. 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 if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.

- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the `String` type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to `create_engine()` as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the `Unicode` or `UnicodeText` types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

  For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **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 a last resort for reading strings from a column with varied or corrupted encodings.

class sqlalchemy.dialects.mssql.**DATETIME2**(*precision=None*, ***kw*)

class sqlalchemy.dialects.mssql.**DATETIMEOFFSET**(*precision=None*, ***kwargs*)

class sqlalchemy.dialects.mssql.**IMAGE**(*length=None*)

__init__(*length=None*)
Construct a LargeBinary type.

Parameters **length** – optional, a length for the column for use in DDL statements, for those binary types that accept a length, such as the MySQL BLOB type.

class sqlalchemy.dialects.mssql.**MONEY**

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.mssql.**NCHAR**(*length=None*, ***kwargs*)
The SQL NCHAR type.

__init__(*length=None*, ***kwargs*)
Create a `Unicode` object.

Parameters are the same as that of `String`, with the exception that `convert_unicode` defaults to `True`.

class sqlalchemy.dialects.mssql.**NTEXT**(*length=None*, ***kwargs*)
MSSQL NTEXT type, for variable-length unicode text up to 2^30 characters.

__init__(*length=None*, ***kwargs*)
Create a Unicode-converting Text type.

Parameters are the same as that of `Text`, with the exception that `convert_unicode` defaults to `True`.

class sqlalchemy.dialects.mssql.**NVARCHAR**(*length=None*, ***kwargs*)
The SQL NVARCHAR type.

__init__(*length=None*, ***kwargs*)
Create a `Unicode` object.

Parameters are the same as that of `String`, with the exception that `convert_unicode` defaults to `True`.

class sqlalchemy.dialects.mssql.**REAL**(***kw*)

class sqlalchemy.dialects.mssql.**ROWVERSION**(*convert_int=False*)
Implement the SQL Server ROWVERSION type.

The ROWVERSION datatype is a SQL Server synonym for the TIMESTAMP datatype, however current SQL Server documentation suggests using ROWVERSION for new datatypes going forward.

The ROWVERSION datatype does **not** reflect (e.g. introspect) from the database as itself; the returned datatype will be *mssql.TIMESTAMP*.

This is a read-only datatype that does not support INSERT of values.

New in version 1.2.

See also:

mssql.TIMESTAMP

__init__(*convert_int=False*)
Construct a TIMESTAMP or ROWVERSION type.

Parameters **convert_int** – if True, binary integer values will be converted to integers on read.

New in version 1.2.

class sqlalchemy.dialects.mssql.**SMALLDATETIME**(*timezone=False*)

__init__(*timezone=False*)
Construct a new `DateTime`.

Parameters **timezone** – boolean. Indicates that the datetime type should enable timezone support, if available on the **base date/time-holding type only**. It is recommended to make use of the `TIMESTAMP` datatype directly when using this flag, as some databases include separate generic date/time-holding types distinct from the timezone-capable TIMESTAMP datatype, such as Oracle.

class sqlalchemy.dialects.mssql.**SMALLMONEY**

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.mssql.**SQL_VARIANT**

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.mssql.TEXT(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
The SQL TEXT type.

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. 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 if a VARCHAR with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.
- **convert_unicode** – When set to True, the String type will assume that input is to be passed as Python unicode objects, and results returned as Python unicode objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the encoding parameter passed to create_engine() as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the Unicode or UnicodeText types should be used regardless, which feature the same behavior of convert_unicode but also indicate an underlying column type that directly supports unicode, such as NVARCHAR.

  For the extremely rare case that Python unicode is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python unicode, the value force can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.
- **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 a last resort for reading strings from a column with varied or corrupted encodings.

class sqlalchemy.dialects.mssql.TIME(*precision=None*, ***kwargs*)

class sqlalchemy.dialects.mssql.TIMESTAMP(*convert_int=False*)
Implement the SQL Server TIMESTAMP type.

Note this is **completely different** than the SQL Standard TIMESTAMP type, which is not supported by SQL Server. It is a read-only datatype that does not support INSERT of values.

New in version 1.2.

See also:

mssql.ROWVERSION

`__init__`(*convert_int=False*)
Construct a TIMESTAMP or ROWVERSION type.

Parameters `convert_int` – if True, binary integer values will be converted to integers on read.

New in version 1.2.

`class sqlalchemy.dialects.mssql.TINYINT`

`__init__`
Initialize self. See help(type(self)) for accurate signature.

`class sqlalchemy.dialects.mssql.UNIQUEIDENTIFIER`

`__init__`
Initialize self. See help(type(self)) for accurate signature.

`class sqlalchemy.dialects.mssql.VARCHAR`(*length=None, collation=None, convert_unicode=False, unicode_error=None, _warn_on_bytestring=False*)
The SQL VARCHAR type.

`__init__`(*length=None, collation=None, convert_unicode=False, unicode_error=None, _warn_on_bytestring=False*)
Create a string-holding type.

Parameters

- `length` – optional, a length for the column for use in DDL and CAST expressions. 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 if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- `collation` – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.
- `convert_unicode` – When set to `True`, the `String` type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to `create_engine()` as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the `Unicode` or `UnicodeText`

types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **`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 a last resort for reading strings from a column with varied or corrupted encodings.

class sqlalchemy.dialects.mssql.**XML**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

MSSQL XML type.

This is a placeholder type for reflection purposes that does not include any Python-side datatype support. It also does not currently support additional arguments, such as "CONTENT", "DOCUMENT", "xml_schema_collection".

New in version 1.1.11.

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **`length`** – optional, a length for the column for use in DDL and CAST expressions. 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 if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **`collation`** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **`convert_unicode`** – When set to `True`, the `String` type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to `create_engine()` as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the `Unicode` or `UnicodeText` types should be used regardless, which feature the same behavior of

convert_unicode but also indicate an underlying column type that directly supports unicode, such as NVARCHAR.

For the extremely rare case that Python unicode is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python unicode, the value force can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- 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 a last resort for reading strings from a column with varied or corrupted encodings.

PyODBC

Connecting to PyODBC

The URL here is to be translated to PyODBC connection strings, as detailed in ConnectionStrings.

DSN Connections

A DSN-based connection is **preferred** overall when using ODBC. A basic DSN-based connection looks like:

```
engine = create_engine("mssql+pyodbc://scott:tiger@some_dsn")
```

Which above, will pass the following connection string to PyODBC:

```
dsn=mydsn;UID=user;PWD=pass
```

If the username and password are omitted, the DSN form will also add the Trusted_Connection=yes directive to the ODBC string.

Hostname Connections

Hostname-based connections are **not preferred**, however are supported. The ODBC driver name must be explicitly specified:

```
engine = create_engine("mssql+pyodbc://scott:tiger@myhost:port/databasename?
↪driver=SQL+Server+Native+Client+10.0")
```

Changed in version 1.0.0: Hostname-based PyODBC connections now require the SQL Server driver name specified explicitly. SQLAlchemy cannot choose an optimal default here as it varies based on platform and installed drivers.

Other keywords interpreted by the Pyodbc dialect to be passed to pyodbc.connect() in both the DSN and hostname cases include: odbc_autotranslate, ansi, unicode_results, autocommit.

Pass through exact Pyodbc string

A PyODBC connection string can also be sent exactly as specified in ConnectionStrings into the driver using the parameter `odbc_connect`. The delimeters must be URL escaped, however, as illustrated below using `urllib.quote_plus`:

```
import urllib
params = urllib.quote_plus("DRIVER={SQL Server Native Client 10.0};SERVER=dagger;DATABASE=test;
↪UID=user;PWD=password")

engine = create_engine("mssql+pyodbc:///?odbc_connect=%s" % params)
```

Driver / Unicode Support

PyODBC works best with Microsoft ODBC drivers, particularly in the area of Unicode support on both Python 2 and Python 3.

Using the FreeTDS ODBC drivers on Linux or OSX with PyODBC is **not** recommended; there have been historically many Unicode-related issues in this area, including before Microsoft offered ODBC drivers for Linux and OSX. Now that Microsoft offers drivers for all platforms, for PyODBC support these are recommended. FreeTDS remains relevant for non-ODBC drivers such as pymssql where it works very well.

Rowcount Support

Pyodbc only has partial support for rowcount. See the notes at *Rowcount Support / ORM Versioning* for important notes when using ORM versioning.

mxODBC

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 unconditionally use string-escaped parameters.

pymssql

pymssql is a Python module that provides a Python DBAPI interface around FreeTDS. Compatible builds are available for Linux, MacOSX and Windows platforms.

Modern versions of this driver work very well with SQL Server and FreeTDS from Linux and is highly recommended.

zxjdbc

AdoDBAPI

Note: The adodbapi dialect is not implemented SQLAlchemy versions 0.6 and above at this time.

4.1.3 MySQL

Supported Versions and Features

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

See the official MySQL documentation for detailed information about features supported in any given server release.

Connection Timeouts and Disconnects

MySQL features an automatic connection close behavior, for connections that have been idle for a fixed period of time, defaulting to eight hours. To circumvent having this issue, use the `create_engine.pool_recycle` option which ensures that a connection will be discarded and replaced with a new one if it has been present in the pool for a fixed number of seconds:

```
engine = create_engine('mysql+mysqldb://...', pool_recycle=3600)
```

For more comprehensive disconnect detection of pooled connections, including accommodation of server restarts and network issues, a pre-ping approach may be employed. See pool_disconnects for current approaches.

See also:

pool_disconnects - Background on several techniques for dealing with timed out connections as well as database restarts.

CREATE TABLE arguments including Storage Engines

MySQL's CREATE TABLE syntax includes a wide array of special options, including `ENGINE`, `CHARSET`, `MAX_ROWS`, `ROW_FORMAT`, `INSERT_METHOD`, and many more. To accommodate the rendering of these arguments, specify the form `mysql_argument_name="value"`. For example, to specify a table with `ENGINE` of `InnoDB`, `CHARSET` of `utf8`, and `KEY_BLOCK_SIZE` of `1024`:

```
Table('mytable', metadata,
      Column('data', String(32)),
      mysql_engine='InnoDB',
      mysql_charset='utf8',
      mysql_key_block_size="1024"
     )
```

The MySQL dialect will normally transfer any keyword specified as `mysql_keyword_name` to be rendered as `KEYWORD_NAME` in the `CREATE TABLE` statement. A handful of these names will render with a space instead of an underscore; to support this, the MySQL dialect has awareness of these particular names, which

include DATA DIRECTORY (e.g. mysql_data_directory), CHARACTER SET (e.g. mysql_character_set) and INDEX DIRECTORY (e.g. mysql_index_directory).

The most common argument is mysql_engine, which refers to the storage engine for the table. Historically, MySQL server installations would default to MyISAM for this value, although newer versions may be defaulting to InnoDB. The InnoDB engine is typically preferred for its support of transactions and foreign keys.

A Table that is created in a MySQL database with a storage engine of MyISAM will be essentially non-transactional, meaning any INSERT/UPDATE/DELETE statement referring to this table will be invoked as autocommit. It also will have no support for foreign key constraints; while the CREATE TABLE statement accepts foreign key options, when using the MyISAM storage engine these arguments are discarded. Reflecting such a table will also produce no foreign key constraint information.

For fully atomic transactions as well as support for foreign key constraints, all participating CREATE TABLE statements must specify a transactional engine, which in the vast majority of cases is InnoDB.

See also:

The InnoDB Storage Engine - on the MySQL website.

Case Sensitivity and Table Reflection

MySQL has inconsistent support for case-sensitive identifier names, basing support on specific details of the underlying operating system. However, it has been observed that no matter what case sensitivity behavior is present, the names of tables in foreign key declarations are *always* received from the database as all-lower case, making it impossible to accurately reflect a schema where inter-related tables use mixed-case identifier names.

Therefore it is strongly advised that table names be declared as all lower case both within SQLAlchemy as well as on the MySQL database itself, especially if database reflection features are to be used.

Transaction Isolation Level

All MySQL dialects support setting of transaction isolation level both via a dialect-specific parameter create_engine.isolation_level accepted by create_engine(), as well as the Connection.execution_options.isolation_level argument as passed to Connection.execution_options(). This feature works by issuing the command SET SESSION TRANSACTION ISOLATION LEVEL <level> for each new connection. For the special AUTOCOMMIT isolation level, DBAPI-specific techniques are used.

To set isolation level using create_engine():

```
engine = create_engine(
                "mysql://scott:tiger@localhost/test",
                isolation_level="READ UNCOMMITTED"
            )
```

To set using per-connection execution options:

```
connection = engine.connect()
connection = connection.execution_options(
    isolation_level="READ COMMITTED"
)
```

Valid values for isolation_level include:

- READ COMMITTED
- READ UNCOMMITTED
- REPEATABLE READ

- SERIALIZABLE
- AUTOCOMMIT

The special AUTOCOMMIT value makes use of the various "autocommit" attributes provided by specific DBAPIs, and is currently supported by MySQLdb, MySQL-Client, MySQL-Connector Python, and PyMySQL. Using it, the MySQL connection will return true for the value of SELECT @@autocommit;.

New in version 1.1: - added support for the AUTOCOMMIT isolation level.

AUTO_INCREMENT Behavior

When creating tables, SQLAlchemy will automatically set AUTO_INCREMENT on the first Integer primary key column which is not marked as a foreign key:

```
>>> 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 passing False to the autoincrement argument of 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)
     )
```

Server Side Cursors

Server-side cursor support is available for the MySQLdb and PyMySQL dialects. From a MySQL point of view this means that the MySQLdb.cursors.SSCursor or pymysql.cursors.SSCursor class is used when building up the cursor which will receive results. The most typical way of invoking this feature is via the Connection.execution_options.stream_results connection execution option. Server side cursors can also be enabled for all SELECT statements unconditionally by passing server_side_cursors=True to create_engine().

New in version 1.1.4: - added server-side cursor support.

Unicode

Charset Selection

Most MySQL DBAPIs offer the option to set the client character set for a connection. This is typically delivered using the charset parameter in the URL, such as:

```
e = create_engine(
    "mysql+pymysql://scott:tiger@localhost/test?charset=utf8")
```

This charset is the **client character set** for the connection. Some MySQL DBAPIs will default this to a value such as latin1, and some will make use of the default-character-set setting in the my.cnf file as well. Documentation for the DBAPI in use should be consulted for specific behavior.

The encoding used for Unicode has traditionally been 'utf8'. However, for MySQL versions 5.5.3 on forward, a new MySQL-specific encoding 'utf8mb4' has been introduced. The rationale for this new

encoding is due to the fact that MySQL's utf-8 encoding only supports codepoints up to three bytes instead of four. Therefore, when communicating with a MySQL database that includes codepoints more than three bytes in size, this new charset is preferred, if supported by both the database as well as the client DBAPI, as in:

```
e = create_engine(
    "mysql+pymysql://scott:tiger@localhost/test?charset=utf8mb4")
```

At the moment, up-to-date versions of MySQLdb and PyMySQL support the `utf8mb4` charset. Other DBAPIs such as MySQL-Connector and OurSQL may **not** support it as of yet.

In order to use `utf8mb4` encoding, changes to the MySQL schema and/or server configuration may be required.

See also:

The utf8mb4 Character Set - in the MySQL documentation

Unicode Encoding / Decoding

All modern MySQL DBAPIs all offer the service of handling the encoding and decoding of unicode data between the Python application space and the database. As this was not always the case, SQLAlchemy also includes a comprehensive system of performing the encode/decode task as well. As only one of these systems should be in use at at time, SQLAlchemy has long included functionality to automatically detect upon first connection whether or not the DBAPI is automatically handling unicode.

Whether or not the MySQL DBAPI will handle encoding can usually be configured using a DBAPI flag `use_unicode`, which is known to be supported at least by MySQLdb, PyMySQL, and MySQL-Connector. Setting this value to `0` in the "connect args" or query string will have the effect of disabling the DBAPI's handling of unicode, such that it instead will return data of the `str` type or `bytes` type, with data in the configured charset:

```
# connect while disabling the DBAPI's unicode encoding/decoding
e = create_engine(
    "mysql+mysqldb://scott:tiger@localhost/test?charset=utf8&use_unicode=0")
```

Current recommendations for modern DBAPIs are as follows:

- It is generally always safe to leave the `use_unicode` flag set at its default; that is, don't use it at all.
- Under Python 3, the `use_unicode=0` flag should **never be used**. SQLAlchemy under Python 3 generally assumes the DBAPI receives and returns string values as Python 3 strings, which are inherently unicode objects.
- Under Python 2 with MySQLdb, the `use_unicode=0` flag will **offer superior performance**, as MySQLdb's unicode converters under Python 2 only have been observed to have unusually slow performance compared to SQLAlchemy's fast C-based encoders/decoders.

In short: don't specify `use_unicode` *at all*, with the possible exception of `use_unicode=0` on MySQLdb with Python 2 **only** for a potential performance gain.

Ansi Quoting Style

MySQL features two varieties of identifier "quoting style", one using backticks and the other using quotes, e.g. `` `some_identifier` `` vs. `"some_identifier"`. All MySQL dialects detect which version is in use by checking the value of `sql_mode` when a connection is first established with a particular `Engine`. This quoting style comes into play when rendering table and column names as well as when reflecting existing database structures. The detection is entirely automatic and no special configuration is needed to use either quoting style.

Changed in version 0.6: detection of ANSI quoting style is entirely automatic, there's no longer any end-user `create_engine()` options in this regard.

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.

- INSERT..ON DUPLICATE KEY UPDATE: See *INSERT…ON DUPLICATE KEY UPDATE (Upsert)*
- SELECT pragma:

  ```
  select(..., prefixes=['HIGH_PRIORITY', 'SQL_SMALL_RESULT'])
  ```

- UPDATE with LIMIT:

  ```
  update(..., mysql_limit=10)
  ```

INSERT…ON DUPLICATE KEY UPDATE (Upsert)

MySQL allows "upserts" (update or insert) of rows into a table via the `ON DUPLICATE KEY UPDATE` clause of the `INSERT` statement. A candidate row will only be inserted if that row does not match an existing primary or unique key in the table; otherwise, an UPDATE will be performed. The statement allows for separate specification of the values to INSERT versus the values for UPDATE.

SQLAlchemy provides `ON DUPLICATE KEY UPDATE` support via the MySQL-specific *mysql.dml.insert()* function, which provides the generative method *on_duplicate_key_update()*:

```
from sqlalchemy.dialects.mysql import insert

insert_stmt = insert(my_table).values(
    id='some_existing_id',
    data='inserted value')

on_duplicate_key_stmt = insert_stmt.on_duplicate_key_update(
    data=insert_stmt.inserted.data,
    status='U'
)

conn.execute(on_duplicate_key_stmt)
```

Unlike Postgresql's "ON CONFLICT" phrase, the "ON DUPLICATE KEY UPDATE" phrase will always match on any primary key or unique key, and will always perform an UPDATE if there's a match; there are no options for it to raise an error or to skip performing an UPDATE.

`ON DUPLICATE KEY UPDATE` is used to perform an update of the already existing row, using any combination of new values as well as values from the proposed insertion. These values are specified using keyword arguments passed to the *on_duplicate_key_update()* given column key values (usually the name of the column, unless it specifies `Column.key`) as keys and literal or SQL expressions as values:

```
on_duplicate_key_stmt = insert_stmt.on_duplicate_key_update(
    data="some data"
```

```
    updated_at=func.current_timestamp()
)
```

Warning: The *Insert.on_duplicate_key_update()* method does **not** take into account Python-side default UPDATE values or generation functions, e.g. e.g. those specified using `Column.onupdate`. These values will not be exercised for an ON DUPLICATE KEY style of UPDATE, unless they are manually specified explicitly in the parameters.

In order to refer to the proposed insertion row, the special alias *inserted* is available as an attribute on the *mysql.dml.Insert* object; this object is a `ColumnCollection` which contains all columns of the target table:

```
from sqlalchemy.dialects.mysql import insert

stmt = insert(my_table).values(
    id='some_id',
    data='inserted value',
    author='jlh')
do_update_stmt = stmt.on_duplicate_key_update(
    data="updated value",
    author=stmt.inserted.author
)
conn.execute(do_update_stmt)
```

When rendered, the "inserted" namespace will produce the expression `VALUES(<columnname>)`.

New in version 1.2: Added support for MySQL ON DUPLICATE KEY UPDATE clause

rowcount Support

SQLAlchemy standardizes the DBAPI `cursor.rowcount` attribute to be the usual definition of "number of rows matched by an UPDATE or DELETE" statement. This is in contradiction to the default setting on most MySQL DBAPI drivers, which is "number of rows actually modified/deleted". For this reason, the SQLAlchemy MySQL dialects always add the `constants.CLIENT.FOUND_ROWS` flag, or whatever is equivalent for the target dialect, upon connection. This setting is currently hardcoded.

See also:

`ResultProxy.rowcount`

CAST Support

MySQL documents the CAST operator as available in version 4.0.2. When using the SQLAlchemy `cast()` function, SQLAlchemy will not render the CAST token on MySQL before this version, based on server version detection, instead rendering the internal expression directly.

CAST may still not be desirable on an early MySQL version post-4.0.2, as it didn't add all datatype support until 4.1.1. If your application falls into this narrow area, the behavior of CAST can be controlled using the sqlalchemy.ext.compiler_toplevel system, as per the recipe below:

```
from sqlalchemy.sql.expression import Cast
from sqlalchemy.ext.compiler import compiles

@compiles(Cast, 'mysql')
def _check_mysql_version(element, compiler, **kw):
    if compiler.dialect.server_version_info < (4, 1, 0):
        return compiler.process(element.clause, **kw)
```

```
    else:
        return compiler.visit_cast(element, **kw)
```

The above function, which only needs to be declared once within an application, overrides the compilation of the `cast()` construct to check for version 4.1.0 before fully rendering CAST; else the internal element of the construct is rendered directly.

MySQL Specific Index Options

MySQL-specific extensions to the `Index` construct are available.

Index Length

MySQL provides an option to create index entries with a certain length, where "length" refers to the number of characters or bytes in each value which will become part of the index. SQLAlchemy provides this feature via the `mysql_length` parameter:

```
Index('my_index', my_table.c.data, mysql_length=10)

Index('a_b_idx', my_table.c.a, my_table.c.b, mysql_length={'a': 4,
                                                           'b': 9})
```

Prefix lengths are given in characters for nonbinary string types and in bytes for binary string types. The value passed to the keyword argument *must* be either an integer (and, thus, specify the same prefix length value for all columns of the index) or a dict in which keys are column names and values are prefix length values for corresponding columns. MySQL only allows a length for a column of an index if it is for a CHAR, VARCHAR, TEXT, BINARY, VARBINARY and BLOB.

New in version 0.8.2: `mysql_length` may now be specified as a dictionary for use with composite indexes.

Index Prefixes

MySQL storage engines permit you to specify an index prefix when creating an index. SQLAlchemy provides this feature via the `mysql_prefix` parameter on `Index`:

```
Index('my_index', my_table.c.data, mysql_prefix='FULLTEXT')
```

The value passed to the keyword argument will be simply passed through to the underlying CREATE INDEX, so it *must* be a valid index prefix for your MySQL storage engine.

New in version 1.1.5.

See also:

CREATE INDEX - MySQL documentation

Index Types

Some MySQL storage engines permit you to specify an index type when creating an index or primary key constraint. SQLAlchemy provides this feature via the `mysql_using` parameter on `Index`:

```
Index('my_index', my_table.c.data, mysql_using='hash')
```

As well as the `mysql_using` parameter on `PrimaryKeyConstraint`:

```
PrimaryKeyConstraint("data", mysql_using='hash')
```

The value passed to the keyword argument will be simply passed through to the underlying CREATE INDEX or PRIMARY KEY clause, so it *must* be a valid index type for your MySQL storage engine.

More information can be found at:

http://dev.mysql.com/doc/refman/5.0/en/create-index.html

http://dev.mysql.com/doc/refman/5.0/en/create-table.html

MySQL Foreign Keys

MySQL's behavior regarding foreign keys has some important caveats.

Foreign Key Arguments to Avoid

MySQL does not support the foreign key arguments "DEFERRABLE", "INITIALLY", or "MATCH". Using the `deferrable` or `initially` keyword argument with `ForeignKeyConstraint` or `ForeignKey` will have the effect of these keywords being rendered in a DDL expression, which will then raise an error on MySQL. In order to use these keywords on a foreign key while having them ignored on a MySQL backend, use a custom compile rule:

```
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.schema import ForeignKeyConstraint

@compiles(ForeignKeyConstraint, "mysql")
def process(element, compiler, **kw):
    element.deferrable = element.initially = None
    return compiler.visit_foreign_key_constraint(element, **kw)
```

Changed in version 0.9.0: - the MySQL backend no longer silently ignores the `deferrable` or `initially` keyword arguments of `ForeignKeyConstraint` and `ForeignKey`.

The "MATCH" keyword is in fact more insidious, and is explicitly disallowed by SQLAlchemy in conjunction with the MySQL backend. This argument is silently ignored by MySQL, but in addition has the effect of ON UPDATE and ON DELETE options also being ignored by the backend. Therefore MATCH should never be used with the MySQL backend; as is the case with DEFERRABLE and INITIALLY, custom compilation rules can be used to correct a MySQL ForeignKeyConstraint at DDL definition time.

New in version 0.9.0: - the MySQL backend will raise a `CompileError` when the `match` keyword is used with `ForeignKeyConstraint` or `ForeignKey`.

Reflection of Foreign Key Constraints

Not all MySQL storage engines support foreign keys. When using the very common `MyISAM` MySQL storage engine, 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
     )
```

See also:

CREATE TABLE arguments including Storage Engines

MySQL Unique Constraints and Reflection

SQLAlchemy supports both the `Index` construct with the flag `unique=True`, indicating a UNIQUE index, as well as the `UniqueConstraint` construct, representing a UNIQUE constraint. Both objects/syntaxes are supported by MySQL when emitting DDL to create these constraints. However, MySQL does not have a unique constraint construct that is separate from a unique index; that is, the "UNIQUE" constraint on MySQL is equivalent to creating a "UNIQUE INDEX".

When reflecting these constructs, the `Inspector.get_indexes()` and the `Inspector.get_unique_constraints()` methods will **both** return an entry for a UNIQUE index in MySQL. However, when performing full table reflection using `Table(..., autoload=True)`, the `UniqueConstraint` construct is **not** part of the fully reflected `Table` construct under any circumstances; this construct is always represented by a `Index` with the `unique=True` setting present in the `Table.indexes` collection.

TIMESTAMP Columns and NULL

MySQL historically enforces that a column which specifies the TIMESTAMP datatype implicitly includes a default value of CURRENT_TIMESTAMP, even though this is not stated, and additionally sets the column as NOT NULL, the opposite behavior vs. that of all other datatypes:

```
mysql> CREATE TABLE ts_test (
    -> a INTEGER,
    -> b INTEGER NOT NULL,
    -> c TIMESTAMP,
    -> d TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -> e TIMESTAMP NULL);
Query OK, 0 rows affected (0.03 sec)

mysql> SHOW CREATE TABLE ts_test;
+---------+---------------------------------------------------
| Table   | Create Table
+---------+---------------------------------------------------
| ts_test | CREATE TABLE `ts_test` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) NOT NULL,
  `c` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `d` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `e` timestamp NULL DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

Above, we see that an INTEGER column defaults to NULL, unless it is specified with NOT NULL. But when the column is of type TIMESTAMP, an implicit default of CURRENT_TIMESTAMP is generated which also coerces the column to be a NOT NULL, even though we did not specify it as such.

This behavior of MySQL can be changed on the MySQL side using the explicit_defaults_for_timestamp configuration flag introduced in MySQL 5.6. With this server setting enabled, TIMESTAMP columns behave like any other datatype on the MySQL side with regards to defaults and nullability.

However, to accommodate the vast majority of MySQL databases that do not specify this new flag, SQLAlchemy emits the "NULL" specifier explicitly with any TIMESTAMP column that does not specify `nullable=False`. In order to accommodate newer databases that specify `explicit_defaults_for_timestamp`, SQLAlchemy also emits NOT NULL for TIMESTAMP columns that do specify `nullable=False`. The following example illustrates:

```
from sqlalchemy import MetaData, Integer, Table, Column, text
from sqlalchemy.dialects.mysql import TIMESTAMP

m = MetaData()
t = Table('ts_test', m,
        Column('a', Integer),
        Column('b', Integer, nullable=False),
```

```
        Column('c', TIMESTAMP),
        Column('d', TIMESTAMP, nullable=False)
    )


from sqlalchemy import create_engine
e = create_engine("mysql://scott:tiger@localhost/test", echo=True)
m.create_all(e)
```

output:

```
CREATE TABLE ts_test (
    a INTEGER,
    b INTEGER NOT NULL,
    c TIMESTAMP NULL,
    d TIMESTAMP NOT NULL
)
```

Changed in version 1.0.0: - SQLAlchemy now renders NULL or NOT NULL in all cases for TIMESTAMP columns, to accommodate `explicit_defaults_for_timestamp`. Prior to this version, it will not render "NOT NULL" for a TIMESTAMP column that is `nullable=False`.

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.**BIGINT**(*display_width=None*, ***kw*)
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.**BINARY**(*length=None*)
The SQL BINARY type.

class sqlalchemy.dialects.mysql.**BIT**(*length=None*)
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.BLOB(length=None)`
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 binary types that accept a length, such as the MySQL BLOB type.

`class sqlalchemy.dialects.mysql.BOOLEAN(create_constraint=True, name=None, _create_events=True)`
The SQL BOOLEAN type.

`__init__(create_constraint=True, name=None, _create_events=True)`
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.CHAR(length=None, **kwargs)`
MySQL CHAR type, for fixed-length character data.

`__init__(length=None, **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.DATE`
The SQL DATE type.

`__init__`
Initialize self. See help(type(self)) for accurate signature.

`class sqlalchemy.dialects.mysql.DATETIME(timezone=False, fsp=None)`
MySQL DATETIME type.

`__init__(timezone=False, fsp=None)`
Construct a MySQL DATETIME type.

Parameters

- `timezone` – not used by the MySQL dialect.
- `fsp` – fractional seconds precision value. MySQL 5.6.4 supports storage of fractional seconds; this parameter will be used when emitting DDL for the DATETIME type.

Note: DBAPI driver support for fractional seconds may be limited; current support includes MySQL Connector/Python.

New in version 0.8.5: Added MySQL-specific *mysql.DATETIME* with fractional seconds support.

class sqlalchemy.dialects.mysql.**DECIMAL**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

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.**DOUBLE**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

MySQL DOUBLE type.

__init__(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Construct a DOUBLE.

Note: The *DOUBLE* type by default converts from float to Decimal, using a truncation that defaults to 10 digits. Specify either `scale=n` or `decimal_return_scale=n` in order to change this scale, or `asdecimal=False` to return values directly as Python floating points.

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.**ENUM**(**enums*, ***kw*)

MySQL ENUM type.

__init__(**enums*, ***kw*)

Construct an ENUM.

E.g.:

```
Column('myenum', ENUM("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). This object may also be a PEP-435-compliant enumerated type.
- **strict** – This flag has no effect.

Changed in version The: MySQL ENUM type as well as the base Enum type now validates all Python data values.

- **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.**FLOAT**(*precision=None*, *scale=None*, *asdecimal=False*, ***kw*)

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.**INTEGER**(*display_width=None*, ***kw*)

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.JSON(*none_as_null=False*)
MySQL JSON type.

MySQL supports JSON as of version 5.7. Note that MariaDB does **not** support JSON at the time of this writing.

The *mysql.JSON* type supports persistence of JSON values as well as the core index operations provided by `types.JSON` datatype, by adapting the operations to render the `JSON_EXTRACT` function at the database level.

New in version 1.1.

class sqlalchemy.dialects.mysql.LONGBLOB(*length=None*)
MySQL LONGBLOB type, for binary data up to 2^32 bytes.

class sqlalchemy.dialects.mysql.LONGTEXT(***kwargs*)
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.MEDIUMBLOB(*length=None*)
MySQL MEDIUMBLOB type, for binary data up to 2^24 bytes.

class sqlalchemy.dialects.mysql.MEDIUMINT(*display_width=None*, ***kw*)
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.MEDIUMTEXT(***kwargs*)
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.**NCHAR**(*length=None*, ***kwargs*)

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.**NUMERIC**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

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.**NVARCHAR**(*length=None*, ***kwargs*)

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.**REAL**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)
MySQL REAL type.

__init__(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)
Construct a REAL.

Note: The *REAL* type by default converts from float to Decimal, using a truncation that defaults to 10 digits. Specify either `scale=n` or `decimal_return_scale=n` in order to change this scale, or `asdecimal=False` to return values directly as Python floating points.

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.**SET**(**values*, ***kw*)
MySQL SET type.

__init__(**values*, ***kw*)
Construct a SET.

E.g.:

```
Column('myset', SET("foo", "bar", "baz"))
```

The list of potential values is required in the case that this set will be used to generate DDL for a table, or if the *SET.retrieve_as_bitwise* flag is set to True.

Parameters

- **values** – The range of valid values for this SET.
- **convert_unicode** – Same flag as that of `String.convert_unicode`.
- **collation** – same as that of `String.collation`
- **charset** – same as that of *VARCHAR.charset*.
- **ascii** – same as that of *VARCHAR.ascii*.
- **unicode** – same as that of *VARCHAR.unicode*.
- **binary** – same as that of *VARCHAR.binary*.
- **quoting** – Defaults to 'auto': automatically determine set value quoting. If all 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.

New in version 0.9.0.

- **retrieve_as_bitwise** – if True, the data for the set type will be persisted and selected using an integer value, where a set is coerced into a bitwise mask for persistence. MySQL allows this mode which has the advantage of being able to store values unambiguously, such as the blank string `''`. The datatype will appear as the expression `col + 0` in a SELECT statement, so that the value is coerced into an integer value in result sets. This flag is required if one wishes to persist a set that can store the blank string `''` as a value.

  Warning: When using *mysql.SET.retrieve_as_bitwise*, it is essential that the list of set values is expressed in the **exact same order** as exists on the MySQL database.

  New in version 1.0.0.

class sqlalchemy.dialects.mysql.**SMALLINT**(*display_width=None*, ***kw*)
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.**TEXT**(*length=None*, ***kw*)
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.**TIME**(*timezone=False*, *fsp=None*)
MySQL TIME type.

__init__(*timezone=False*, *fsp=None*)
Construct a MySQL TIME type.

Parameters

- **timezone** – not used by the MySQL dialect.
- **fsp** – fractional seconds precision value. MySQL 5.6 supports storage of fractional seconds; this parameter will be used when emitting DDL for the TIME type.

Note: DBAPI driver support for fractional seconds may be limited; current support includes MySQL Connector/Python.

New in version 0.8: The MySQL-specific TIME type as well as fractional seconds support.

class sqlalchemy.dialects.mysql.**TIMESTAMP**(*timezone=False*, *fsp=None*)
MySQL TIMESTAMP type.

__init__(*timezone=False*, *fsp=None*)
Construct a MySQL TIMESTAMP type.

Parameters

- **timezone** – not used by the MySQL dialect.
- **fsp** – fractional seconds precision value. MySQL 5.6.4 supports storage of fractional seconds; this parameter will be used when emitting DDL for the TIMESTAMP type.

Note: DBAPI driver support for fractional seconds may be limited; current support includes MySQL Connector/Python.

New in version 0.8.5: Added MySQL-specific *mysql.TIMESTAMP* with fractional seconds support.

class sqlalchemy.dialects.mysql.**TINYBLOB**(*length=None*)
MySQL TINYBLOB type, for binary data up to 2^8 bytes.

class sqlalchemy.dialects.mysql.**TINYINT**(*display_width=None*, ***kw*)
MySQL TINYINT type.

__init__(*display_width=None*, ***kw*)
Construct a TINYINT.

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.**TINYTEXT**(***kwargs*)
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.VARBINARY(*length=None*)
The SQL VARBINARY type.

class sqlalchemy.dialects.mysql.VARCHAR(*length=None*, ***kwargs*)
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.YEAR(*display_width=None*)
MySQL YEAR type, for single byte storage of years 1901-2155.

MySQL DML Constructs

sqlalchemy.dialects.mysql.dml.insert(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)
Construct a new *Insert* object.

This constructor is mirrored as a public API function; see `insert()` for a full usage and argument description.

`class sqlalchemy.dialects.mysql.dml.Insert(`*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*`)`

MySQL-specific implementation of INSERT.

Adds methods for MySQL-specific syntaxes such as ON DUPLICATE KEY UPDATE.

New in version 1.2.

`inserted`

Provide the "inserted" namespace for an ON DUPLICATE KEY UPDATE statement

MySQL's ON DUPLICATE KEY UPDATE clause allows reference to the row that would be inserted, via a special function called `VALUES()`. This attribute provides all columns in this row to be referenaceable such that they will render within a `VALUES()` function inside the ON DUPLICATE KEY UPDATE clause. The attribute is named `.inserted` so as not to conflict with the existing `Insert.values()` method.

See also:

INSERT…ON DUPLICATE KEY UPDATE (Upsert) - example of how to use *`Insert.inserted`*

`on_duplicate_key_update(`***kw*`)`

Specifies the ON DUPLICATE KEY UPDATE clause.

Parameters `**kw` – Column keys linked to UPDATE values. The values may be any SQL expression or supported literal Python values.

> **Warning:** This dictionary does **not** take into account Python-specified default UPDATE values or generation functions, e.g. those specified using `Column.onupdate`. These values will not be exercised for an ON DUPLICATE KEY UPDATE style of UPDATE, unless values are manually specified here.

New in version 1.2.

See also:

INSERT…ON DUPLICATE KEY UPDATE (Upsert)

MySQL-Python

Unicode

Please see *Unicode* for current recommendations on unicode handling.

Py3K Support

Currently, MySQLdb only runs on Python 2 and development has been stopped. mysqlclient is fork of MySQLdb and provides Python 3 support as well as some bugfixes.

Using MySQLdb with Google Cloud SQL

Google Cloud SQL now recommends use of the MySQLdb dialect. Connect using a URL like the following:

```
mysql+mysqldb://root@/<dbname>?unix_socket=/cloudsql/<projectid>:<instancename>
```

Server Side Cursors

The mysqldb dialect supports server-side cursors. See *Server Side Cursors*.

pymysql

Unicode

Please see *Unicode* for current recommendations on unicode handling.

MySQL-Python Compatibility

The pymysql DBAPI is a pure Python port of the MySQL-python (MySQLdb) driver, and targets 100% compatibility. Most behavioral notes for MySQL-python apply to the pymysql driver as well.

MySQL-Connector

Unicode

Please see *Unicode* for current recommendations on unicode handling.

cymysql

OurSQL

Unicode

Please see *Unicode* for current recommendations on unicode handling.

Google App Engine

Pooling

Google App Engine connections appear to be randomly recycled, so the dialect does not pool connections. The `NullPool` implementation is installed within the `Engine` by default.

pyodbc

zxjdbc

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 overridden via a `create_engine` URL parameter.

4.1.4 Oracle

Connect Arguments

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

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

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
)
```

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.

LIMIT/OFFSET Support

Oracle has no support for the LIMIT or OFFSET keywords. SQLAlchemy uses a wrapped subquery approach in conjunction with ROWNUM. The exact methodology is taken from http://www.oracle.com/technetwork/issue-archive/2006/06-sep/o56asktom-086197.html .

There are two options which affect its behavior:

- the "FIRST ROWS()" optimization keyword is not used by default. To enable the usage of this optimization directive, specify `optimize_limits=True` to `create_engine()`.

- the values passed for the limit/offset are sent as bound parameters. Some users have observed that Oracle produces a poor query plan when the values are sent as binds and not rendered literally. To render the limit/offset values literally within the SQL statement, specify `use_binds_for_limits=False` to `create_engine()`.

Some users have reported better performance when the entirely different approach of a window query is used, i.e. ROW_NUMBER() OVER (ORDER BY), to provide LIMIT/OFFSET (note that the majority of users don't observe this). To suit this case the method used for LIMIT/OFFSET can be replaced entirely. See the recipe at http://www.sqlalchemy.org/trac/wiki/UsageRecipes/WindowFunctionsByDefault which installs a select compiler that overrides the generation of limit/offset with a window function.

RETURNING Support

The Oracle database supports a limited form of RETURNING, in order to retrieve result sets of matched rows from INSERT, UPDATE and DELETE statements. Oracle's RETURNING..INTO syntax only supports one row being returned, as it relies upon OUT parameters in order to function. In addition, supported DBAPIs have further limitations (see *RETURNING Support*).

SQLAlchemy's "implicit returning" feature, which employs RETURNING within an INSERT and sometimes an UPDATE statement in order to fetch newly generated primary key values and other SQL defaults and expressions, is normally enabled on the Oracle backend. By default, "implicit returning" typically only fetches the value of a single `nextval(some_seq)` expression embedded into an INSERT in order to increment a sequence within an INSERT statement and get the value back at the same time. To disable this feature across the board, specify `implicit_returning=False` to `create_engine()`:

```
engine = create_engine("oracle://scott:tiger@dsn",
                       implicit_returning=False)
```

Implicit returning can also be disabled on a table-by-table basis as a table option:

```
# Core Table
my_table = Table("my_table", metadata, ..., implicit_returning=False)


# declarative
class MyClass(Base):
    __tablename__ = 'my_table'
    __table_args__ = {"implicit_returning": False}
```

See also:

RETURNING Support - additional cx_oracle-specific restrictions on implicit returning.

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().

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.

Synonym/DBLINK Reflection

When using reflection with Table objects, the dialect can optionally search for tables indicated by synonyms, either in local or remote schemas or accessed over DBLINK, by passing the flag `oracle_resolve_synonyms=True` as a keyword argument to the `Table` construct:

```
some_table = Table('some_table', autoload=True,
                           autoload_with=some_engine,
                           oracle_resolve_synonyms=True)
```

When this flag is set, the given name (such as `some_table` above) will be searched not just in the `ALL_TABLES` view, but also within the `ALL_SYNONYMS` view to see if this name is actually a synonym to another name. If the synonym is located and refers to a DBLINK, the oracle dialect knows how to locate the table's information using DBLINK syntax(e.g. `@dblink`).

`oracle_resolve_synonyms` is accepted wherever reflection arguments are accepted, including methods such as `MetaData.reflect()` and `Inspector.get_columns()`.

If synonyms are not in use, this flag should be left disabled.

Constraint Reflection

The Oracle dialect can return information about foreign key, unique, and CHECK constraints, as well as indexes on tables.

Raw information regarding these constraints can be acquired using `Inspector.get_foreign_keys()`, `Inspector.get_unique_constraints()`, `Inspector.get_check_constraints()`, and `Inspector.get_indexes()`.

Changed in version 1.2: The Oracle dialect can now reflect UNIQUE and CHECK constraints.

When using reflection at the `Table` level, the `Table` will also include these constraints.

Note the following caveats:

- When using the `Inspector.get_check_constraints()` method, Oracle builds a special "IS NOT NULL" constraint for columns that specify "NOT NULL". This constraint is **not** returned by default; to include the "IS NOT NULL" constraints, pass the flag `include_all=True`:

  ```
  from sqlalchemy import create_engine, inspect

  engine = create_engine("oracle+cx_oracle://s:t@dsn")
  inspector = inspect(engine)
  all_check_constraints = inspector.get_check_constraints(
      "some_table", include_all=True)
  ```

- in most cases, when reflecting a `Table`, a UNIQUE constraint will **not** be available as a `UniqueConstraint` object, as Oracle mirrors unique constraints with a UNIQUE index in most cases (the exception seems to be when two or more unique constraints represent the same columns); the `Table` will instead represent these using `Index` with the `unique=True` flag set.
- Oracle creates an implicit index for the primary key of a table; this index is **excluded** from all index results.
- the list of columns reflected for an index will not include column names that start with SYS_NC.

Table names with SYSTEM/SYSAUX tablespaces

The `Inspector.get_table_names()` and `Inspector.get_temp_table_names()` methods each return a list of table names for the current engine. These methods are also part of the reflection which occurs within an operation such as `MetaData.reflect()`. By default, these operations exclude the `SYSTEM` and `SYSAUX` tablespaces from the operation. In order to change this, the default list of tablespaces excluded can be changed at the engine level using the `exclude_tablespaces` parameter:

```
# exclude SYSAUX and SOME_TABLESPACE, but not SYSTEM
e = create_engine(
  "oracle://scott:tiger@xe",
  exclude_tablespaces=["SYSAUX", "SOME_TABLESPACE"])
```

New in version 1.1.

DateTime Compatibility

Oracle has no datatype known as `DATETIME`, it instead has only `DATE`, which can actually store a date and time value. For this reason, the Oracle dialect provides a type *oracle.DATE* which is a subclass of `DateTime`. This type has no special behavior, and is only present as a "marker" for this type; additionally, when a database column is reflected and the type is reported as `DATE`, the time-supporting *oracle.DATE* type is used.

Changed in version 0.9.4: Added *oracle.DATE* to subclass `DateTime`. This is a change as previous versions would reflect a `DATE` column as `types.DATE`, which subclasses `Date`. The only significance here is for schemes that are examining the type of column for use in special Python translations or for migrating schemas to other database backends.

Oracle Table Options

The CREATE TABLE phrase supports the following options with Oracle in conjunction with the `Table` construct:

- `ON COMMIT`:

```
Table(
    "some_table", metadata, ...,
    prefixes=['GLOBAL TEMPORARY'], oracle_on_commit='PRESERVE ROWS')
```

New in version 1.0.0.

- `COMPRESS`:

```
 Table('mytable', metadata, Column('data', String(32)),
     oracle_compress=True)

 Table('mytable', metadata, Column('data', String(32)),
     oracle_compress=6)

The ``oracle_compress`` parameter accepts either an integer compression
level, or ``True`` to use the default compression level.
```

New in version 1.0.0.

Oracle Specific Index Options

Bitmap Indexes

You can specify the `oracle_bitmap` parameter to create a bitmap index instead of a B-tree index:

```
Index('my_index', my_table.c.data, oracle_bitmap=True)
```

Bitmap indexes cannot be unique and cannot be compressed. SQLAlchemy will not check for such limitations, only the database will.

New in version 1.0.0.

Index compression

Oracle has a more efficient storage mode for indexes containing lots of repeated values. Use the `oracle_compress` parameter to turn on key c ompression:

```
Index('my_index', my_table.c.data, oracle_compress=True)

Index('my_index', my_table.c.data1, my_table.c.data2, unique=True,
       oracle_compress=1)
```

The `oracle_compress` parameter accepts either an integer specifying the number of prefix columns to compress, or `True` to use the default (all columns for non-unique indexes, all but the last column for unique indexes).

New in version 1.0.0.

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, \
            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.**BFILE**(*length=None*)

__init__(*length=None*)
Construct a LargeBinary type.

Parameters **length** – optional, a length for the column for use in DDL statements, for those binary types that accept a length, such as the MySQL BLOB type.

class sqlalchemy.dialects.oracle.**DATE**(*timezone=False*)
Provide the oracle DATE type.

This type has no special Python behavior, except that it subclasses `types.DateTime`; this is to suit the fact that the Oracle `DATE` type supports a time value.

New in version 0.9.4.

__init__(*timezone=False*)
Construct a new `DateTime`.

Parameters **timezone** – boolean. Indicates that the datetime type should enable timezone support, if available on the **base date/time-holding type only**. It is recommended to make use of the `TIMESTAMP` datatype directly

when using this flag, as some databases include separate generic date/time-holding types distinct from the timezone-capable TIMESTAMP datatype, such as Oracle.

class sqlalchemy.dialects.oracle.**DOUBLE_PRECISION**(*precision=None*, *scale=None*, *asdecimal=None*)

class sqlalchemy.dialects.oracle.**INTERVAL**(*day_precision=None*, *second_precision=None*)

__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.**NCLOB**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. 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 if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the `String` type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to `create_engine()` as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the `Unicode` or `UnicodeText` types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **`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 a last resort for reading strings from a column with varied or corrupted encodings.

class sqlalchemy.dialects.oracle.**NUMBER**(*precision=None*, *scale=None*, *asdecimal=None*)

class sqlalchemy.dialects.oracle.**LONG**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **`length`** – optional, a length for the column for use in DDL and CAST expressions. 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 if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **`collation`** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and PostgreSQL. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **`convert_unicode`** – When set to `True`, the `String` type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to `create_engine()` as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the `Unicode` or `UnicodeText` types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

  For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- `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 a last resort for reading strings from a column with varied or corrupted encodings.

`class sqlalchemy.dialects.oracle.RAW(`*length=None*`)`

cx_Oracle

Additional Connect Arguments

When connecting with `dbname` 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:

- `arraysize` - set the cx_oracle.arraysize value on cursors, defaulted to 50. This setting is significant with cx_Oracle as the contents of LOB objects are only readable within a "live" row (e.g. within a batch of 50 rows).
- `auto_convert_lobs` - defaults to True; See *LOB Objects*.
- `coerce_to_unicode` - see *Unicode* for detail.
- `coerce_to_decimal` - see *Precision Numerics* for detail.
- `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 the cx_Oracle DBAPI itself.
- `service_name` - An option to use connection string (DSN) with `SERVICE_NAME` instead of `SID`. It can't be passed when a `database` part is given. E.g. `oracle+cx_oracle://scott:tiger@host:1521/?service_name=hr` is a valid url. This value is only available as a URL query string argument.

  New in version 1.0.0.

Unicode

The cx_Oracle DBAPI as of version 5 fully supports unicode, and has the ability to return string results as Python unicode objects natively.

When used in Python 3, cx_Oracle returns all strings as Python unicode objects (that is, plain `str` in Python 3). In Python 2, it will return as Python unicode those column values that are of type `NVARCHAR` or `NCLOB`. For column values that are of type `VARCHAR` or other non-unicode string types, it will return values as Python strings (e.g. bytestrings).

The cx_Oracle SQLAlchemy dialect presents several different options for the use case of receiving `VARCHAR` column values as Python unicode objects under Python 2:

- When using Core expression objects as well as the ORM, SQLAlchemy's unicode-decoding services are available, which are established by using either the `Unicode` datatype or by using the `String` datatype with `String.convert_unicode` set to True.

- When using raw SQL strings, typing behavior can be added for unicode conversion using the `text()` construct:

```
from sqlalchemy import text, Unicode
result = conn.execute(
    text("select username from user").columns(username=Unicode))
```

- Otherwise, when using raw SQL strings sent directly to an `.execute()` method without any Core typing behavior added, the flag `coerce_to_unicode=True` flag can be passed to `create_engine()` which will add an unconditional unicode processor to cx_Oracle for all string values:

```
engine = create_engine("oracle+cx_oracle://dsn", coerce_to_unicode=True)
```

  The above approach will add significant latency to result-set fetches of plain string values.

RETURNING Support

The cx_Oracle dialect implements RETURNING using OUT parameters. The dialect supports RETURNING fully, however cx_Oracle 6 is recommended for complete support.

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, which takes place within a cx_Oracle outputtypehandler.

cx_Oracle prior to version 6 would require that LOB objects be read before a new batch of rows would be read, as determined by the `cursor.arraysize`. As of the 6 series, this limitation has been lifted. Nevertheless, because SQLAlchemy pre-reads these LOBs up front, this issue is avoided in any case.

To disable the auto "read()" feature of the dialect, the flag `auto_convert_lobs=False` may be passed to `create_engine()`. Under the cx_Oracle 5 series, having this flag turned off means there is the chance of reading from a stale LOB object if not read as it is fetched. With cx_Oracle 6, this issue is resolved.

Changed in version 1.2: the LOB handling system has been greatly simplified internally to make use of outputtypehandlers, and no longer makes use of alternate "buffered" result set objects.

Two Phase Transactions Not Supported

Two phase transactions are **not supported** under cx_Oracle due to poor driver support. As of cx_Oracle 6.0b1, the interface for two phase transactions has been changed to be more of a direct pass-through to the underlying OCI layer with less automation. The additional logic to support this system is not implemented in SQLAlchemy.

Precision Numerics

SQLAlchemy's numeric types can handle receiving and returning values as Python `Decimal` objects or float objects. When a `Numeric` object, or a subclass such as `Float`, *`oracle.DOUBLE_PRECISION`* etc. is in use, the `Numeric.asdecimal` flag determines if values should be coerced to `Decimal` upon return, or returned as float objects. To make matters more complicated under Oracle, Oracle's `NUMBER` type can also represent integer values if the "scale" is zero, so the Oracle-specific *`oracle.NUMBER`* type takes this into account as well.

The cx_Oracle dialect makes extensive use of connection- and cursor-level "outputtypehandler" callables in order to coerce numeric values as requested. These callables are specific to the specific flavor of `Numeric` in use, as well as if no SQLAlchemy typing objects are present. There are observed scenarios where Oracle may sends incomplete or ambiguous information about the numeric types being returned, such as a query

where the numeric types are buried under multiple levels of subquery. The type handlers do their best to make the right decision in all cases, deferring to the underlying cx_Oracle DBAPI for all those cases where the driver can make the best decision.

When no typing objects are present, as when executing plain SQL strings, a default "outputtypehandler" is present which will generally return numeric values which specify precision and scale as Python `Decimal` objects. To disable this coercion to decimal for performance reasons, pass the flag `coerce_to_decimal=False` to `create_engine()`:

```
engine = create_engine("oracle+cx_oracle://dsn", coerce_to_decimal=False)
```

The `coerce_to_decimal` flag only impacts the results of plain string SQL staements that are not otherwise associated with a `Numeric` SQLAlchemy type (or a subclass of such).

Changed in version 1.2: The numeric handling system for cx_Oracle has been reworked to take advantage of newer cx_Oracle features as well as better integration of outputtypehandlers.

zxjdbc

4.1.5 PostgreSQL

Sequences/SERIAL/IDENTITY

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

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()`.

Postgresql 10 IDENTITY columns

Postgresql 10 has a new IDENTITY feature that supersedes the use of SERIAL. Built-in support for rendering of IDENTITY is not available yet, however the following compilation hook may be used to replace occurrences of SERIAL with IDENTITY:

```
from sqlalchemy.schema import CreateColumn
from sqlalchemy.ext.compiler import compiles


@compiles(CreateColumn, 'postgresql')
def use_identity(element, compiler, **kw):
    text = compiler.visit_create_column(element, **kw)
```

```
    text = text.replace("SERIAL", "INT GENERATED BY DEFAULT AS IDENTITY")
    return text
```

Using the above, a table such as:

```
t = Table(
    't', m,
    Column('id', Integer, primary_key=True),
    Column('data', String)
)
```

Will generate on the backing database as:

```
CREATE TABLE t (
    id INT GENERATED BY DEFAULT AS IDENTITY NOT NULL,
    data VARCHAR,
    PRIMARY KEY (id)
)
```

Transaction Isolation Level

All PostgreSQL dialects support setting of transaction isolation level both via a dialect-specific parameter `create_engine.isolation_level` accepted by `create_engine()`, as well as the `Connection.execution_options.isolation_level` argument as passed to `Connection.execution_options()`. When using a non-psycopg2 dialect, this feature works by issuing the command `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL <level>` for each new connection. For the special AUTOCOMMIT isolation level, DBAPI-specific techniques are used.

To set isolation level using `create_engine()`:

```
engine = create_engine(
    "postgresql+pg8000://scott:tiger@localhost/test",
    isolation_level="READ UNCOMMITTED"
)
```

To set using per-connection execution options:

```
connection = engine.connect()
connection = connection.execution_options(
    isolation_level="READ COMMITTED"
)
```

Valid values for `isolation_level` include:

- `READ COMMITTED`
- `READ UNCOMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`
- `AUTOCOMMIT` - on psycopg2 / pg8000 only

See also:

Psycopg2 Transaction Isolation Level

pg8000 Transaction Isolation Level

Remote-Schema Table Introspection and PostgreSQL search_path

The PostgreSQL dialect can reflect tables from any schema. The `Table.schema` argument, or alternatively the `MetaData.reflect.schema` argument determines which schema will be searched for the table or tables. The reflected `Table` objects will in all cases retain this `.schema` attribute as was specified. However, with regards to tables which these `Table` objects refer to via foreign key constraint, a decision must be made as to how the `.schema` is represented in those remote tables, in the case where that remote schema name is also a member of the current PostgreSQL search path.

By default, the PostgreSQL dialect mimics the behavior encouraged by PostgreSQL's own `pg_get_constraintdef()` builtin procedure. This function returns a sample definition for a particular foreign key constraint, omitting the referenced schema name from that definition when the name is also in the PostgreSQL schema search path. The interaction below illustrates this behavior:

```
test=> CREATE TABLE test_schema.referred(id INTEGER PRIMARY KEY);
CREATE TABLE
test=> CREATE TABLE referring(
test(>         id INTEGER PRIMARY KEY,
test(>         referred_id INTEGER REFERENCES test_schema.referred(id));
CREATE TABLE
test=> SET search_path TO public, test_schema;
test=> SELECT pg_catalog.pg_get_constraintdef(r.oid, true) FROM
test-> pg_catalog.pg_class c JOIN pg_catalog.pg_namespace n
test-> ON n.oid = c.relnamespace
test-> JOIN pg_catalog.pg_constraint r  ON c.oid = r.conrelid
test-> WHERE c.relname='referring' AND r.contype = 'f'
test-> ;
               pg_get_constraintdef
---------------------------------------------------
 FOREIGN KEY (referred_id) REFERENCES referred(id)
(1 row)
```

Above, we created a table `referred` as a member of the remote schema `test_schema`, however when we added `test_schema` to the PG `search_path` and then asked `pg_get_constraintdef()` for the `FOREIGN KEY` syntax, `test_schema` was not included in the output of the function.

On the other hand, if we set the search path back to the typical default of `public`:

```
test=> SET search_path TO public;
SET
```

The same query against `pg_get_constraintdef()` now returns the fully schema-qualified name for us:

```
test=> SELECT pg_catalog.pg_get_constraintdef(r.oid, true) FROM
test-> pg_catalog.pg_class c JOIN pg_catalog.pg_namespace n
test-> ON n.oid = c.relnamespace
test-> JOIN pg_catalog.pg_constraint r  ON c.oid = r.conrelid
test-> WHERE c.relname='referring' AND r.contype = 'f';
                     pg_get_constraintdef
---------------------------------------------------------------
 FOREIGN KEY (referred_id) REFERENCES test_schema.referred(id)
(1 row)
```

SQLAlchemy will by default use the return value of `pg_get_constraintdef()` in order to determine the remote schema name. That is, if our `search_path` were set to include `test_schema`, and we invoked a table reflection process as follows:

```
>>> from sqlalchemy import Table, MetaData, create_engine
>>> engine = create_engine("postgresql://scott:tiger@localhost/test")
>>> with engine.connect() as conn:
...     conn.execute("SET search_path TO test_schema, public")
...     meta = MetaData()
```

```
...     referring = Table('referring', meta,
...                       autoload=True, autoload_with=conn)
...
<sqlalchemy.engine.result.ResultProxy object at 0x101612ed0>
```

The above process would deliver to the `MetaData.tables` collection `referred` table named **without** the schema:

```
>>> meta.tables['referred'].schema is None
True
```

To alter the behavior of reflection such that the referred schema is maintained regardless of the `search_path` setting, use the `postgresql_ignore_search_path` option, which can be specified as a dialect-specific argument to both `Table` as well as `MetaData.reflect()`:

```
>>> with engine.connect() as conn:
...     conn.execute("SET search_path TO test_schema, public")
...     meta = MetaData()
...     referring = Table('referring', meta, autoload=True,
...                       autoload_with=conn,
...                       postgresql_ignore_search_path=True)
...
<sqlalchemy.engine.result.ResultProxy object at 0x1016126d0>
```

We will now have `test_schema.referred` stored as schema-qualified:

```
>>> meta.tables['test_schema.referred'].schema
'test_schema'
```

Best Practices for PostgreSQL Schema reflection

The description of PostgreSQL schema reflection behavior is complex, and is the product of many years of dealing with widely varied use cases and user preferences. But in fact, there's no need to understand any of it if you just stick to the simplest use pattern: leave the `search_path` set to its default of `public` only, never refer to the name `public` as an explicit schema name otherwise, and refer to all other schema names explicitly when building up a `Table` object. The options described here are only for those users who can't, or prefer not to, stay within these guidelines.

Note that **in all cases**, the "default" schema is always reflected as `None`. The "default" schema on PostgreSQL is that which is returned by the PostgreSQL `current_schema()` function. On a typical PostgreSQL installation, this is the name `public`. So a table that refers to another which is in the `public` (i.e. default) schema will always have the `.schema` attribute set to `None`.

New in version 0.9.2: Added the `postgresql_ignore_search_path` dialect-level option accepted by `Table` and `MetaData.reflect()`.

See also:

The Schema Search Path - on the PostgreSQL website.

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()
```

INSERT...ON CONFLICT (Upsert)

Starting with version 9.5, PostgreSQL allows "upserts" (update or insert) of rows into a table via the `ON CONFLICT` clause of the `INSERT` statement. A candidate row will only be inserted if that row does not violate any unique constraints. In the case of a unique constraint violation, a secondary action can occur which can be either "DO UPDATE", indicating that the data in the target row should be updated, or "DO NOTHING", which indicates to silently skip this row.

Conflicts are determined using existing unique constraints and indexes. These constraints may be identified either using their name as stated in DDL, or they may be *inferred* by stating the columns and conditions that comprise the indexes.

SQLAlchemy provides `ON CONFLICT` support via the PostgreSQL-specific *postgresql.dml.insert()* function, which provides the generative methods *on_conflict_do_update()* and *on_conflict_do_nothing()*:

```
from sqlalchemy.dialects.postgresql import insert

insert_stmt = insert(my_table).values(
    id='some_existing_id',
    data='inserted value')

do_nothing_stmt = insert_stmt.on_conflict_do_nothing(
    index_elements=['id']
)

conn.execute(do_nothing_stmt)

do_update_stmt = insert_stmt.on_conflict_do_update(
    constraint='pk_my_table',
    set_=dict(data='updated value')
)

conn.execute(do_update_stmt)
```

Both methods supply the "target" of the conflict using either the named constraint or by column inference:

- The *Insert.on_conflict_do_update.index_elements* argument specifies a sequence containing string column names, `Column` objects, and/or SQL expression elements, which would identify a unique index:

  ```
  do_update_stmt = insert_stmt.on_conflict_do_update(
      index_elements=['id'],
      set_=dict(data='updated value')
  )

  do_update_stmt = insert_stmt.on_conflict_do_update(
      index_elements=[my_table.c.id],
  ```

```
    set_=dict(data='updated value')
)
```

- When using *Insert.on_conflict_do_update.index_elements* to infer an index, a partial index can be inferred by also specifying the use the *Insert.on_conflict_do_update.index_where* parameter:

```
from sqlalchemy.dialects.postgresql import insert

stmt = insert(my_table).values(user_email='a@b.com', data='inserted data')
stmt = stmt.on_conflict_do_update(
    index_elements=[my_table.c.user_email],
    index_where=my_table.c.user_email.like('%@gmail.com'),
    set_=dict(data=stmt.excluded.data)
    )
conn.execute(stmt)
```

- The *Insert.on_conflict_do_update.constraint* argument is used to specify an index directly rather than inferring it. This can be the name of a UNIQUE constraint, a PRIMARY KEY constraint, or an INDEX:

```
do_update_stmt = insert_stmt.on_conflict_do_update(
    constraint='my_table_idx_1',
    set_=dict(data='updated value')
)

do_update_stmt = insert_stmt.on_conflict_do_update(
    constraint='my_table_pk',
    set_=dict(data='updated value')
)
```

- The *Insert.on_conflict_do_update.constraint* argument may also refer to a SQLAlchemy construct representing a constraint, e.g. `UniqueConstraint`, `PrimaryKeyConstraint`, `Index`, or *ExcludeConstraint*. In this use, if the constraint has a name, it is used directly. Otherwise, if the constraint is unnamed, then inference will be used, where the expressions and optional WHERE clause of the constraint will be spelled out in the construct. This use is especially convenient to refer to the named or unnamed primary key of a `Table` using the `Table.primary_key` attribute:

```
do_update_stmt = insert_stmt.on_conflict_do_update(
    constraint=my_table.primary_key,
    set_=dict(data='updated value')
)
```

`ON CONFLICT...DO UPDATE` is used to perform an update of the already existing row, using any combination of new values as well as values from the proposed insertion. These values are specified using the *Insert.on_conflict_do_update.set_* parameter. This parameter accepts a dictionary which consists of direct values for UPDATE:

```
from sqlalchemy.dialects.postgresql import insert

stmt = insert(my_table).values(id='some_id', data='inserted value')
do_update_stmt = stmt.on_conflict_do_update(
    index_elements=['id'],
    set_=dict(data='updated value')
    )
conn.execute(do_update_stmt)
```

Warning: The *Insert.on_conflict_do_update()* method does **not** take into account Python-side default UPDATE values or generation functions, e.g. e.g. those specified using `Column.onupdate`.

These values will not be exercised for an ON CONFLICT style of UPDATE, unless they are manually specified in the *Insert.on_conflict_do_update.set_* dictionary.

In order to refer to the proposed insertion row, the special alias *excluded* is available as an attribute on the *postgresql.dml.Insert* object; this object is a `ColumnCollection` which alias contains all columns of the target table:

```
from sqlalchemy.dialects.postgresql import insert

stmt = insert(my_table).values(
    id='some_id',
    data='inserted value',
    author='jlh')
do_update_stmt = stmt.on_conflict_do_update(
    index_elements=['id'],
    set_=dict(data='updated value', author=stmt.excluded.author)
    )
conn.execute(do_update_stmt)
```

The *Insert.on_conflict_do_update()* method also accepts a WHERE clause using the *Insert.on_conflict_do_update.where* parameter, which will limit those rows which receive an UPDATE:

```
from sqlalchemy.dialects.postgresql import insert

stmt = insert(my_table).values(
    id='some_id',
    data='inserted value',
    author='jlh')
on_update_stmt = stmt.on_conflict_do_update(
    index_elements=['id'],
    set_=dict(data='updated value', author=stmt.excluded.author)
    where=(my_table.c.status == 2)
    )
conn.execute(on_update_stmt)
```

`ON CONFLICT` may also be used to skip inserting a row entirely if any conflict with a unique or exclusion constraint occurs; below this is illustrated using the *on_conflict_do_nothing()* method:

```
from sqlalchemy.dialects.postgresql import insert

stmt = insert(my_table).values(id='some_id', data='inserted value')
stmt = stmt.on_conflict_do_nothing(index_elements=['id'])
conn.execute(stmt)
```

If `DO NOTHING` is used without specifying any columns or constraint, it has the effect of skipping the INSERT for any unique or exclusion constraint violation which occurs:

```
from sqlalchemy.dialects.postgresql import insert

stmt = insert(my_table).values(id='some_id', data='inserted value')
stmt = stmt.on_conflict_do_nothing()
conn.execute(stmt)
```

New in version 1.1: Added support for PostgreSQL ON CONFLICT clauses

See also:

INSERT .. ON CONFLICT - in the PostgreSQL documentation.

Full Text Search

SQLAlchemy makes available the PostgreSQL `@@` operator via the `ColumnElement.match()` method on any textual column expression. On a PostgreSQL dialect, an expression like the following:

```
select([sometable.c.text.match("search string")])
```

will emit to the database:

```
SELECT text @@ to_tsquery('search string') FROM table
```

The PostgreSQL text search functions such as `to_tsquery()` and `to_tsvector()` are available explicitly using the standard `func` construct. For example:

```
select([
    func.to_tsvector('fat cats ate rats').match('cat & rat')
])
```

Emits the equivalent of:

```
SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')
```

The *postgresql.TSVECTOR* type can provide for explicit CAST:

```
from sqlalchemy.dialects.postgresql import TSVECTOR
from sqlalchemy import select, cast
select([cast("some text", TSVECTOR)])
```

produces a statement equivalent to:

```
SELECT CAST('some text' AS TSVECTOR) AS anon_1
```

Full Text Searches in PostgreSQL are influenced by a combination of: the PostgresSQL setting of `default_text_search_config`, the `regconfig` used to build the GIN/GiST indexes, and the `regconfig` optionally passed in during a query.

When performing a Full Text Search against a column that has a GIN or GiST index that is already pre-computed (which is common on full text searches) one may need to explicitly pass in a particular PostgresSQL `regconfig` value to ensure the query-planner utilizes the index and does not re-compute the column on demand.

In order to provide for this explicit query planning, or to use different search strategies, the `match` method accepts a `postgresql_regconfig` keyword argument:

```
select([mytable.c.id]).where(
    mytable.c.title.match('somestring', postgresql_regconfig='english')
)
```

Emits the equivalent of:

```
SELECT mytable.id FROM mytable
WHERE mytable.title @@ to_tsquery('english', 'somestring')
```

One can also specifically pass in a *'regconfig'* value to the `to_tsvector()` command as the initial argument:

```
select([mytable.c.id]).where(
        func.to_tsvector('english', mytable.c.title )\
        .match('somestring', postgresql_regconfig='english')
    )
```

produces a statement equivalent to:

```
SELECT mytable.id FROM mytable
WHERE to_tsvector('english', mytable.title) @@
    to_tsquery('english', 'somestring')
```

It is recommended that you use the `EXPLAIN ANALYZE...` tool from PostgresSQL to ensure that you are generating queries with SQLAlchemy that take full advantage of any indexes you may have created for full text search.

FROM ONLY ...

The dialect supports PostgreSQL's ONLY keyword for targeting only a particular table in an inheritance hierarchy. This can be used to produce the `SELECT ... FROM ONLY`, `UPDATE ONLY ...`, and `DELETE FROM ONLY ...` syntaxes. It uses SQLAlchemy's hints mechanism:

```
# SELECT ... FROM ONLY ...
result = table.select().with_hint(table, 'ONLY', 'postgresql')
print result.fetchall()

# UPDATE ONLY ...
table.update(values=dict(foo='bar')).with_hint('ONLY',
                                               dialect_name='postgresql')

# DELETE FROM ONLY ...
table.delete().with_hint('ONLY', dialect_name='postgresql')
```

PostgreSQL-Specific Index Options

Several extensions to the `Index` construct are available, specific to the PostgreSQL dialect.

Partial Indexes

Partial indexes add criterion to the index definition so that the index is applied to a subset of rows. These can be specified on `Index` using the `postgresql_where` keyword argument:

```
Index('my_index', my_table.c.id, postgresql_where=my_table.c.value > 10)
```

Operator Classes

PostgreSQL allows the specification of an *operator class* for each column of an index (see http://www.postgresql.org/docs/8.3/interactive/indexes-opclass.html). The `Index` construct allows these to be specified via the `postgresql_ops` keyword argument:

```
Index(
    'my_index', my_table.c.id, my_table.c.data,
    postgresql_ops={
        'data': 'text_pattern_ops',
        'id': 'int4_ops'
    })
```

Note that the keys in the `postgresql_ops` dictionary are the "key" name of the `Column`, i.e. the name used to access it from the `.c` collection of `Table`, which can be configured to be different than the actual name of the column as expressed in the database.

If `postgresql_ops` is to be used against a complex SQL expression such as a function call, then to apply to the column it must be given a label that is identified in the dictionary by name, e.g.:

```
Index(
    'my_index', my_table.c.id,
    func.lower(my_table.c.data).label('data_lower'),
    postgresql_ops={
        'data_lower': 'text_pattern_ops',
        'id': 'int4_ops'
    })
```

Index Types

PostgreSQL provides several index types: B-Tree, Hash, GiST, and GIN, as well as the ability for users to create their own (see http://www.postgresql.org/docs/8.3/static/indexes-types.html). These can be specified on `Index` using the `postgresql_using` keyword argument:

```
Index('my_index', my_table.c.data, postgresql_using='gin')
```

The value passed to the keyword argument will be simply passed through to the underlying CREATE INDEX command, so it *must* be a valid index type for your version of PostgreSQL.

Index Storage Parameters

PostgreSQL allows storage parameters to be set on indexes. The storage parameters available depend on the index method used by the index. Storage parameters can be specified on `Index` using the `postgresql_with` keyword argument:

```
Index('my_index', my_table.c.data, postgresql_with={"fillfactor": 50})
```

New in version 1.0.6.

PostgreSQL allows to define the tablespace in which to create the index. The tablespace can be specified on `Index` using the `postgresql_tablespace` keyword argument:

```
Index('my_index', my_table.c.data, postgresql_tablespace='my_tablespace')
```

New in version 1.1.

Note that the same option is available on `Table` as well.

Indexes with CONCURRENTLY

The PostgreSQL index option CONCURRENTLY is supported by passing the flag `postgresql_concurrently` to the `Index` construct:

```
tbl = Table('testtbl', m, Column('data', Integer))

idx1 = Index('test_idx1', tbl.c.data, postgresql_concurrently=True)
```

The above index construct will render DDL for CREATE INDEX, assuming PostgreSQL 8.2 or higher is detected or for a connection-less dialect, as:

```
CREATE INDEX CONCURRENTLY test_idx1 ON testtbl (data)
```

For DROP INDEX, assuming PostgreSQL 9.2 or higher is detected or for a connection-less dialect, it will emit:

```
DROP INDEX CONCURRENTLY test_idx1
```

New in version 1.1: support for CONCURRENTLY on DROP INDEX. The CONCURRENTLY keyword is now only emitted if a high enough version of PostgreSQL is detected on the connection (or for a connection-less dialect).

When using CONCURRENTLY, the Postgresql database requires that the statement be invoked outside of a transaction block. The Python DBAPI enforces that even for a single statement, a transaction is present, so to use this construct, the DBAPI's "autocommit" mode must be used:

```
metadata = MetaData()
table = Table(
    "foo", metadata,
    Column("id", String))
index = Index(
    "foo_idx", table.c.id, postgresql_concurrently=True)

with engine.connect() as conn:
    with conn.execution_options(isolation_level='AUTOCOMMIT'):
        table.create(conn)
```

See also:

Transaction Isolation Level

PostgreSQL Index Reflection

The PostgreSQL database creates a UNIQUE INDEX implicitly whenever the UNIQUE CONSTRAINT construct is used. When inspecting a table using `Inspector`, the `Inspector.get_indexes()` and the `Inspector.get_unique_constraints()` will report on these two constructs distinctly; in the case of the index, the key `duplicates_constraint` will be present in the index entry if it is detected as mirroring a constraint. When performing reflection using `Table(..., autoload=True)`, the UNIQUE INDEX is **not** returned in `Table.indexes` when it is detected as mirroring a `UniqueConstraint` in the `Table.constraints` collection.

Changed in version 1.0.0: - `Table` reflection now includes `UniqueConstraint` objects present in the `Table.constraints` collection; the PostgreSQL backend will no longer include a "mirrored" `Index` construct in `Table.indexes` if it is detected as corresponding to a unique constraint.

Special Reflection Options

The `Inspector` used for the PostgreSQL backend is an instance of *PGInspector*, which offers additional methods:

```
from sqlalchemy import create_engine, inspect

engine = create_engine("postgresql+psycopg2://localhost/test")
insp = inspect(engine)  # will be a PGInspector

print(insp.get_enums())
```

class sqlalchemy.dialects.postgresql.base.**PGInspector**(*conn*)

get_enums(*schema=None*)

Return a list of ENUM objects.

Each member is a dictionary containing these fields:

- name - name of the enum
- schema - the schema name for the enum.
- visible - boolean, whether or not this enum is visible in the default search path.

- labels - a list of string labels that apply to the enum.

Parameters **schema** – schema name. If None, the default schema (typically 'public') is used. May also be set to '*' to indicate load enums for all schemas.

New in version 1.0.0.

`get_foreign_table_names`(*schema=None*)
Return a list of FOREIGN TABLE names.

Behavior is similar to that of `Inspector.get_table_names()`, except that the list is limited to those tables tha report a `relkind` value of `f`.

New in version 1.0.0.

`get_table_oid`(*table_name*, *schema=None*)
Return the OID for the given table name.

`get_view_names`(*schema=None*, *include=('plain', 'materialized')*)
Return all view names in *schema*.

Parameters

- **schema** – Optional, retrieve names from a non-default schema. For special quoting, use `quoted_name`.
- **include** – specify which types of views to return. Passed as a string value (for a single type) or a tuple (for any number of types). Defaults to `('plain', 'materialized')`.

  New in version 1.1.

PostgreSQL Table Options

Several options for CREATE TABLE are supported directly by the PostgreSQL dialect in conjunction with the `Table` construct:

- `TABLESPACE`:

```
Table("some_table", metadata, ..., postgresql_tablespace='some_tablespace')
```

  The above option is also available on the `Index` construct.

- `ON COMMIT`:

```
Table("some_table", metadata, ..., postgresql_on_commit='PRESERVE ROWS')
```

- `WITH OIDS`:

```
Table("some_table", metadata, ..., postgresql_with_oids=True)
```

- `WITHOUT OIDS`:

```
Table("some_table", metadata, ..., postgresql_with_oids=False)
```

- `INHERITS`:

```
Table("some_table", metadata, ..., postgresql_inherits="some_supertable")

Table("some_table", metadata, ..., postgresql_inherits=("t1", "t2", ...))
```

New in version 1.0.0.

See also:

PostgreSQL CREATE TABLE options

ARRAY Types

The PostgreSQL dialect supports arrays, both as multidimensional column types as well as array literals:

- *postgresql.ARRAY* - ARRAY datatype
- *postgresql.array* - array literal
- *postgresql.array_agg()* - ARRAY_AGG SQL function
- *postgresql.aggregate_order_by* - helper for PG's ORDER BY aggregate function syntax.

JSON Types

The PostgreSQL dialect supports both JSON and JSONB datatypes, including psycopg2's native support and support for all of PostgreSQL's special operators:

- *postgresql.JSON*
- *postgresql.JSONB*

HSTORE Type

The PostgreSQL HSTORE type as well as hstore literals are supported:

- *postgresql.HSTORE* - HSTORE datatype
- *postgresql.hstore* - hstore literal

ENUM Types

PostgreSQL has an independently creatable TYPE structure which is used to implement an enumerated type. This approach introduces significant complexity on the SQLAlchemy side in terms of when this type should be CREATED and DROPPED. The type object is also an independently reflectable entity. The following sections should be consulted:

- *postgresql.ENUM* - DDL and typing support for ENUM.
- *PGInspector.get_enums()* - retrieve a listing of current ENUM types
- *postgresql.ENUM.create()* , *postgresql.ENUM.drop()* - individual CREATE and DROP commands for ENUM.

Using ENUM with ARRAY

The combination of ENUM and ARRAY is not directly supported by backend DBAPIs at this time. In order to send and receive an ARRAY of ENUM, use the following workaround type:

```
class ArrayOfEnum(ARRAY):

    def bind_expression(self, bindvalue):
        return sa.cast(bindvalue, self)

    def result_processor(self, dialect, coltype):
        super_rp = super(ArrayOfEnum, self).result_processor(
            dialect, coltype)

        def handle_raw_string(value):
            inner = re.match(r"^{(.*)}$", value).group(1)
            return inner.split(",") if inner else []

        def process(value):
```

```
            if value is None:
                return None
            return super_rp(handle_raw_string(value))
        return process
```

E.g.:

```
Table(
    'mydata', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', ArrayOfEnum(ENUM('a', 'b, 'c', name='myenum')))

)
```

This type is not included as a built-in type as it would be incompatible with a DBAPI that suddenly decides to support ARRAY of ENUM directly in a new version.

Using JSON/JSONB with ARRAY

Similar to using ENUM, for an ARRAY of JSON/JSONB we need to render the appropriate CAST, however current psycopg2 drivers seem to handle the result for ARRAY of JSON automatically, so the type is simpler:

```
class CastingArray(ARRAY):
    def bind_expression(self, bindvalue):
        return sa.cast(bindvalue, self)
```

E.g.:

```
Table(
    'mydata', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', CastingArray(JSONB))
)
```

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, HSTORE, INET, INTEGER, \
    INTERVAL, JSON, JSONB, MACADDR, MONEY, NUMERIC, OID, REAL, SMALLINT, TEXT, \
    TIME, TIMESTAMP, UUID, VARCHAR, INT4RANGE, INT8RANGE, NUMRANGE, \
    DATERANGE, TSRANGE, TSTZRANGE, TSVECTOR
```

Types which are specific to PostgreSQL, or have PostgreSQL-specific construction arguments, are as follows:

class sqlalchemy.dialects.postgresql.**aggregate_order_by**(*target*, *order_by*)

Represent a PostgreSQL aggregate order by expression.

E.g.:

```
from sqlalchemy.dialects.postgresql import aggregate_order_by
expr = func.array_agg(aggregate_order_by(table.c.a, table.c.b.desc()))
stmt = select([expr])
```

would represent the expression:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Similarly:

```
expr = func.string_agg(
    table.c.a,
    aggregate_order_by(literal_column("','"), table.c.a)
)
stmt = select([expr])
```

Would represent:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

New in version 1.1.

See also:

`array_agg`

class sqlalchemy.dialects.postgresql.**array**(*clauses*, ***kw*)

A PostgreSQL ARRAY literal.

This is used to produce ARRAY literals in SQL expressions, e.g.:

```
from sqlalchemy.dialects.postgresql import array
from sqlalchemy.dialects import postgresql
from sqlalchemy import select, func

stmt = select([
                array([1,2]) + array([3,4,5])
            ])

print stmt.compile(dialect=postgresql.dialect())
```

Produces the SQL:

```
SELECT ARRAY[%(param_1)s, %(param_2)s] ||
    ARRAY[%(param_3)s, %(param_4)s, %(param_5)s]) AS anon_1
```

An instance of *array* will always have the datatype *ARRAY*. The "inner" type of the array is inferred from the values present, unless the `type_` keyword argument is passed:

```
array(['foo', 'bar'], type_=CHAR)
```

New in version 0.8: Added the *array* literal type.

See also:

postgresql.ARRAY

class sqlalchemy.dialects.postgresql.**ARRAY**(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)

PostgreSQL ARRAY type.

Changed in version 1.1: The *postgresql.ARRAY* type is now a subclass of the core `types.ARRAY` type.

The *postgresql.ARRAY* type is constructed in the same way as the core `types.ARRAY` type; a member type is required, and a number of dimensions is recommended if the type is to be used for more than one dimension:

```
from sqlalchemy.dialects import postgresql

mytable = Table("mytable", metadata,
        Column("data", postgresql.ARRAY(Integer, dimensions=2))
    )
```

The *postgresql.ARRAY* type provides all operations defined on the core `types.ARRAY` type, including support for "dimensions", indexed access, and simple matching such as `types.ARRAY.Comparator.any()` and `types.ARRAY.Comparator.all()`. *postgresql.ARRAY* class also provides PostgreSQL-specific methods for containment operations, including *postgresql.ARRAY.Comparator.contains()* *postgresql.ARRAY.Comparator.contained_by()*, and *postgresql.ARRAY.Comparator.overlap()*, e.g.:

```
mytable.c.data.contains([1, 2])
```

The *postgresql.ARRAY* type may not be supported on all PostgreSQL DBAPIs; it is currently known to work on psycopg2 only.

Additionally, the *postgresql.ARRAY* type does not work directly in conjunction with the *ENUM* type. For a workaround, see the special type at *Using ENUM with ARRAY*.

See also:

`types.ARRAY` - base array type

postgresql.array - produces a literal array value.

class Comparator(*expr*)

Define comparison operations for *ARRAY*.

Note that these operations are in addition to those provided by the base `types.ARRAY.Comparator` class, including `types.ARRAY.Comparator.any()` and `types.ARRAY.Comparator.all()`.

contained_by(*other*)

Boolean expression. Test if elements are a proper subset of the elements of the argument array expression.

contains(*other*, ***kwargs*)

Boolean expression. Test if elements are a superset of the elements of the argument array expression.

overlap(*other*)

Boolean expression. Test if array has elements in common with an argument array expression.

__init__(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=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.
- **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.

- **dimensions** – if non-None, the ARRAY will assume a fixed number of dimensions. This will cause the DDL emitted for this ARRAY to include the exact number of bracket clauses `[]`, and will also optimize the performance of the type overall. Note that PG arrays are always implicitly "non-dimensioned", meaning they can store any number of dimensions no matter how they were declared.
- **zero_indexes=False** – when True, index values will be converted between Python zero-based and PostgreSQL one-based indexes, e.g. a value of one will be added to all index values before passing to the database.

  New in version 0.9.5.

sqlalchemy.dialects.postgresql.**array_agg**(**arg*, ***kw*)
PostgreSQL-specific form of `array_agg`, ensures return type is *postgresql.ARRAY* and not the plain `types.ARRAY`.

New in version 1.1.

sqlalchemy.dialects.postgresql.**Any**(*other*, *arrexpr*, *operator=<built-in function eq>*)
A synonym for the `ARRAY.Comparator.any()` method.

This method is legacy and is here for backwards-compatibility.

See also:

`expression.any_()`

sqlalchemy.dialects.postgresql.**All**(*other*, *arrexpr*, *operator=<built-in function eq>*)
A synonym for the `ARRAY.Comparator.all()` method.

This method is legacy and is here for backwards-compatibility.

See also:

`expression.all_()`

class sqlalchemy.dialects.postgresql.**BIT**(*length=None*, *varying=False*)

class sqlalchemy.dialects.postgresql.**BYTEA**(*length=None*)

__init__(*length=None*)
Construct a LargeBinary type.

Parameters **length** – optional, a length for the column for use in DDL statements, for those binary types that accept a length, such as the MySQL BLOB type.

class sqlalchemy.dialects.postgresql.**CIDR**

class sqlalchemy.dialects.postgresql.**DOUBLE_PRECISION**(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)

__init__(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***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.
- **decimal_return_scale** – Default scale to use when converting from floats to Python decimals. Floating point values will typically be much

longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Note that the MySQL float types, which do include "scale", will use "scale" as the default for decimal_return_scale, if not otherwise specified.

New in version 0.9.0.

- **`**kwargs`** – deprecated. Additional arguments here are ignored by the default `Float` type. For database specific floats that support additional arguments, see that dialect's documentation for details, such as *sqlalchemy.dialects.mysql.FLOAT*.

class sqlalchemy.dialects.postgresql.**ENUM**(**enums*, ***kw*)

PostgreSQL ENUM type.

This is a subclass of `types.Enum` which includes support for PG's `CREATE TYPE` and `DROP TYPE`.

When the builtin type `types.Enum` is used and the `Enum.native_enum` flag is left at its default of True, the PostgreSQL backend will use a *postgresql.ENUM* type as the implementation, so the special create/drop rules will be used.

The create/drop behavior of ENUM is necessarily intricate, due to the awkward relationship the ENUM type has in relationship to the parent table, in that it may be "owned" by just a single table, or may be shared among many tables.

When using `types.Enum` or *postgresql.ENUM* in an "inline" fashion, the `CREATE TYPE` and `DROP TYPE` is emitted corresponding to when the `Table.create()` and `Table.drop()` methods are called:

```
table = Table('sometable', metadata,
    Column('some_enum', ENUM('a', 'b', 'c', name='myenum'))
)

table.create(engine)  # will emit CREATE ENUM and CREATE TABLE
table.drop(engine)  # will emit DROP TABLE and DROP ENUM
```

To use a common enumerated type between multiple tables, the best practice is to declare the `types.Enum` or *postgresql.ENUM* independently, and associate it with the `MetaData` object itself:

```
my_enum = ENUM('a', 'b', 'c', name='myenum', metadata=metadata)

t1 = Table('sometable_one', metadata,
    Column('some_enum', myenum)
)

t2 = Table('sometable_two', metadata,
    Column('some_enum', myenum)
)
```

When this pattern is used, care must still be taken at the level of individual table creates. Emitting CREATE TABLE without also specifying `checkfirst=True` will still cause issues:

```
t1.create(engine) # will fail: no such type 'myenum'
```

If we specify `checkfirst=True`, the individual table-level create operation will check for the `ENUM` and create if not exists:

```
# will check if enum exists, and emit CREATE TYPE if not
t1.create(engine, checkfirst=True)
```

When using a metadata-level ENUM type, the type will always be created and dropped if either the metadata-wide create/drop is called:

```
metadata.create_all(engine)  # will emit CREATE TYPE
metadata.drop_all(engine)  # will emit DROP TYPE
```

The type can also be created and dropped directly:

```
my_enum.create(engine)
my_enum.drop(engine)
```

Changed in version 1.0.0: The PostgreSQL *postgresql.ENUM* type now behaves more strictly with regards to CREATE/DROP. A metadata-level ENUM type will only be created and dropped at the metadata level, not the table level, with the exception of `table.create(checkfirst=True)`. The `table.drop()` call will now emit a DROP TYPE for a table-level enumerated type.

`__init__`(**enums*, ***kw*)
Construct an *ENUM*.

Arguments are the same as that of `types.Enum`, but also including the following parameters.

Parameters `create_type` – Defaults to True. Indicates that `CREATE TYPE` should be emitted, after optionally checking for the presence of the type, when the parent table is being created; and additionally that `DROP TYPE` is called when the table is dropped. When `False`, no check will be performed and no `CREATE TYPE` or `DROP TYPE` is emitted, unless *create()* or *drop()* are called directly. Setting to `False` is helpful when invoking a creation scheme to a SQL file without access to the actual database - the *create()* and *drop()* methods can be used to emit SQL to a target bind.

New in version 0.7.4.

`create`(*bind=None*, *checkfirst=True*)
Emit `CREATE TYPE` for this *ENUM*.

If the underlying dialect does not support PostgreSQL CREATE TYPE, no action is taken.

Parameters

- `bind` – a connectable `Engine`, `Connection`, or similar object to emit SQL.
- `checkfirst` – if `True`, a query against the PG catalog will be first performed to see if the type does not exist already before creating.

`drop`(*bind=None*, *checkfirst=True*)
Emit `DROP TYPE` for this *ENUM*.

If the underlying dialect does not support PostgreSQL DROP TYPE, no action is taken.

Parameters

- `bind` – a connectable `Engine`, `Connection`, or similar object to emit SQL.
- `checkfirst` – if `True`, a query against the PG catalog will be first performed to see if the type actually exists before dropping.

`class sqlalchemy.dialects.postgresql.HSTORE`(*text_type=None*)
Represent the PostgreSQL HSTORE type.

The *HSTORE* type stores dictionaries containing strings, e.g.:

```
data_table = Table('data_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', HSTORE)
)

with engine.connect() as conn:
    conn.execute(
```

```
        data_table.insert(),
        data = {"key1": "value1", "key2": "value2"}
    )
```

HSTORE provides for a wide range of operations, including:

- Index operations:

```
data_table.c.data['some key'] == 'some value'
```

- Containment operations:

```
data_table.c.data.has_key('some key')

data_table.c.data.has_all(['one', 'two', 'three'])
```

- Concatenation:

```
data_table.c.data + {"k1": "v1"}
```

For a full list of special methods see `HSTORE.comparator_factory`.

For usage with the SQLAlchemy ORM, it may be desirable to combine the usage of *HSTORE* with `MutableDict` dictionary now part of the `sqlalchemy.ext.mutable` extension. This extension will allow "in-place" changes to the dictionary, e.g. addition of new keys or replacement/removal of existing keys to/from the current dictionary, to produce events which will be detected by the unit of work:

```
from sqlalchemy.ext.mutable import MutableDict

class MyClass(Base):
    __tablename__ = 'data_table'

    id = Column(Integer, primary_key=True)
    data = Column(MutableDict.as_mutable(HSTORE))

my_object = session.query(MyClass).one()

# in-place mutation, requires Mutable extension
# in order for the ORM to detect
my_object.data['some_key'] = 'some value'

session.commit()
```

When the `sqlalchemy.ext.mutable` extension is not used, the ORM will not be alerted to any changes to the contents of an existing dictionary, unless that dictionary value is re-assigned to the HSTORE-attribute itself, thus generating a change event.

New in version 0.8.

See also:

hstore - render the PostgreSQL `hstore()` function.

class Comparator(*expr*)

Define comparison operations for *HSTORE*.

array()

Text array expression. Returns array of alternating keys and values.

contained_by(*other*)

Boolean expression. Test if keys are a proper subset of the keys of the argument jsonb expression.

contains(*other*, ***kwargs*)

Boolean expression. Test if keys (or array) are a superset of/contained the keys of the argument jsonb expression.

defined(*key*)
Boolean expression. Test for presence of a non-NULL value for the key. Note that the key may be a SQLA expression.

delete(*key*)
HStore expression. Returns the contents of this hstore with the given key deleted. Note that the key may be a SQLA expression.

has_all(*other*)
Boolean expression. Test for presence of all keys in jsonb

has_any(*other*)
Boolean expression. Test for presence of any key in jsonb

has_key(*other*)
Boolean expression. Test for presence of a key. Note that the key may be a SQLA expression.

keys()
Text array expression. Returns array of keys.

matrix()
Text array expression. Returns array of [key, value] pairs.

slice(*array*)
HStore expression. Returns a subset of an hstore defined by array of keys.

vals()
Text array expression. Returns array of values.

comparator_factory
alias of *Comparator*

class sqlalchemy.dialects.postgresql.**hstore**(**args*, ***kwargs*)
Construct an hstore value within a SQL expression using the PostgreSQL `hstore()` function.

The *hstore* function accepts one or two arguments as described in the PostgreSQL documentation.

E.g.:

```
from sqlalchemy.dialects.postgresql import array, hstore

select([hstore('key1', 'value1')])

select([
        hstore(
            array(['key1', 'key2', 'key3']),
            array(['value1', 'value2', 'value3'])
        )
    ])
```

New in version 0.8.

See also:

HSTORE - the PostgreSQL `HSTORE` datatype.

type
alias of *HSTORE*

class sqlalchemy.dialects.postgresql.**INET**

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.postgresql.**INTERVAL**(*precision=None*, *fields=None*)
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*, *fields=None*)
Construct an INTERVAL.

Parameters

- **precision** – optional integer precision value
- **fields** – string fields specifier. allows storage of fields to be limited, such as `"YEAR"`, `"MONTH"`, `"DAY TO HOUR"`, etc.

  New in version 1.2.

`class sqlalchemy.dialects.postgresql.JSON`(*none_as_null=False*, *astext_type=None*)
Represent the PostgreSQL JSON type.

This type is a specialization of the Core-level `types.JSON` type. Be sure to read the documentation for `types.JSON` for important tips regarding treatment of NULL values and ORM use.

Changed in version 1.1: *postgresql.JSON* is now a PostgreSQL- specific specialization of the new `types.JSON` type.

The operators provided by the PostgreSQL version of *JSON* include:

- Index operations (the `->` operator):

```
data_table.c.data['some key']

data_table.c.data[5]
```

- Index operations returning text (the `->>` operator):

```
data_table.c.data['some key'].astext == 'some value'
```

- Index operations with CAST (equivalent to `CAST(col ->> ['some key'] AS <type>)`):

```
data_table.c.data['some key'].astext.cast(Integer) == 5
```

- Path index operations (the `#>` operator):

```
data_table.c.data[('key_1', 'key_2', 5, ..., 'key_n')]
```

- Path index operations returning text (the `#>>` operator):

```
data_table.c.data[('key_1', 'key_2', 5, ..., 'key_n')].astext == 'some value'
```

Changed in version 1.1: The `ColumnElement.cast()` operator on JSON objects now requires that the *JSON.Comparator.astext* modifier be called explicitly, if the cast works only from a textual string.

Index operations return an expression object whose type defaults to *JSON* by default, so that further JSON-oriented instructions may be called upon the result type.

Custom serializers and deserializers are specified at the dialect level, that is using `create_engine()`. The reason for this is that when using psycopg2, the DBAPI only allows serializers at the per-cursor or per-connection level. E.g.:

```
engine = create_engine("postgresql://scott:tiger@localhost/test",
                       json_serializer=my_serialize_fn,
                       json_deserializer=my_deserialize_fn
                )
```

When using the psycopg2 dialect, the json_deserializer is registered against the database using `psycopg2.extras.register_default_json`.

See also:

`types.JSON` - Core level JSON type

JSONB

class Comparator(*expr*)
Define comparison operations for *JSON*.

astext
On an indexed expression, use the "astext" (e.g. "->>") conversion when rendered in SQL.

E.g.:

```
select([data_table.c.data['some key'].astext])
```

See also:

`ColumnElement.cast()`

comparator_factory
alias of *Comparator*

class sqlalchemy.dialects.postgresql.JSONB(*none_as_null=False*, *astext_type=None*)
Represent the PostgreSQL JSONB type.

The *JSONB* type stores arbitrary JSONB format data, e.g.:

```
data_table = Table('data_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', JSONB)
)

with engine.connect() as conn:
    conn.execute(
        data_table.insert(),
        data = {"key1": "value1", "key2": "value2"}
    )
```

The *JSONB* type includes all operations provided by *JSON*, including the same behaviors for indexing operations. It also adds additional operators specific to JSONB, including *JSONB.Comparator.has_key()*, *JSONB.Comparator.has_all()*, *JSONB.Comparator.has_any()*, *JSONB.Comparator.contains()*, and *JSONB.Comparator.contained_by()*.

Like the *JSON* type, the *JSONB* type does not detect in-place changes when used with the ORM, unless the `sqlalchemy.ext.mutable` extension is used.

Custom serializers and deserializers are shared with the *JSON* class, using the `json_serializer` and `json_deserializer` keyword arguments. These must be specified at the dialect level using `create_engine()`. When using psycopg2, the serializers are associated with the jsonb type using `psycopg2.extras.register_default_jsonb` on a per-connection basis, in the same way that `psycopg2.extras.register_default_json` is used to register these handlers with the json type.

New in version 0.9.7.

See also:

JSON

class Comparator(*expr*)
Define comparison operations for *JSON*.

contained_by(*other*)
Boolean expression. Test if keys are a proper subset of the keys of the argument jsonb expression.

contains(*other*, ***kwargs*)
Boolean expression. Test if keys (or array) are a superset of/contained the keys of the argument jsonb expression.

has_all(*other*)
Boolean expression. Test for presence of all keys in jsonb

has_any(*other*)
Boolean expression. Test for presence of any key in jsonb

has_key(*other*)
Boolean expression. Test for presence of a key. Note that the key may be a SQLA expression.

comparator_factory
alias of *Comparator*

class sqlalchemy.dialects.postgresql.MACADDR

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.postgresql.MONEY
Provide the PostgreSQL MONEY type.

New in version 1.2.

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.postgresql.OID
Provide the PostgreSQL OID type.

New in version 0.9.5.

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.postgresql.REAL(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
The SQL REAL type.

__init__(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***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.
- **decimal_return_scale** – Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Note that the MySQL float types, which do include "scale", will use "scale" as the default for decimal_return_scale, if not otherwise specified.

  New in version 0.9.0.
- ****kwargs** – deprecated. Additional arguments here are ignored by the default `Float` type. For database specific floats that support additional arguments, see that dialect's documentation for details, such as *sqlalchemy.dialects.mysql.FLOAT*.

class sqlalchemy.dialects.postgresql.TSVECTOR
The *postgresql.TSVECTOR* type implements the PostgreSQL text search type TSVECTOR.

It can be used to do full text queries on natural language documents.

New in version 0.9.0.

See also:

Full Text Search

__init__
Initialize self. See help(type(self)) for accurate signature.

class sqlalchemy.dialects.postgresql.UUID(*as_uuid=False*)
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.

Range Types

The new range column types found in PostgreSQL 9.2 onwards are catered for by the following types:

class sqlalchemy.dialects.postgresql.INT4RANGE
Represent the PostgreSQL INT4RANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.INT8RANGE
Represent the PostgreSQL INT8RANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.NUMRANGE
Represent the PostgreSQL NUMRANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.DATERANGE
Represent the PostgreSQL DATERANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.TSRANGE
Represent the PostgreSQL TSRANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.TSTZRANGE
Represent the PostgreSQL TSTZRANGE type.

New in version 0.8.2.

The types above get most of their functionality from the following mixin:

class sqlalchemy.dialects.postgresql.ranges.RangeOperators
This mixin provides functionality for the Range Operators listed in Table 9-44 of the postgres documentation for Range Functions and Operators. It is used by all the range types provided in the `postgres` dialect and can likely be used for any range types you create yourself.

No extra support is provided for the Range Functions listed in Table 9-45 of the postgres documentation. For these, the normal `func()` object should be used.

New in version 0.8.2: Support for PostgreSQL RANGE operations.

class comparator_factory(*expr*)
Define comparison operations for range types.

adjacent_to(*other*)
Boolean expression. Returns true if the range in the column is adjacent to the range in the operand.

contained_by(*other*)
Boolean expression. Returns true if the column is contained within the right hand operand.

contains(*other*, ***kw*)
Boolean expression. Returns true if the right hand operand, which can be an element or a range, is contained within the column.

not_extend_left_of(*other*)
Boolean expression. Returns true if the range in the column does not extend left of the range in the operand.

not_extend_right_of(*other*)
Boolean expression. Returns true if the range in the column does not extend right of the range in the operand.

overlaps(*other*)
Boolean expression. Returns true if the column overlaps (has points in common with) the right hand operand.

strictly_left_of(*other*)
Boolean expression. Returns true if the column is strictly left of the right hand operand.

strictly_right_of(*other*)
Boolean expression. Returns true if the column is strictly right of the right hand operand.

Warning: The range type DDL support should work with any Postgres DBAPI driver, however the data types returned may vary. If you are using `psycopg2`, it's recommended to upgrade to version 2.5 or later before using these column types.

When instantiating models that use these column types, you should pass whatever data type is expected by the DBAPI driver you're using for the column type. For `psycopg2` these are `psycopg2.extras.NumericRange`, `psycopg2.extras.DateRange`, `psycopg2.extras.DateTimeRange` and `psycopg2.extras.DateTimeTZRange` or the class you've registered with `psycopg2.extras.register_range`.

For example:

```
from psycopg2.extras import DateTimeRange
from sqlalchemy.dialects.postgresql import TSRANGE

class RoomBooking(Base):

    __tablename__ = 'room_booking'

    room = Column(Integer(), primary_key=True)
    during = Column(TSRANGE())

booking = RoomBooking(
    room=101,
    during=DateTimeRange(datetime(2013, 3, 23), None)
)
```

PostgreSQL Constraint Types

SQLAlchemy supports PostgreSQL EXCLUDE constraints via the *ExcludeConstraint* class:

class sqlalchemy.dialects.postgresql.ExcludeConstraint(**elements*, ***kw*)
A table-level EXCLUDE constraint.

Defines an EXCLUDE constraint as described in the postgres documentation.

__init__(**elements*, ***kw*)
Create an *ExcludeConstraint* object.

E.g.:

```
const = ExcludeConstraint(
    (Column('period'), '&&'),
    (Column('group'), '='),
    where=(Column('group') != 'some group')
)
```

The constraint is normally embedded into the `Table` construct directly, or added later using `append_constraint()`:

```
some_table = Table(
    'some_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('period', TSRANGE()),
    Column('group', String)
)

some_table.append_constraint(
    ExcludeConstraint(
        (some_table.c.period, '&&'),
        (some_table.c.group, '='),
        where=some_table.c.group != 'some group',
        name='some_table_excl_const'
    )
)
```

Parameters

- ***elements** – A sequence of two tuples of the form `(column, operator)` where "column" is a SQL expression element or a raw SQL string, most typically a `Column` object, and "operator" is a string containing the operator to use.

  Note: A plain string passed for the value of "column" is interpreted as an arbitrary SQL expression; when passing a plain string, any necessary quoting and escaping syntaxes must be applied manually. In order to specify a column name when a `Column` object is not available, while ensuring that any necessary quoting rules take effect, an ad-hoc `Column` or `sql.expression.column()` object may be used.

- **name** – Optional, the in-database name of this constraint.
- **deferrable** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **initially** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- **using** – Optional string. If set, emit USING <index_method> when issuing DDL for this constraint. Defaults to 'gist'.

- **where** – Optional SQL expression construct or literal SQL string. If set, emit WHERE <predicate> when issuing DDL for this constraint.

  Note: A plain string passed here is interpreted as an arbitrary SQL expression; when passing a plain string, any necessary quoting and escaping syntaxes must be applied manually.

For example:

```
from sqlalchemy.dialects.postgresql import ExcludeConstraint, TSRANGE

class RoomBooking(Base):

    __tablename__ = 'room_booking'

    room = Column(Integer(), primary_key=True)
    during = Column(TSRANGE())

    __table_args__ = (
        ExcludeConstraint(('room', '='), ('during', '&&')),
    )
```

PostgreSQL DML Constructs

sqlalchemy.dialects.postgresql.dml.**insert**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)

Construct a new *Insert* object.

This constructor is mirrored as a public API function; see **insert()** for a full usage and argument description.

class sqlalchemy.dialects.postgresql.dml.**Insert**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)

PostgreSQL-specific implementation of INSERT.

Adds methods for PG-specific syntaxes such as ON CONFLICT.

New in version 1.1.

excluded

Provide the **excluded** namespace for an ON CONFLICT statement

PG's ON CONFLICT clause allows reference to the row that would be inserted, known as **excluded**. This attribute provides all columns in this row to be referenaceable.

See also:

INSERT…ON CONFLICT (Upsert) - example of how to use *Insert.excluded*

on_conflict_do_nothing(*constraint=None*, *index_elements=None*, *index_where=None*)

Specifies a DO NOTHING action for ON CONFLICT clause.

The **constraint** and **index_elements** arguments are optional, but only one of these can be specified.

Parameters

- **constraint** – The name of a unique or exclusion constraint on the table, or the constraint object itself if it has a .name attribute.

- **index_elements** – A sequence consisting of string column names, `Column` objects, or other column expression objects that will be used to infer a target index.
- **index_where** – Additional WHERE criterion that can be used to infer a conditional target index.

  New in version 1.1.

See also:

INSERT…ON CONFLICT (Upsert)

`on_conflict_do_update`(*constraint=None*, *index_elements=None*, *index_where=None*, *set_=None*, *where=None*)

Specifies a DO UPDATE SET action for ON CONFLICT clause.

Either the `constraint` or `index_elements` argument is required, but only one of these can be specified.

Parameters

- **constraint** – The name of a unique or exclusion constraint on the table, or the constraint object itself if it has a .name attribute.
- **index_elements** – A sequence consisting of string column names, `Column` objects, or other column expression objects that will be used to infer a target index.
- **index_where** – Additional WHERE criterion that can be used to infer a conditional target index.
- **set_** – Required argument. A dictionary or other mapping object with column names as keys and expressions or literals as values, specifying the `SET` actions to take. If the target `Column` specifies a ".key" attribute distinct from the column name, that key should be used.

  Warning: This dictionary does **not** take into account Python-specified default UPDATE values or generation functions, e.g. those specified using `Column.onupdate`. These values will not be exercised for an ON CONFLICT style of UPDATE, unless they are manually specified in the *`Insert.on_conflict_do_update.set_`* dictionary.

- **where** – Optional argument. If present, can be a literal SQL string or an acceptable expression for a `WHERE` clause that restricts the rows affected by `DO UPDATE SET`. Rows not meeting the `WHERE` condition will not be updated (effectively a `DO NOTHING` for those rows).

  New in version 1.1.

See also:

INSERT…ON CONFLICT (Upsert)

psycopg2

psycopg2 Connect Arguments

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. Note that the `Connection.execution_options.stream_results` execution option is a more targeted way of enabling this mode on a per-execution basis.

- `use_native_unicode`: Enable the usage of Psycopg2 "native unicode" mode per connection. True by default.

  See also:

  Disabling Native Unicode

- `isolation_level`: This option, available for all PostgreSQL dialects, includes the `AUTOCOMMIT` isolation level when using the psycopg2 dialect.

  See also:

  Psycopg2 Transaction Isolation Level

- `client_encoding`: sets the client encoding in a libpq-agnostic way, using psycopg2's `set_client_encoding()` method.

  See also:

  Unicode with Psycopg2

- `use_batch_mode`: This flag allows `psycopg2.extras.execute_batch` for `cursor.executemany()` calls performed by the `Engine`. It is currently experimental but may well become True by default as it is critical for executemany performance.

  See also:

  Psycopg2 Batch Mode (Fast Execution)

Unix Domain Connections

psycopg2 supports connecting via Unix domain connections. When the `host` portion of the URL is omitted, SQLAlchemy passes `None` to psycopg2, which specifies Unix-domain communication rather than TCP/IP communication:

```
create_engine("postgresql+psycopg2://user:password@/dbname")
```

By default, the socket file used is to connect to a Unix-domain socket in `/tmp`, or whatever socket directory was specified when PostgreSQL was built. This value can be overridden by passing a pathname to psycopg2, using `host` as an additional keyword argument:

```
create_engine("postgresql+psycopg2://user:password@/dbname?\
```

host=/var/lib/postgresql")

See also:

PQconnectdbParams

Per-Statement/Connection Execution Options

The following DBAPI-specific options are respected when used with `Connection.execution_options()`, `Executable.execution_options()`, `Query.execution_options()`, in addition to those not specific to DBAPIs:

- `isolation_level` - Set the transaction isolation level for the lifespan of a `Connection` (can only be set on a connection, not a statement or query). See *Psycopg2 Transaction Isolation Level*.

- `stream_results` - Enable or disable usage of psycopg2 server side cursors - this feature makes use of "named" cursors in combination with special result handling methods so that result rows are not fully buffered. If `None` or not set, the `server_side_cursors` option of the `Engine` is used.
- `max_row_buffer` - when using `stream_results`, an integer value that specifies the maximum number of rows to buffer at a time. This is interpreted by the `BufferedRowResultProxy`, and if omitted the buffer will grow to ultimately store 1000 rows at a time.

  New in version 1.0.6.

Psycopg2 Batch Mode (Fast Execution)

Modern versions of psycopg2 include a feature known as Fast Execution Helpers, which have been shown in benchmarking to improve psycopg2's executemany() performance with INSERTS by multiple orders of magnitude. SQLAlchemy allows this extension to be used for all `executemany()` style calls invoked by an `Engine` when used with multiple parameter sets, by adding the `use_batch_mode` flag to `create_engine()`:

```
engine = create_engine(
    "postgresql+psycopg2://scott:tiger@host/dbname",
    use_batch_mode=True)
```

Batch mode is considered to be **experimental** at this time, however may be enabled by default in a future release.

New in version 1.2.0.

Unicode with Psycopg2

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. Psycopg2 here will encode/decode string values based on the current "client encoding" setting; by default this is the value in the `postgresql.conf` file, which often defaults to `SQL_ASCII`. Typically, this can be changed to `utf8`, as a more useful default:

```
# postgresql.conf file

# client_encoding = sql_ascii # actually, defaults to database
                              # encoding
client_encoding = utf8
```

A second way to affect the client encoding is to set it within Psycopg2 locally. SQLAlchemy will call psycopg2's `psycopg2:connection.set_client_encoding()` method on all new connections based on the value passed to `create_engine()` using the `client_encoding` parameter:

```
# set_client_encoding() setting;
# works for *all* PostgreSQL versions
engine = create_engine("postgresql://user:pass@host/dbname",
                       client_encoding='utf8')
```

This overrides the encoding specified in the PostgreSQL client configuration. When using the parameter in this way, the psycopg2 driver emits `SET client_encoding TO 'utf8'` on the connection explicitly, and works in all PostgreSQL versions.

Note that the `client_encoding` setting as passed to `create_engine()` is **not the same** as the more recently added `client_encoding` parameter now supported by libpq directly. This is enabled when `client_encoding` is passed directly to `psycopg2.connect()`, and from SQLAlchemy is passed using the `create_engine.connect_args` parameter:

```
# libpq direct parameter setting;
# only works for PostgreSQL **9.1 and above**
engine = create_engine("postgresql://user:pass@host/dbname",
                       connect_args={'client_encoding': 'utf8'})

# using the query string is equivalent
engine = create_engine("postgresql://user:pass@host/dbname?client_encoding=utf8")
```

The above parameter was only added to libpq as of version 9.1 of PostgreSQL, so using the previous method is better for cross-version support.

Disabling Native Unicode

SQLAlchemy can also be instructed to skip the usage of the psycopg2 `UNICODE` extension and to instead utilize its own unicode encode/decode services, which are normally reserved only for those DBAPIs that don't fully support unicode directly. Passing `use_native_unicode=False` to `create_engine()` will disable usage of `psycopg2.extensions.UNICODE`. SQLAlchemy will instead encode data itself into Python bytestrings on the way in and coerce from bytes on the way back, using the value of the `create_engine()` `encoding` parameter, which defaults to `utf-8`. SQLAlchemy's own unicode encode/decode functionality is steadily becoming obsolete as most DBAPIs now support unicode fully.

Bound Parameter Styles

The default parameter style for the psycopg2 dialect is "pyformat", where SQL is rendered using `%(paramname)s` style. This format has the limitation that it does not accommodate the unusual case of parameter names that actually contain percent or parenthesis symbols; as SQLAlchemy in many cases generates bound parameter names based on the name of a column, the presence of these characters in a column name can lead to problems.

There are two solutions to the issue of a `schema.Column` that contains one of these characters in its name. One is to specify the `schema.Column.key` for columns that have such names:

```
measurement = Table('measurement', metadata,
    Column('Size (meters)', Integer, key='size_meters')
)
```

Above, an INSERT statement such as `measurement.insert()` will use `size_meters` as the parameter name, and a SQL expression such as `measurement.c.size_meters > 10` will derive the bound parameter name from the `size_meters` key as well.

Changed in version 1.0.0: - SQL expressions will use `Column.key` as the source of naming when anonymous bound parameters are created in SQL expressions; previously, this behavior only applied to `Table.insert()` and `Table.update()` parameter names.

The other solution is to use a positional format; psycopg2 allows use of the "format" paramstyle, which can be passed to `create_engine.paramstyle`:

```
engine = create_engine(
    'postgresql://scott:tiger@localhost:5432/test', paramstyle='format')
```

With the above engine, instead of a statement like:

```
INSERT INTO measurement ("Size (meters)") VALUES (%(Size (meters))s)
{'Size (meters)': 1}
```

we instead see:

```
INSERT INTO measurement ("Size (meters)") VALUES (%s)
(1, )
```

Where above, the dictionary style is converted into a tuple with positional style.

Transactions

The psycopg2 dialect fully supports SAVEPOINT and two-phase commit operations.

Psycopg2 Transaction Isolation Level

As discussed in *Transaction Isolation Level*, all PostgreSQL dialects support setting of transaction isolation level both via the `isolation_level` parameter passed to `create_engine()`, as well as the `isolation_level` argument used by `Connection.execution_options()`. When using the psycopg2 dialect, these options make use of psycopg2's `set_isolation_level()` connection method, rather than emitting a PostgreSQL directive; this is because psycopg2's API-level setting is always emitted at the start of each transaction in any case.

The psycopg2 dialect supports these constants for isolation level:

- `READ COMMITTED`
- `READ UNCOMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`
- `AUTOCOMMIT`

New in version 0.8.2: support for AUTOCOMMIT isolation level when using psycopg2.

See also:

Transaction Isolation Level

pg8000 Transaction Isolation Level

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)
```

HSTORE type

The `psycopg2` DBAPI includes an extension to natively handle marshalling of the HSTORE type. The SQLAlchemy psycopg2 dialect will enable this extension by default when psycopg2 version 2.4 or greater is used, and it is detected that the target database has the HSTORE type set up for use. In other words, when the dialect makes the first connection, a sequence like the following is performed:

1. Request the available HSTORE oids using `psycopg2.extras.HstoreAdapter.get_oids()`. If this function returns a list of HSTORE identifiers, we then determine that the `HSTORE` extension is present. This function is **skipped** if the version of psycopg2 installed is less than version 2.4.
2. If the `use_native_hstore` flag is at its default of `True`, and we've detected that `HSTORE` oids are available, the `psycopg2.extensions.register_hstore()` extension is invoked for all connections.

The `register_hstore()` extension has the effect of **all Python dictionaries being accepted as parameters regardless of the type of target column in SQL**. The dictionaries are converted by this extension into a textual HSTORE expression. If this behavior is not desired, disable the use of the hstore extension by setting `use_native_hstore` to `False` as follows:

```
engine = create_engine("postgresql+psycopg2://scott:tiger@localhost/test",
            use_native_hstore=False)
```

The HSTORE type is **still supported** when the psycopg2.extensions.register_hstore() extension is not used. It merely means that the coercion between Python dictionaries and the HSTORE string format, on both the parameter side and the result side, will take place within SQLAlchemy's own marshalling logic, and not that of psycopg2 which may be more performant.

pg8000

Unicode

pg8000 will encode / decode string values between it and the server using the PostgreSQL client_encoding parameter; by default this is the value in the postgresql.conf file, which often defaults to SQL_ASCII. Typically, this can be changed to utf-8, as a more useful default:

```
#client_encoding = sql_ascii # actually, defaults to database
                             # encoding
client_encoding = utf8
```

The client_encoding can be overridden for a session by executing the SQL:

SET CLIENT_ENCODING TO 'utf8';

SQLAlchemy will execute this SQL on all new connections based on the value passed to create_engine() using the client_encoding parameter:

```
engine = create_engine(
    "postgresql+pg8000://user:pass@host/dbname", client_encoding='utf8')
```

pg8000 Transaction Isolation Level

The pg8000 dialect offers the same isolation level settings as that of the *psycopg2* dialect:

- READ COMMITTED
- READ UNCOMMITTED
- REPEATABLE READ
- SERIALIZABLE
- AUTOCOMMIT

New in version 0.9.5: support for AUTOCOMMIT isolation level when using pg8000.

See also:

Transaction Isolation Level

Psycopg2 Transaction Isolation Level

psycopg2cffi

psycopg2cffi is an adaptation of psycopg2, using CFFI for the C layer. This makes it suitable for use in e.g. PyPy. Documentation is as per psycopg2.

New in version 1.0.0.

See also:

sqlalchemy.dialects.postgresql.psycopg2

py-postgresql

pygresql

zxjdbc

4.1.6 SQLite

Date and Time Types

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

Ensuring Text affinity

The DDL rendered for these types is the standard `DATE`, `TIME` and `DATETIME` indicators. However, custom storage formats can also be applied to these types. When the storage format is detected as containing no alpha characters, the DDL for these types is rendered as `DATE_CHAR`, `TIME_CHAR`, and `DATETIME_CHAR`, so that the column continues to have textual affinity.

See also:

Type Affinity - in the SQLite documentation

SQLite Auto Incrementing Behavior

Background on SQLite's autoincrement is at: http://sqlite.org/autoinc.html

Key concepts:

- SQLite has an implicit "auto increment" feature that takes place for any non-composite primary-key column that is specifically created using "INTEGER PRIMARY KEY" for the type + primary key.
- SQLite also has an explicit "AUTOINCREMENT" keyword, that is **not** equivalent to the implicit autoincrement feature; this keyword is not recommended for general use. SQLAlchemy does not render this keyword unless a special SQLite-specific directive is used (see below). However, it still requires that the column's type is named "INTEGER".

Using the AUTOINCREMENT Keyword

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)
```

Allowing autoincrement behavior SQLAlchemy types other than Integer/INTEGER

SQLite's typing model is based on naming conventions. Among other things, this means that any type name which contains the substring `"INT"` will be determined to be of "integer affinity". A type

named "BIGINT", "SPECIAL_INT" or even "XYZINTQPR", will be considered by SQLite to be of "integer" affinity. However, **the SQLite autoincrement feature, whether implicitly or explicitly enabled, requires that the name of the column's type is exactly the string "INTEGER"**. Therefore, if an application uses a type like BigInteger for a primary key, on SQLite this type will need to be rendered as the name "INTEGER" when emitting the initial CREATE TABLE statement in order for the autoincrement behavior to be available.

One approach to achieve this is to use Integer on SQLite only using TypeEngine.with_variant():

```
table = Table(
    "my_table", metadata,
    Column("id", BigInteger().with_variant(Integer, "sqlite"), primary_key=True)
)
```

Another is to use a subclass of BigInteger that overrides its DDL name to be INTEGER when compiled against SQLite:

```
from sqlalchemy import BigInteger
from sqlalchemy.ext.compiler import compiles

class SLBigInteger(BigInteger):
    pass

@compiles(SLBigInteger, 'sqlite')
def bi_c(element, compiler, **kw):
    return "INTEGER"

@compiles(SLBigInteger)
def bi_c(element, compiler, **kw):
    return compiler.visit_BIGINT(element, **kw)


table = Table(
    "my_table", metadata,
    Column("id", SLBigInteger(), primary_key=True)
)
```

See also:

TypeEngine.with_variant()

sqlalchemy.ext.compiler_toplevel

Datatypes In SQLite Version 3

Database Locking Behavior / Concurrency

SQLite is not designed for a high level of write concurrency. The database itself, being a file, is locked completely during write operations within transactions, meaning exactly one "connection" (in reality a file handle) has exclusive access to the database during this period - all other "connections" will be blocked during this time.

The Python DBAPI specification also calls for a connection model that is always in a transaction; there is no connection.begin() method, only connection.commit() and connection.rollback(), upon which a new transaction is to be begun immediately. This may seem to imply that the SQLite driver would in theory allow only a single filehandle on a particular database file at any time; however, there are several factors both within SQlite itself as well as within the pysqlite driver which loosen this restriction significantly.

However, no matter what locking modes are used, SQLite will still always lock the database file once a transaction is started and DML (e.g. INSERT, UPDATE, DELETE) has at least been emitted, and this will block other transactions at least at the point that they also attempt to emit DML. By default, the length of time on this block is very short before it times out with an error.

This behavior becomes more critical when used in conjunction with the SQLAlchemy ORM. SQLAlchemy's `Session` object by default runs within a transaction, and with its autoflush model, may emit DML preceding any SELECT statement. This may lead to a SQLite database that locks more quickly than is expected. The locking mode of SQLite and the pysqlite driver can be manipulated to some degree, however it should be noted that achieving a high degree of write-concurrency with SQLite is a losing battle.

For more information on SQLite's lack of write concurrency by design, please see Situations Where Another RDBMS May Work Better - High Concurrency near the bottom of the page.

The following subsections introduce areas that are impacted by SQLite's file-based architecture and additionally will usually require workarounds to work when using the pysqlite driver.

Transaction Isolation Level

SQLite supports "transaction isolation" in a non-standard way, along two axes. One is that of the PRAGMA read_uncommitted instruction. This setting can essentially switch SQLite between its default mode of `SERIALIZABLE` isolation, and a "dirty read" isolation mode normally referred to as `READ UNCOMMITTED`.

SQLAlchemy ties into this PRAGMA statement using the `create_engine.isolation_level` parameter of `create_engine()`. Valid values for this parameter when used with SQLite are `"SERIALIZABLE"` and `"READ UNCOMMITTED"` corresponding to a value of 0 and 1, respectively. SQLite defaults to `SERIALIZABLE`, however its behavior is impacted by the pysqlite driver's default behavior.

The other axis along which SQLite's transactional locking is impacted is via the nature of the `BEGIN` statement used. The three varieties are "deferred", "immediate", and "exclusive", as described at BEGIN TRANSACTION. A straight `BEGIN` statement uses the "deferred" mode, where the the database file is not locked until the first read or write operation, and read access remains open to other transactions until the first write operation. But again, it is critical to note that the pysqlite driver interferes with this behavior by *not even emitting BEGIN* until the first write operation.

> **Warning:** SQLite's transactional scope is impacted by unresolved issues in the pysqlite driver, which defers BEGIN statements to a greater degree than is often feasible. See the section *Serializable isolation / Savepoints / Transactional DDL* for techniques to work around this behavior.

SAVEPOINT Support

SQLite supports SAVEPOINTs, which only function once a transaction is begun. SQLAlchemy's SAVEPOINT support is available using the `Connection.begin_nested()` method at the Core level, and `Session.begin_nested()` at the ORM level. However, SAVEPOINTs won't work at all with pysqlite unless workarounds are taken.

> **Warning:** SQLite's SAVEPOINT feature is impacted by unresolved issues in the pysqlite driver, which defers BEGIN statements to a greater degree than is often feasible. See the section *Serializable isolation / Savepoints / Transactional DDL* for techniques to work around this behavior.

Transactional DDL

The SQLite database supports transactional DDL as well. In this case, the pysqlite driver is not only failing to start transactions, it also is ending any existing transction when DDL is detected, so again, workarounds are required.

Warning: SQLite's transactional DDL is impacted by unresolved issues in the pysqlite driver, which fails to emit BEGIN and additionally forces a COMMIT to cancel any transaction when DDL is encountered. See the section *Serializable isolation / Savepoints / Transactional DDL* for techniques to work around this behavior.

Foreign Key Support

SQLite supports FOREIGN KEY syntax when emitting CREATE statements for tables, however by default these constraints have no effect on the operation of the table.

Constraint checking on SQLite has three prerequisites:

- At least version 3.6.19 of SQLite must be in use
- The SQLite library must be compiled *without* the SQLITE_OMIT_FOREIGN_KEY or SQLITE_OMIT_TRIGGER symbols enabled.
- The `PRAGMA foreign_keys = ON` statement must be emitted on all connections before use.

SQLAlchemy allows for the `PRAGMA` statement to be emitted automatically for new connections through the usage of events:

```
from sqlalchemy.engine import Engine
from sqlalchemy import event

@event.listens_for(Engine, "connect")
def set_sqlite_pragma(dbapi_connection, connection_record):
    cursor = dbapi_connection.cursor()
    cursor.execute("PRAGMA foreign_keys=ON")
    cursor.close()
```

Warning: When SQLite foreign keys are enabled, it is **not possible** to emit CREATE or DROP statements for tables that contain mutually-dependent foreign key constraints; to emit the DDL for these tables requires that ALTER TABLE be used to create or drop these constraints separately, for which SQLite has no support.

See also:

SQLite Foreign Key Support - on the SQLite web site.

event_toplevel - SQLAlchemy event API.

use_alter - more information on SQLAlchemy's facilities for handling mutually-dependent foreign key constraints.

Type Reflection

SQLite types are unlike those of most other database backends, in that the string name of the type usually does not correspond to a "type" in a one-to-one fashion. Instead, SQLite links per-column typing behavior to one of five so-called "type affinities" based on a string matching pattern for the type.

SQLAlchemy's reflection process, when inspecting types, uses a simple lookup table to link the keywords returned to provided SQLAlchemy types. This lookup table is present within the SQLite dialect as it is for all other dialects. However, the SQLite dialect has a different "fallback" routine for when a particular type name is not located in the lookup map; it instead implements the SQLite "type affinity" scheme located at http://www.sqlite.org/datatype3.html section 2.1.

The provided typemap will make direct associations from an exact string name match for the following types:

BIGINT, BLOB, BOOLEAN, BOOLEAN, CHAR, DATE, DATETIME, FLOAT, DECIMAL, FLOAT, INTEGER, INTEGER, NUMERIC, REAL, SMALLINT, TEXT, TIME, TIMESTAMP, VARCHAR, NVARCHAR, NCHAR

When a type name does not match one of the above types, the "type affinity" lookup is used instead:

- INTEGER is returned if the type name includes the string INT
- TEXT is returned if the type name includes the string CHAR, CLOB or TEXT
- NullType is returned if the type name includes the string BLOB
- REAL is returned if the type name includes the string REAL, FLOA or DOUB.
- Otherwise, the NUMERIC type is used.

New in version 0.9.3: Support for SQLite type affinity rules when reflecting columns.

Partial Indexes

A partial index, e.g. one which uses a WHERE clause, can be specified with the DDL system using the argument sqlite_where:

```
tbl = Table('testtbl', m, Column('data', Integer))
idx = Index('test_idx1', tbl.c.data,
            sqlite_where=and_(tbl.c.data > 5, tbl.c.data < 10))
```

The index will be rendered at create time as:

```
CREATE INDEX test_idx1 ON testtbl (data)
WHERE data > 5 AND data < 10
```

New in version 0.9.9.

Dotted Column Names

Using table or column names that explicitly have periods in them is **not recommended**. While this is generally a bad idea for relational databases in general, as the dot is a syntactically significant character, the SQLite driver up until version **3.10.0** of SQLite has a bug which requires that SQLAlchemy filter out these dots in result sets.

Changed in version 1.1: The following SQLite issue has been resolved as of version 3.10.0 of SQLite. SQLAlchemy as of **1.1** automatically disables its internal workarounds based on detection of this version.

The bug, entirely outside of SQLAlchemy, can be illustrated thusly:

```
import sqlite3

assert sqlite3.sqlite_version_info < (3, 10, 0), "bug is fixed in this version"

conn = sqlite3.connect(":memory:")
cursor = conn.cursor()

cursor.execute("create table x (a integer, b integer)")
cursor.execute("insert into x (a, b) values (1, 1)")
cursor.execute("insert into x (a, b) values (2, 2)")

cursor.execute("select x.a, x.b from x")
assert [c[0] for c in cursor.description] == ['a', 'b']

cursor.execute('''
    select x.a, x.b from x where a=1
    union
    select x.a, x.b from x where a=2
''')
```

```
assert [c[0] for c in cursor.description] == ['a', 'b'], \
    [c[0] for c in cursor.description]
```

The second assertion fails:

```
Traceback (most recent call last):
  File "test.py", line 19, in <module>
    [c[0] for c in cursor.description]
AssertionError: ['x.a', 'x.b']
```

Where above, the driver incorrectly reports the names of the columns including the name of the table, which is entirely inconsistent vs. when the UNION is not present.

SQLAlchemy relies upon column names being predictable in how they match to the original statement, so the SQLAlchemy dialect has no choice but to filter these out:

```
from sqlalchemy import create_engine

eng = create_engine("sqlite://")
conn = eng.connect()

conn.execute("create table x (a integer, b integer)")
conn.execute("insert into x (a, b) values (1, 1)")
conn.execute("insert into x (a, b) values (2, 2)")

result = conn.execute("select x.a, x.b from x")
assert result.keys() == ["a", "b"]

result = conn.execute('''
    select x.a, x.b from x where a=1
    union
    select x.a, x.b from x where a=2
''')
assert result.keys() == ["a", "b"]
```

Note that above, even though SQLAlchemy filters out the dots, *both names are still addressable*:

```
>>> row = result.first()
>>> row["a"]
1
>>> row["x.a"]
1
>>> row["b"]
1
>>> row["x.b"]
1
```

Therefore, the workaround applied by SQLAlchemy only impacts `ResultProxy.keys()` and `RowProxy.keys()` in the public API. In the very specific case where an application is forced to use column names that contain dots, and the functionality of `ResultProxy.keys()` and `RowProxy.keys()` is required to return these dotted names unmodified, the `sqlite_raw_colnames` execution option may be provided, either on a per-`Connection` basis:

```
result = conn.execution_options(sqlite_raw_colnames=True).execute('''
    select x.a, x.b from x where a=1
    union
    select x.a, x.b from x where a=2
''')
assert result.keys() == ["x.a", "x.b"]
```

or on a per-`Engine` basis:

```
engine = create_engine("sqlite://", execution_options={"sqlite_raw_colnames": True})
```

When using the per-`Engine` execution option, note that **Core and ORM queries that use UNION may not function properly**.

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
```

class sqlalchemy.dialects.sqlite.**DATETIME**(**args*, ***kwargs*)

Represent a Python datetime object in SQLite using a string.

The default string storage format is:

```
"%(year)04d-%(month)02d-%(day)02d %(hour)02d:%(min)02d:%(second)02d.%(microsecond)06d"
```

e.g.:

```
2011-03-15 12:05:57.10558
```

The storage format can be customized to some degree using the `storage_format` and `regexp` parameters, such as:

```
import re
from sqlalchemy.dialects.sqlite import DATETIME

dt = DATETIME(storage_format="%(year)04d/%(month)02d/%(day)02d "
                             "%(hour)02d:%(min)02d:%(second)02d",
              regexp=r"(\d+)/(\d+)/(\d+) (\d+)-(\d+)-(\d+)"
)
```

Parameters

- **storage_format** – format string which will be applied to the dict with keys year, month, day, hour, minute, second, and microsecond.
- **regexp** – regular expression which will be applied to incoming result rows. If the regexp contains named groups, the resulting match dict is applied to the Python datetime() constructor as keyword arguments. Otherwise, if positional groups are used, the datetime() constructor is called with positional arguments via `*map(int, match_obj.groups(0))`.

class sqlalchemy.dialects.sqlite.**DATE**(*storage_format=None*, *regexp=None*, ***kw*)

Represent a Python date object in SQLite using a string.

The default string storage format is:

```
"%(year)04d-%(month)02d-%(day)02d"
```

e.g.:

```
2011-03-15
```

The storage format can be customized to some degree using the `storage_format` and `regexp` parameters, such as:

```
import re
from sqlalchemy.dialects.sqlite import DATE

d = DATE(
        storage_format="%(month)02d/%(day)02d/%(year)04d",
        regexp=re.compile("(?P<month>\d+)/(?P<day>\d+)/(?P<year>\d+)")
    )
```

Parameters

- **storage_format** – format string which will be applied to the dict with keys year, month, and day.
- **regexp** – regular expression which will be applied to incoming result rows. If the regexp contains named groups, the resulting match dict is applied to the Python date() constructor as keyword arguments. Otherwise, if positional groups are used, the date() constructor is called with positional arguments via `*map(int, match_obj.groups(0))`.

class sqlalchemy.dialects.sqlite.TIME(*args, **kwargs)

Represent a Python time object in SQLite using a string.

The default string storage format is:

```
"%(hour)02d:%(minute)02d:%(second)02d.%(microsecond)06d"
```

e.g.:

```
12:05:57.10558
```

The storage format can be customized to some degree using the `storage_format` and `regexp` parameters, such as:

```
import re
from sqlalchemy.dialects.sqlite import TIME

t = TIME(storage_format="%(hour)02d-%(minute)02d-"
                        "%(second)02d-%(microsecond)06d",
         regexp=re.compile("(\d+)-(\d+)-(\d+)-(?:-(\d+))?")
)
```

Parameters

- **storage_format** – format string which will be applied to the dict with keys hour, minute, second, and microsecond.
- **regexp** – regular expression which will be applied to incoming result rows. If the regexp contains named groups, the resulting match dict is applied to the Python time() constructor as keyword arguments. Otherwise, if positional groups are used, the time() constructor is called with positional arguments via `*map(int, match_obj.groups(0))`.

Pysqlite

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)
```

Connect Strings

The file specification for the SQLite database is taken as the "database" portion of the URL. Note that the format of a SQLAlchemy 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_COLNAMES},
    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/Pooling Behavior

Pysqlite's default behavior is to prohibit the usage of a single connection in more than one thread. This is originally intended to work with older versions of SQLite that did not support multithreaded operation under various circumstances. In particular, older SQLite versions did not allow a `:memory:` database to be used in multiple threads under any circumstances.

Pysqlite does include a now-undocumented flag known as `check_same_thread` which will disable this check, however note that pysqlite connections are still not safe to use in concurrently in multiple threads. In particular, any statement execution calls would need to be externally mutexed, as Pysqlite does not provide for thread-safe propagation of error messages among other things. So while even `:memory:` databases can be shared among threads in modern SQLite, Pysqlite doesn't provide enough thread-safety to make this usage worth it.

SQLAlchemy sets up pooling to work with Pysqlite's default behavior:

- When a `:memory:` SQLite database is specified, the dialect by default will use `SingletonThreadPool`. This pool maintains a single connection per thread, so that all access to the engine within the current thread use the same `:memory:` database - other threads would access a different `:memory:` database.
- When a file-based database is specified, the dialect will use `NullPool` as the source of connections. This pool closes and discards connections which are returned to the pool immediately. SQLite file-based connections have extremely low overhead, so pooling is not necessary. The scheme also prevents a connection from being used again in a different thread and works best with SQLite's coarse-grained file locking.

  Changed in version 0.7: Default selection of `NullPool` for SQLite file-based databases. Previous versions select `SingletonThreadPool` by default for all SQLite databases.

Using a Memory Database in Multiple Threads

To use a `:memory:` database in a multithreaded scenario, the same connection object must be shared among threads, since the database exists only within the scope of that connection. The `StaticPool` implementation will maintain a single connection globally, and the `check_same_thread` flag can be passed to Pysqlite as `False`:

```
from sqlalchemy.pool import StaticPool
engine = create_engine('sqlite://',
                    connect_args={'check_same_thread':False},
                    poolclass=StaticPool)
```

Note that using a `:memory:` database in multiple threads requires a recent version of SQLite.

Using Temporary Tables with SQLite

Due to the way SQLite deals with temporary tables, if you wish to use a temporary table in a file-based SQLite database across multiple checkouts from the connection pool, such as when using an ORM `Session` where the temporary table should continue to remain after `Session.commit()` or `Session.rollback()` is called, a pool which maintains a single connection must be used. Use `SingletonThreadPool` if the scope is only needed within the current thread, or `StaticPool` is scope is needed within multiple threads for this case:

```
# maintain the same connection per thread
from sqlalchemy.pool import SingletonThreadPool
engine = create_engine('sqlite:///mydb.db',
                    poolclass=SingletonThreadPool)


# maintain the same connection across all threads
from sqlalchemy.pool import StaticPool
engine = create_engine('sqlite:///mydb.db',
                    poolclass=StaticPool)
```

Note that `SingletonThreadPool` should be configured for the number of threads that are to be used; beyond that number, connections will be closed out in a non deterministic way.

Unicode

The pysqlite driver only returns Python `unicode` objects in result sets, never plain strings, and accommodates `unicode` objects within bound parameter values in all cases. Regardless of the SQLAlchemy string type in use, string-based result values will by Python `unicode` in Python 2. The `Unicode` type should still be used to indicate those columns that require unicode, however, so that non-`unicode` values passed inadvertently will emit a warning. Pysqlite will emit an error if a non-`unicode` string is passed containing non-ASCII characters.

Serializable isolation / Savepoints / Transactional DDL

In the section *Database Locking Behavior / Concurrency*, we refer to the pysqlite driver's assortment of issues that prevent several features of SQLite from working correctly. The pysqlite DBAPI driver has several long-standing bugs which impact the correctness of its transactional behavior. In its default mode of operation, SQLite features such as SERIALIZABLE isolation, transactional DDL, and SAVEPOINT support are non-functional, and in order to use these features, workarounds must be taken.

The issue is essentially that the driver attempts to second-guess the user's intent, failing to start transactions and sometimes ending them prematurely, in an effort to minimize the SQLite databases's file locking behavior, even though SQLite itself uses "shared" locks for read-only activities.

SQLAlchemy chooses to not alter this behavior by default, as it is the long-expected behavior of the pysqlite driver; if and when the pysqlite driver attempts to repair these issues, that will be more of a driver towards defaults for SQLAlchemy.

The good news is that with a few events, we can implement transactional support fully, by disabling pysqlite's feature entirely and emitting BEGIN ourselves. This is achieved using two event listeners:

```
from sqlalchemy import create_engine, event

engine = create_engine("sqlite:///myfile.db")

@event.listens_for(engine, "connect")
def do_connect(dbapi_connection, connection_record):
    # disable pysqlite's emitting of the BEGIN statement entirely.
    # also stops it from emitting COMMIT before any DDL.
    dbapi_connection.isolation_level = None

@event.listens_for(engine, "begin")
def do_begin(conn):
    # emit our own BEGIN
    conn.execute("BEGIN")
```

Above, we intercept a new pysqlite connection and disable any transactional integration. Then, at the point at which SQLAlchemy knows that transaction scope is to begin, we emit `"BEGIN"` ourselves.

When we take control of "BEGIN", we can also control directly SQLite's locking modes, introduced at BEGIN TRANSACTION, by adding the desired locking mode to our "BEGIN":

```
@event.listens_for(engine, "begin")
def do_begin(conn):
    conn.execute("BEGIN EXCLUSIVE")
```

See also:

BEGIN TRANSACTION - on the SQLite site

sqlite3 SELECT does not BEGIN a transaction - on the Python bug tracker

sqlite3 module breaks transactions and potentially corrupts data - on the Python bug tracker

Pysqlcipher

Driver

The driver here is the pysqlcipher driver, which makes use of the SQLCipher engine. This system essentially introduces new PRAGMA commands to SQLite which allows the setting of a passphrase and other encryption parameters, allowing the database file to be encrypted.

pysqlcipher3 is a fork of *pysqlcipher* with support for Python 3, the driver is the same.

Connect Strings

The format of the connect string is in every way the same as that of the *pysqlite* driver, except that the "password" field is now accepted, which should contain a passphrase:

```
e = create_engine('sqlite+pysqlcipher://:testing@/foo.db')
```

For an absolute file path, two leading slashes should be used for the database name:

```
e = create_engine('sqlite+pysqlcipher://:testing@//path/to/foo.db')
```

A selection of additional encryption-related pragmas supported by SQLCipher as documented at https://www.zetetic.net/sqlcipher/sqlcipher-api/ can be passed in the query string, and will result in that PRAGMA being called for each new connection. Currently, `cipher`, `kdf_iter` `cipher_page_size` and `cipher_use_hmac` are supported:

```
e = create_engine('sqlite+pysqlcipher://:testing@/foo.db?cipher=aes-256-cfb&kdf_iter=64000')
```

Pooling Behavior

The driver makes a change to the default pool behavior of pysqlite as described in *Threading/Pooling Behavior*. The pysqlcipher driver has been observed to be significantly slower on connection than the pysqlite driver, most likely due to the encryption overhead, so the dialect here defaults to using the `SingletonThreadPool` implementation, instead of the `NullPool` pool used by pysqlite. As always, the pool implementation is entirely configurable using the `create_engine.poolclass` parameter; the `StaticPool` may be more feasible for single-threaded use, or `NullPool` may be used to prevent unencrypted connections from being held open for long periods of time, at the expense of slower startup time for new connections.

4.1.7 Sybase

Note: The Sybase dialect functions on current SQLAlchemy versions but is not regularly tested, and may have many issues and caveats not currently handled.

python-sybase

Unicode Support

The python-sybase driver does not appear to support non-ASCII strings of any kind at this time.

pyodbc

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
```

mxodbc

Note: This dialect is a stub only and is likely non functional at this time.

4.2 External Dialects

Changed in version 0.8: As of SQLAlchemy 0.8, several dialects have been moved to external projects, and dialects for new databases will also be published as external projects. The rationale here is to keep the base SQLAlchemy install and test suite from growing inordinately large.

The "classic" dialects such as SQLite, MySQL, PostgreSQL, Oracle, SQL Server, and Firebird will remain in the Core for the time being.

Changed in version 1.0: The Drizzle dialect has been moved into the third party system.

Current external dialect projects for SQLAlchemy include:

4.2.1 Production Ready

- ibm_db_sa - driver for IBM DB2 and Informix, developed jointly by IBM and SQLAlchemy developers.

- sqlalchemy-redshift - driver for Amazon Redshift, adapts the existing PostgreSQL/psycopg2 driver.
- sqlalchemy_exasol - driver for EXASolution.
- sqlalchemy-sqlany - driver for SAP Sybase SQL Anywhere, developed by SAP.
- sqlalchemy-monetdb - driver for MonetDB.
- snowflake-sqlalchemy - driver for Snowflake.
- sqlalchemy-tds - driver for MS-SQL, on top of pythone-tds.
- crate - driver for CrateDB.

4.2.2 Experimental / Incomplete

Dialects that are in an incomplete state or are considered somewhat experimental.

- CALCHIPAN - Adapts Pandas dataframes to SQLAlchemy.
- sqlalchemy-cubrid - driver for the CUBRID database.

4.2.3 Attic

Dialects in the "attic" are those that were contributed for SQLAlchemy long ago but have received little attention or demand since then, and are now moved out to their own repositories in at best a semi-working state. Community members interested in these dialects should feel free to pick up on their current codebase and fork off into working libraries.

- sqlalchemy-access - driver for Microsoft Access.
- sqlalchemy-drizzle - driver for the Drizzle MySQL variant.
- sqlalchemy-informixdb - driver for the informixdb DBAPI.
- sqlalchemy-maxdb - driver for the MaxDB database

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS

The Frequently Asked Questions section is a growing collection of commonly observed questions to well-known issues.

5.1 Connections / Engines

5.1.1 How do I configure logging?

See dbengine_logging.

5.1.2 How do I pool database connections? Are my connections pooled?

SQLAlchemy performs application-level connection pooling automatically in most cases. With the exception of SQLite, a `Engine` object refers to a `QueuePool` as a source of connectivity.

For more detail, see engines_toplevel and pooling_toplevel.

5.1.3 How do I pass custom connect arguments to my database API?

The `create_engine()` call accepts additional arguments either directly via the `connect_args` keyword argument:

```
e = create_engine("mysql://scott:tiger@localhost/test",
                  connect_args={"encoding": "utf8"})
```

Or for basic string and integer arguments, they can usually be specified in the query string of the URL:

```
e = create_engine("mysql://scott:tiger@localhost/test?encoding=utf8")
```

See also:

custom_dbapi_args

5.1.4 "MySQL Server has gone away"

The primary cause of this error is that the MySQL connection has timed out and has been closed by the server. The MySQL server closes connections which have been idle a period of time which defaults to eight hours. To accommodate this, the immediate setting is to enable the `create_engine.pool_recycle` setting, which will ensure that a connection which is older than a set amount of seconds will be discarded and replaced with a new connection when it is next checked out.

For the more general case of accommodating database restarts and other temporary loss of connectivity due to network issues, connections that are in the pool may be recycled in response to more generalized disconnect detection techniques. The section pool_disconnects provides background on both "pessimistic" (e.g. pre-ping) and "optimistic" (e.g. graceful recovery) techniques. Modern SQLAlchemy tends to favor the "pessimistic" approach.

See also:

pool_disconnects

5.1.5 "Commands out of sync; you can't run this command now" / "This result object does not return rows. It has been closed automatically"

The MySQL drivers have a fairly wide class of failure modes whereby the state of the connection to the server is in an invalid state. Typically, when the connection is used again, one of these two error messages will occur. The reason is because the state of the server has been changed to one in which the client library does not expect, such that when the client library emits a new statement on the connection, the server does not respond as expected.

In SQLAlchemy, because database connections are pooled, the issue of the messaging being out of sync on a connection becomes more important, since when an operation fails, if the connection itself is in an unusable state, if it goes back into the connection pool, it will malfunction when checked out again. The mitigation for this issue is that the connection is **invalidated** when such a failure mode occurs so that the underlying database connection to MySQL is discarded. This invalidation occurs automatically for many known failure modes and can also be called explicitly via the `Connection.invalidate()` method.

There is also a second class of failure modes within this category where a context manager such as `with session.begin_nested():` wants to "roll back" the transaction when an error occurs; however within some failure modes of the connection, the rollback itself (which can also be a RELEASE SAVEPOINT operation) also fails, causing misleading stack traces.

Originally, the cause of this error used to be fairly simple, it meant that a multithreaded program was invoking commands on a single connection from more than one thread. This applied to the original "MySQLdb" native-C driver that was pretty much the only driver in use. However, with the introduction of pure Python drivers like PyMySQL and MySQL-connector-Python, as well as increased use of tools such as gevent/eventlet, multiprocessing (often with Celery), and others, there is a whole series of factors that has been known to cause this problem, some of which have been improved across SQLAlchemy versions but others which are unavoidable:

- **Sharing a connection among threads** - This is the original reason these kinds of errors occurred. A program used the same connection in two or more threads at the same time, meaning multiple

sets of messages got mixed up on the connection, putting the server-side session into a state that the client no longer knows how to interpret. However, other causes are usually more likely today.

- **Sharing the filehandle for the connection among processes** - This usually occurs when a program uses `os.fork()` to spawn a new process, and a TCP connection that is present in th parent process gets shared into one or more child processes. As multiple processes are now emitting messages to essentially the same filehandle, the server receives interleaved messages and breaks the state of the connection.

  This scenario can occur very easily if a program uses Python's "multiprocessing" module and makes use of an `Engine` that was created in the parent process. It's common that "multiprocessing" is in use when using tools like Celery. The correct approach should be either that a new `Engine` is produced when a child process first starts, discarding any `Engine` that came down from the parent process; or, the `Engine` that's inherited from the parent process can have it's internal pool of connections disposed by calling `Engine.dispose()`.

- **Greenlet Monkeypatching w/ Exits** - When using a library like gevent or eventlet that monkeypatches the Python networking API, libraries like PyMySQL are now working in an asynchronous mode of operation, even though they are not developed explicitly against this model. A common issue is that a greenthread is interrupted, often due to timeout logic in the application. This results in the `GreenletExit` exception being raised, and the pure-Python MySQL driver is interrupted from its work, which may have been that it was receiving a response from the server or preparing to otherwise reset the state of the connection. When the exception cuts all that work short, the conversation between client and server is now out of sync and subsequent usage of the connection may fail. SQLAlchemy as of version 1.1.0 knows how to guard against this, as if a database operation is interrupted by a so-called "exit exception", which includes `GreenletExit` and any other subclass of Python `BaseException` that is not also a subclass of `Exception`, the connection is invalidated.

- **Rollbacks / SAVEPOINT releases failing** - Some classes of error cause the connection to be unusable within the context of a transaction, as well as when operating in a "SAVEPOINT" block. In these cases, the failure on the connection has rendered any SAVEPOINT as no longer existing, yet when SQLAlchemy, or the application, attempts to "roll back" this savepoint, the "RELEASE SAVEPOINT" operation fails, typically with a message like "savepoint does not exist". In this case, under Python 3 there will be a chain of exceptions output, where the ultimate "cause" of the error will be displayed as well. Under Python 2, there are no "chained" exceptions, however recent versions of SQLAlchemy will attempt to emit a warning illustrating the original failure cause, while still throwing the immediate error which is the failure of the ROLLBACK.

5.1.6 Why does SQLAlchemy issue so many ROLLBACKs?

SQLAlchemy currently assumes DBAPI connections are in "non-autocommit" mode - this is the default behavior of the Python database API, meaning it must be assumed that a transaction is always in progress. The connection pool issues `connection.rollback()` when a connection is returned. This is so that any transactional resources remaining on the connection are released. On a database like PostgreSQL or MSSQL where table resources are aggressively locked, this is critical so that rows and tables don't remain locked within connections that are no longer in use. An application can otherwise hang. It's not just for locks, however, and is equally critical on any database that has any kind of transaction isolation, including MySQL with InnoDB. Any connection that is still inside an old transaction will return stale data, if that data was already queried on that connection within isolation. For background on why you might see stale data even on MySQL, see http://dev.mysql.com/doc/refman/5.1/en/innodb-transaction-model.html

I'm on MyISAM - how do I turn it off?

The behavior of the connection pool's connection return behavior can be configured using `reset_on_return`:

```
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool
```

```
engine = create_engine('mysql://scott:tiger@localhost/myisam_database', pool=QueuePool(reset_
↪on_return=False))
```

I'm on SQL Server - how do I turn those ROLLBACKs into COMMITs?

`reset_on_return` accepts the values `commit`, `rollback` in addition to `True`, `False`, and `None`. Setting to `commit` will cause a COMMIT as any connection is returned to the pool:

```
engine = create_engine('mssql://scott:tiger@mydsn', pool=QueuePool(reset_on_return='commit'))
```

5.1.7 I am using multiple connections with a SQLite database (typically to test transaction operation), and my test program is not working!

If using a SQLite `:memory:` database, or a version of SQLAlchemy prior to version 0.7, the default connection pool is the `SingletonThreadPool`, which maintains exactly one SQLite connection per thread. So two connections in use in the same thread will actually be the same SQLite connection. Make sure you're not using a :memory: database and use `NullPool`, which is the default for non-memory databases in current SQLAlchemy versions.

See also:

Threading/Pooling Behavior - info on PySQLite's behavior.

5.1.8 How do I get at the raw DBAPI connection when using an Engine?

With a regular SA engine-level Connection, you can get at a pool-proxied version of the DBAPI connection via the `Connection.connection` attribute on `Connection`, and for the really-real DBAPI connection you can call the `ConnectionFairy.connection` attribute on that - but there should never be any need to access the non-pool-proxied DBAPI connection, as all methods are proxied through:

```
engine = create_engine(...)
conn = engine.connect()
conn.connection.<do DBAPI things>
cursor = conn.connection.cursor(<DBAPI specific arguments..>)
```

You must ensure that you revert any isolation level settings or other operation-specific settings on the connection back to normal before returning it to the pool.

As an alternative to reverting settings, you can call the `Connection.detach()` method on either `Connection` or the proxied connection, which will de-associate the connection from the pool such that it will be closed and discarded when `Connection.close()` is called:

```
conn = engine.connect()
conn.detach()  # detaches the DBAPI connection from the connection pool
conn.connection.<go nuts>
conn.close()  # connection is closed for real, the pool replaces it with a new connection
```

5.1.9 How do I use engines / connections / sessions with Python multiprocessing, or os.fork()?

The key goal with multiple python processes is to prevent any database connections from being shared across processes. Depending on specifics of the driver and OS, the issues that arise here range from non-working connections to socket connections that are used by multiple processes concurrently, leading to broken messaging (the latter case is typically the most common).

The SQLAlchemy `Engine` object refers to a connection pool of existing database connections. So when this object is replicated to a child process, the goal is to ensure that no database connections are carried over. There are three general approaches to this:

1. Disable pooling using `NullPool`. This is the most simplistic, one shot system that prevents the `Engine` from using any connection more than once.
2. Call `Engine.dispose()` on any given `Engine` as soon one is within the new process. In Python multiprocessing, constructs such as `multiprocessing.Pool` include "initializer" hooks which are a place that this can be performed; otherwise at the top of where `os.fork()` or where the `Process` object begins the child fork, a single call to `Engine.dispose()` will ensure any remaining connections are flushed.
3. An event handler can be applied to the connection pool that tests for connections being shared across process boundaries, and invalidates them. This looks like the following:

```
import os
import warnings

from sqlalchemy import event
from sqlalchemy import exc

def add_engine_pidguard(engine):
    """Add multiprocessing guards.

    Forces a connection to be reconnected if it is detected
    as having been shared to a sub-process.

    """

    @event.listens_for(engine, "connect")
    def connect(dbapi_connection, connection_record):
        connection_record.info['pid'] = os.getpid()

    @event.listens_for(engine, "checkout")
    def checkout(dbapi_connection, connection_record, connection_proxy):
        pid = os.getpid()
        if connection_record.info['pid'] != pid:
            # substitute log.debug() or similar here as desired
            warnings.warn(
                "Parent process %(orig)s forked (%(newproc)s) with an open "
                "database connection, "
                "which is being discarded and recreated." %
                {"newproc": pid, "orig": connection_record.info['pid']})
            connection_record.connection = connection_proxy.connection = None
            raise exc.DisconnectionError(
                "Connection record belongs to pid %s, "
                "attempting to check out in pid %s" %
                (connection_record.info['pid'], pid)
            )
```

These events are applied to an `Engine` as soon as its created:

```
engine = create_engine("...")

add_engine_pidguard(engine)
```

The above strategies will accommodate the case of an `Engine` being shared among processes. However, for the case of a transaction-active `Session` or `Connection` being shared, there's no automatic fix for this; an application needs to ensure a new child process only initiate new `Connection` objects and transactions, as well as ORM `Session` objects. For a `Session` object, technically this is only needed if the session is currently transaction-bound, however the scope of a single `Session` is in any case intended to be kept within a single call stack in any case (e.g. not a global object, not shared between processes or threads).

5.2 MetaData / Schema

5.2.1 My program is hanging when I say `table.drop()` / `metadata.drop_all()`

This usually corresponds to two conditions: 1. using PostgreSQL, which is really strict about table locks, and 2. you have a connection still open which contains locks on the table and is distinct from the connection being used for the DROP statement. Heres the most minimal version of the pattern:

```
connection = engine.connect()
result = connection.execute(mytable.select())

mytable.drop(engine)
```

Above, a connection pool connection is still checked out; furthermore, the result object above also maintains a link to this connection. If "implicit execution" is used, the result will hold this connection opened until the result object is closed or all rows are exhausted.

The call to `mytable.drop(engine)` attempts to emit DROP TABLE on a second connection procured from the `Engine` which will lock.

The solution is to close out all connections before emitting DROP TABLE:

```
connection = engine.connect()
result = connection.execute(mytable.select())

# fully read result sets
result.fetchall()

# close connections
connection.close()

# now locks are removed
mytable.drop(engine)
```

5.2.2 Does SQLAlchemy support ALTER TABLE, CREATE VIEW, CREATE TRIGGER, Schema Upgrade Functionality?

General ALTER support isn't present in SQLAlchemy directly. For special DDL on an ad-hoc basis, the `DDL` and related constructs can be used. See core/ddl for a discussion on this subject.

A more comprehensive option is to use schema migration tools, such as Alembic or SQLAlchemy-Migrate; see schema_migrations for discussion on this.

5.2.3 How can I sort Table objects in order of their dependency?

This is available via the `MetaData.sorted_tables` function:

```
metadata = MetaData()
# ... add Table objects to metadata
ti = metadata.sorted_tables:
for t in ti:
    print(t)
```

5.2.4 How can I get the CREATE TABLE/ DROP TABLE output as a string?

Modern SQLAlchemy has clause constructs which represent DDL operations. These can be rendered to strings like any other SQL expression:

```
from sqlalchemy.schema import CreateTable

print(CreateTable(mytable))
```

To get the string specific to a certain engine:

```
print(CreateTable(mytable).compile(engine))
```

There's also a special form of `Engine` that can let you dump an entire metadata creation sequence, using this recipe:

```
def dump(sql, *multiparams, **params):
    print(sql.compile(dialect=engine.dialect))
engine = create_engine('postgresql://', strategy='mock', executor=dump)
metadata.create_all(engine, checkfirst=False)
```

The Alembic tool also supports an "offline" SQL generation mode that renders database migrations as SQL scripts.

5.2.5 How can I subclass Table/Column to provide certain behaviors/configurations?

`Table` and `Column` are not good targets for direct subclassing. However, there are simple ways to get on-construction behaviors using creation functions, and behaviors related to the linkages between schema objects such as constraint conventions or naming conventions using attachment events. An example of many of these techniques can be seen at Naming Conventions.

5.3 SQL Expressions

5.3.1 How do I render SQL expressions as strings, possibly with bound parameters inlined?

The "stringification" of a SQLAlchemy statement or Query in the vast majority of cases is as simple as:

```
print(str(statement))
```

this applies both to an ORM `Query` as well as any `select()` or other statement. Additionally, to get the statement as compiled to a specific dialect or engine, if the statement itself is not already bound to one you can pass this in to `ClauseElement.compile()`:

```
print(statement.compile(someengine))
```

or without an `Engine`:

```
from sqlalchemy.dialects import postgresql
print(statement.compile(dialect=postgresql.dialect()))
```

When given an ORM `Query` object, in order to get at the `ClauseElement.compile()` method we only need access the `statement` accessor first:

```
statement = query.statement
print(statement.compile(someengine))
```

The above forms will render the SQL statement as it is passed to the Python DBAPI, which includes that bound parameters are not rendered inline. SQLAlchemy normally does not stringify bound parameters, as this is handled appropriately by the Python DBAPI, not to mention bypassing bound parameters is probably the most widely exploited security hole in modern web applications. SQLAlchemy has limited ability to do this stringification in certain circumstances such as that of emitting DDL. In order to access this functionality one can use the `literal_binds` flag, passed to `compile_kwargs`:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print(s.compile(compile_kwargs={"literal_binds": True}))
```

the above approach has the caveats that it is only supported for basic types, such as ints and strings, and furthermore if a `bindparam()` without a pre-set value is used directly, it won't be able to stringify that either.

To support inline literal rendering for types not supported, implement a `TypeDecorator` for the target type which includes a `TypeDecorator.process_literal_param()` method:

```
from sqlalchemy import TypeDecorator, Integer


class MyFancyType(TypeDecorator):
    impl = Integer

    def process_literal_param(self, value, dialect):
        return "my_fancy_formatting(%s)" % value

from sqlalchemy import Table, Column, MetaData

tab = Table('mytable', MetaData(), Column('x', MyFancyType()))

print(
    tab.select().where(tab.c.x > 5).compile(
        compile_kwargs={"literal_binds": True})
)
```

producing output like:

```
SELECT mytable.x
FROM mytable
WHERE mytable.x > my_fancy_formatting(5)
```

5.3.2 I'm using op() to generate a custom operator and my parenthesis are not coming out correctly

The `Operators.op()` method allows one to create a custom database operator otherwise not known by SQLAlchemy:

```
>>> print(column('q').op('->')(column('p')))
q -> p
```

However, when using it on the right side of a compound expression, it doesn't generate parenthesis as we expect:

```
>>> print((column('q1') + column('q2')).op('->')(column('p')))
q1 + q2 -> p
```

Where above, we probably want `(q1 + q2) -> p`.

The solution to this case is to set the precedence of the operator, using the `Operators.op.precedence` parameter, to a high number, where 100 is the maximum value, and the highest number used by any SQLAlchemy operator is currently 15:

```
>>> print((column('q1') + column('q2')).op('->', precedence=100)(column('p')))
(q1 + q2) -> p
```

We can also usually force parenthesization around a binary expression (e.g. an expression that has left/right operands and an operator) using the `ColumnElement.self_group()` method:

```
>>> print((column('q1') + column('q2')).self_group().op('->')(column('p')))
(q1 + q2) -> p
```

Why are the parentheses rules like this?

A lot of databases barf when there are excessive parenthesis or when parenthesis are in unusual places they doesn't expect, so SQLAlchemy does not generate parenthesis based on groupings, it uses operator precedence and if the operator is known to be associative, so that parenthesis are generated minimally. Otherwise, an expression like:

```
column('a') & column('b') & column('c') & column('d')
```

would produce:

```
(((a AND b) AND c) AND d)
```

which is fine but would probably annoy people (and be reported as a bug). In other cases, it leads to things that are more likely to confuse databases or at the very least readability, such as:

```
column('q', ARRAY(Integer, dimensions=2))[5][6]
```

would produce:

```
((q[5])[6])
```

There are also some edge cases where we get things like `"(x) = 7"` and databases really don't like that either. So parenthesization doesn't naively parenthesize, it uses operator precedence and associativity to determine groupings.

For `Operators.op()`, the value of precedence defaults to zero.

What if we defaulted the value of `Operators.op.precedence` to 100, e.g. the highest? Then this expression makes more parenthesis, but is otherwise OK, that is, these two are equivalent:

```
>>> print (column('q') - column('y')).op('+', precedence=100)(column('z'))
(q - y) + z
>>> print (column('q') - column('y')).op('+')(column('z'))
q - y + z
```

but these two are not:

```
>>> print column('q') - column('y').op('+', precedence=100)(column('z'))
q - y + z
>>> print column('q') - column('y').op('+')(column('z'))
q - (y + z)
```

For now, it's not clear that as long as we are doing parenthesization based on operator precedence and associativity, if there is really a way to parenthesize automatically for a generic operator with no precedence given that is going to work in all cases, because sometimes you want a custom op to have a lower precedence than the other operators and sometimes you want it to be higher.

It is possible that maybe if the "binary" expression above forced the use of the `self_group()` method when `op()` is called, making the assumption that a compound expression on the left side can always be parenthesized harmlessly. Perhaps this change can be made at some point, however for the time being keeping the parenthesization rules more internally consistent seems to be the safer approach.

5.4 ORM Configuration

- *How do I map a table that has no primary key?*
- *How do I configure a Column that is a Python reserved word or similar?*
- *How do I get a list of all columns, relationships, mapped attributes, etc. given a mapped class?*
- *I'm getting a warning or error about "Implicitly combining column X under attribute Y"*
- *I'm using Declarative and setting primaryjoin/secondaryjoin using an `and_()` or `or_()`, and I am getting an error message about foreign keys.*
- *Why is `ORDER BY` required with `LIMIT` (especially with `subqueryload()`)?*

5.4.1 How do I map a table that has no primary key?

The SQLAlchemy ORM, in order to map to a particular table, needs there to be at least one column denoted as a primary key column; multiple-column, i.e. composite, primary keys are of course entirely feasible as well. These columns do **not** need to be actually known to the database as primary key columns, though it's a good idea that they are. It's only necessary that the columns *behave* as a primary key does, e.g. as a unique and not nullable identifier for a row.

Most ORMs require that objects have some kind of primary key defined because the object in memory must correspond to a uniquely identifiable row in the database table; at the very least, this allows the object can be targeted for UPDATE and DELETE statements which will affect only that object's row and no other. However, the importance of the primary key goes far beyond that. In SQLAlchemy, all ORM-mapped objects are at all times linked uniquely within a `Session` to their specific database row using a pattern called the identity map, a pattern that's central to the unit of work system employed by SQLAlchemy, and is also key to the most common (and not-so-common) patterns of ORM usage.

Note: It's important to note that we're only talking about the SQLAlchemy ORM; an application which builds on Core and deals only with `Table` objects, `select()` constructs and the like, **does not** need any primary key to be present on or associated with a table in any way (though again, in SQL, all

tables should really have some kind of primary key, lest you need to actually update or delete specific rows).

In almost all cases, a table does have a so-called candidate key, which is a column or series of columns that uniquely identify a row. If a table truly doesn't have this, and has actual fully duplicate rows, the table is not corresponding to first normal form and cannot be mapped. Otherwise, whatever columns comprise the best candidate key can be applied directly to the mapper:

```
class SomeClass(Base):
    __table__ = some_table_with_no_pk
    __mapper_args__ = {
        'primary_key':[some_table_with_no_pk.c.uid, some_table_with_no_pk.c.bar]
    }
```

Better yet is when using fully declared table metadata, use the `primary_key=True` flag on those columns:

```
class SomeClass(Base):
    __tablename__ = "some_table_with_no_pk"

    uid = Column(Integer, primary_key=True)
    bar = Column(String, primary_key=True)
```

All tables in a relational database should have primary keys. Even a many-to-many association table - the primary key would be the composite of the two association columns:

```
CREATE TABLE my_association (
  user_id INTEGER REFERENCES user(id),
  account_id INTEGER REFERENCES account(id),
  PRIMARY KEY (user_id, account_id)
)
```

5.4.2 How do I configure a Column that is a Python reserved word or similar?

Column-based attributes can be given any name desired in the mapping. See mapper_column_distinct_names.

5.4.3 How do I get a list of all columns, relationships, mapped attributes, etc. given a mapped class?

This information is all available from the `Mapper` object.

To get at the `Mapper` for a particular mapped class, call the `inspect()` function on it:

```
from sqlalchemy import inspect

mapper = inspect(MyClass)
```

From there, all information about the class can be accessed through properties such as:

- `Mapper.attrs` - a namespace of all mapped attributes. The attributes themselves are instances of `MapperProperty`, which contain additional attributes that can lead to the mapped SQL expression or column, if applicable.
- `Mapper.column_attrs` - the mapped attribute namespace limited to column and SQL expression attributes. You might want to use `Mapper.columns` to get at the `Column` objects directly.
- `Mapper.relationships` - namespace of all `RelationshipProperty` attributes.
- `Mapper.all_orm_descriptors` - namespace of all mapped attributes, plus user-defined attributes defined using systems such as `hybrid_property`, `AssociationProxy` and others.

- `Mapper.columns` - A namespace of `Column` objects and other named SQL expressions associated with the mapping.
- `Mapper.mapped_table` - The `Table` or other selectable to which this mapper is mapped.
- `Mapper.local_table` - The `Table` that is "local" to this mapper; this differs from `Mapper.mapped_table` in the case of a mapper mapped using inheritance to a composed selectable.

5.4.4 I'm getting a warning or error about "Implicitly combining column X under attribute Y"

This condition refers to when a mapping contains two columns that are being mapped under the same attribute name due to their name, but there's no indication that this is intentional. A mapped class needs to have explicit names for every attribute that is to store an independent value; when two columns have the same name and aren't disambiguated, they fall under the same attribute and the effect is that the value from one column is **copied** into the other, based on which column was assigned to the attribute first.

This behavior is often desirable and is allowed without warning in the case where the two columns are linked together via a foreign key relationship within an inheritance mapping. When the warning or exception occurs, the issue can be resolved by either assigning the columns to differently-named attributes, or if combining them together is desired, by using `column_property()` to make this explicit.

Given the example as follows:

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

Base = declarative_base()

class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

class B(A):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    a_id = Column(Integer, ForeignKey('a.id'))
```

As of SQLAlchemy version 0.9.5, the above condition is detected, and will warn that the `id` column of `A` and `B` is being combined under the same-named attribute `id`, which above is a serious issue since it means that a `B` object's primary key will always mirror that of its `A`.

A mapping which resolves this is as follows:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

class B(A):
    __tablename__ = 'b'

    b_id = Column('id', Integer, primary_key=True)
    a_id = Column(Integer, ForeignKey('a.id'))
```

Suppose we did want `A.id` and `B.id` to be mirrors of each other, despite the fact that `B.a_id` is where `A.id` is related. We could combine them together using `column_property()`:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

class B(A):
    __tablename__ = 'b'

    # probably not what you want, but this is a demonstration
    id = column_property(Column(Integer, primary_key=True), A.id)
    a_id = Column(Integer, ForeignKey('a.id'))
```

5.4.5 I'm using Declarative and setting primaryjoin/secondaryjoin using an and_() or or_(), and I am getting an error message about foreign keys.

Are you doing this?:

```
class MyClass(Base):
    # ....

    foo = relationship("Dest", primaryjoin=and_("MyClass.id==Dest.foo_id", "MyClass.foo==Dest.
↪bar"))
```

That's an `and_()` of two string expressions, which SQLAlchemy cannot apply any mapping towards. Declarative allows `relationship()` arguments to be specified as strings, which are converted into expression objects using `eval()`. But this doesn't occur inside of an `and_()` expression - it's a special operation declarative applies only to the *entirety* of what's passed to primaryjoin or other arguments as a string:

```
class MyClass(Base):
    # ....

    foo = relationship("Dest", primaryjoin="and_(MyClass.id==Dest.foo_id, MyClass.foo==Dest.
↪bar)")
```

Or if the objects you need are already available, skip the strings:

```
class MyClass(Base):
    # ....

    foo = relationship(Dest, primaryjoin=and_(MyClass.id==Dest.foo_id, MyClass.foo==Dest.bar))
```

The same idea applies to all the other arguments, such as `foreign_keys`:

```
# wrong !
foo = relationship(Dest, foreign_keys=["Dest.foo_id", "Dest.bar_id"])

# correct !
foo = relationship(Dest, foreign_keys="[Dest.foo_id, Dest.bar_id]")

# also correct !
foo = relationship(Dest, foreign_keys=[Dest.foo_id, Dest.bar_id])

# if you're using columns from the class that you're inside of, just use the column objects !
class MyClass(Base):
    foo_id = Column(...)
    bar_id = Column(...)
    # ...

    foo = relationship(Dest, foreign_keys=[foo_id, bar_id])
```

5.4.6 Why is ORDER BY required with LIMIT (especially with subqueryload())?

A relational database can return rows in any arbitrary order, when an explicit ordering is not set. While this ordering very often corresponds to the natural order of rows within a table, this is not the case for all databases and all queries. The consequence of this is that any query that limits rows using `LIMIT` or `OFFSET` should **always** specify an `ORDER BY`. Otherwise, it is not deterministic which rows will actually be returned.

When we use a SQLAlchemy method like `Query.first()`, we are in fact applying a `LIMIT` of one to the query, so without an explicit ordering it is not deterministic what row we actually get back. While we may not notice this for simple queries on databases that usually returns rows in their natural order, it becomes much more of an issue if we also use `orm.subqueryload()` to load related collections, and we may not be loading the collections as intended.

SQLAlchemy implements `orm.subqueryload()` by issuing a separate query, the results of which are matched up to the results from the first query. We see two queries emitted like this:

```
>>> session.query(User).options(subqueryload(User.addresses)).all()
{opensql}-- the "main" query
SELECT users.id AS users_id
FROM users
{stop}
{opensql}-- the "load" query issued by subqueryload
SELECT addresses.id AS addresses_id,
       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) AS anon_1
JOIN addresses ON anon_1.users_id = addresses.user_id
ORDER BY anon_1.users_id
```

The second query embeds the first query as a source of rows. When the inner query uses `OFFSET` and/or `LIMIT` without ordering, the two queries may not see the same results:

```
>>> user = session.query(User).options(subqueryload(User.addresses)).first()
{opensql}-- the "main" query
SELECT users.id AS users_id
FROM users
 LIMIT 1
{stop}
{opensql}-- the "load" query issued by subqueryload
SELECT addresses.id AS addresses_id,
       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 LIMIT 1) AS anon_1
JOIN addresses ON anon_1.users_id = addresses.user_id
ORDER BY anon_1.users_id
```

Depending on database specifics, there is a chance we may get a result like the following for the two queries:

```
-- query #1
+--------+
|users_id|
+--------+
|       1|
+--------+

-- query #2
+------------+-----------------+---------------+
|addresses_id|addresses_user_id|anon_1_users_id|
+------------+-----------------+---------------+
|           3|                2|              2|
+------------+-----------------+---------------+
```

```
|          4|                2|              2|
+-----------+-----------------+---------------+
```

Above, we receive two `addresses` rows for `user.id` of 2, and none for 1. We've wasted two rows and failed to actually load the collection. This is an insidious error because without looking at the SQL and the results, the ORM will not show that there's any issue; if we access the `addresses` for the `User` we have, it will emit a lazy load for the collection and we won't see that anything actually went wrong.

The solution to this problem is to always specify a deterministic sort order, so that the main query always returns the same set of rows. This generally means that you should `Query.order_by()` on a unique column on the table. The primary key is a good choice for this:

```
session.query(User).options(subqueryload(User.addresses)).order_by(User.id).first()
```

Note that the `joinedload()` eager loader strategy does not suffer from the same problem because only one query is ever issued, so the load query cannot be different from the main query. Similarly, the `selectinload()` eager loader strategy also does not have this issue as it links its collection loads directly to primary key values just loaded.

See also:

subqueryload_ordering

5.5 Performance

5.5.1 How can I profile a SQLAlchemy powered application?

Looking for performance issues typically involves two stratgies. One is query profiling, and the other is code profiling.

Query Profiling

Sometimes just plain SQL logging (enabled via python's logging module or via the `echo=True` argument on `create_engine()`) can give an idea how long things are taking. For example, if you log something right after a SQL operation, you'd see something like this in your log:

```
17:37:48,325 INFO  [sqlalchemy.engine.base.Engine.0x...048c] SELECT ...
17:37:48,326 INFO  [sqlalchemy.engine.base.Engine.0x...048c] {<params>}
17:37:48,660 DEBUG [myapp.somemessage]
```

if you logged `myapp.somemessage` right after the operation, you know it took 334ms to complete the SQL part of things.

Logging SQL will also illustrate if dozens/hundreds of queries are being issued which could be better organized into much fewer queries. When using the SQLAlchemy ORM, the "eager loading" feature is provided to partially (`contains_eager()`) or fully (`joinedload()`, `subqueryload()`) automate this activity, but without the ORM "eager loading" typically means to use joins so that results across multiple tables can be loaded in one result set instead of multiplying numbers of queries as more depth is added (i.e. `r + r*r2 + r*r2*r3` ...)

For more long-term profiling of queries, or to implement an application-side "slow query" monitor, events can be used to intercept cursor executions, using a recipe like the following:

```
from sqlalchemy import event
from sqlalchemy.engine import Engine
import time
import logging

logging.basicConfig()
logger = logging.getLogger("myapp.sqltime")
logger.setLevel(logging.DEBUG)

@event.listens_for(Engine, "before_cursor_execute")
def before_cursor_execute(conn, cursor, statement,
                          parameters, context, executemany):
    conn.info.setdefault('query_start_time', []).append(time.time())
    logger.debug("Start Query: %s", statement)

@event.listens_for(Engine, "after_cursor_execute")
def after_cursor_execute(conn, cursor, statement,
                         parameters, context, executemany):
    total = time.time() - conn.info['query_start_time'].pop(-1)
    logger.debug("Query Complete!")
    logger.debug("Total Time: %f", total)
```

Above, we use the `ConnectionEvents.before_cursor_execute()` and `ConnectionEvents.after_cursor_execute()` events to establish an interception point around when a statement is executed. We attach a timer onto the connection using the `_ConnectionRecord.info` dictionary; we use a stack here for the occasional case where the cursor execute events may be nested.

Code Profiling

If logging reveals that individual queries are taking too long, you'd need a breakdown of how much time was spent within the database processing the query, sending results over the network, being handled by the DBAPI, and finally being received by SQLAlchemy's result set and/or ORM layer. Each of these stages can present their own individual bottlenecks, depending on specifics.

For that you need to use the Python Profiling Module. Below is a simple recipe which works profiling into a context manager:

```
import cProfile
import StringIO
import pstats
import contextlib

@contextlib.contextmanager
def profiled():
    pr = cProfile.Profile()
    pr.enable()
    yield
    pr.disable()
    s = StringIO.StringIO()
    ps = pstats.Stats(pr, stream=s).sort_stats('cumulative')
    ps.print_stats()
    # uncomment this to see who's calling what
```

```
    # ps.print_callers()
    print(s.getvalue())
```

To profile a section of code:

```
with profiled():
    Session.query(FooClass).filter(FooClass.somevalue==8).all()
```

The output of profiling can be used to give an idea where time is being spent. A section of profiling output looks like this:

```
13726 function calls (13042 primitive calls) in 0.014 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
222/21    0.001    0.000    0.011    0.001 lib/sqlalchemy/orm/loading.py:26(instances)
220/20    0.002    0.000    0.010    0.001 lib/sqlalchemy/orm/loading.py:327(_instance)
220/20    0.000    0.000    0.010    0.000 lib/sqlalchemy/orm/loading.py:284(populate_state)
   20    0.000    0.000    0.010    0.000 lib/sqlalchemy/orm/strategies.py:987(load_collection_
↪from_subq)
   20    0.000    0.000    0.009    0.000 lib/sqlalchemy/orm/strategies.py:935(get)
    1    0.000    0.000    0.009    0.009 lib/sqlalchemy/orm/strategies.py:940(_load)
   21    0.000    0.000    0.008    0.000 lib/sqlalchemy/orm/strategies.py:942(<genexpr>)
    2    0.000    0.000    0.004    0.002 lib/sqlalchemy/orm/query.py:2400(__iter__)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/orm/query.py:2414(_execute_and_
↪instances)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/engine/base.py:659(execute)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/sql/elements.py:321(_execute_on_
↪connection)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/engine/base.py:788(_execute_
↪clauseelement)

...
```

Above, we can see that the `instances()` SQLAlchemy function was called 222 times (recursively, and 21 times from the outside), taking a total of .011 seconds for all calls combined.

Execution Slowness

The specifics of these calls can tell us where the time is being spent. If for example, you see time being spent within `cursor.execute()`, e.g. against the DBAPI:

```
2    0.102    0.102    0.204    0.102 {method 'execute' of 'sqlite3.Cursor' objects}
```

this would indicate that the database is taking a long time to start returning results, and it means your query should be optimized, either by adding indexes or restructuring the query and/or underlying schema. For that task, analysis of the query plan is warranted, using a system such as EXPLAIN, SHOW PLAN, etc. as is provided by the database backend.

Result Fetching Slowness - Core

If on the other hand you see many thousands of calls related to fetching rows, or very long calls to `fetchall()`, it may mean your query is returning more rows than expected, or that the fetching of rows itself is slow. The ORM itself typically uses `fetchall()` to fetch rows (or `fetchmany()` if the `Query.yield_per()` option is used).

An inordinately large number of rows would be indicated by a very slow call to `fetchall()` at the DBAPI level:

```
2    0.300    0.600    0.300    0.600 {method 'fetchall' of 'sqlite3.Cursor' objects}
```

An unexpectedly large number of rows, even if the ultimate result doesn't seem to have many rows, can be the result of a cartesian product - when multiple sets of rows are combined together without appropriately joining the tables together. It's often easy to produce this behavior with SQLAlchemy Core or ORM query if the wrong `Column` objects are used in a complex query, pulling in additional FROM clauses that are unexpected.

On the other hand, a fast call to `fetchall()` at the DBAPI level, but then slowness when SQLAlchemy's `ResultProxy` is asked to do a `fetchall()`, may indicate slowness in processing of datatypes, such as unicode conversions and similar:

```
# the DBAPI cursor is fast...
2    0.020    0.040    0.020    0.040 {method 'fetchall' of 'sqlite3.Cursor' objects}

...

# but SQLAlchemy's result proxy is slow, this is type-level processing
2    0.100    0.200    0.100    0.200 lib/sqlalchemy/engine/result.py:778(fetchall)
```

In some cases, a backend might be doing type-level processing that isn't needed. More specifically, seeing calls within the type API that are slow are better indicators - below is what it looks like when we use a type like this:

```
from sqlalchemy import TypeDecorator
import time

class Foo(TypeDecorator):
    impl = String

    def process_result_value(self, value, thing):
        # intentionally add slowness for illustration purposes
        time.sleep(.001)
        return value
```

the profiling output of this intentionally slow operation can be seen like this:

```
200    0.001    0.000    0.237    0.001 lib/sqlalchemy/sql/type_api.py:911(process)
200    0.001    0.000    0.236    0.001 test.py:28(process_result_value)
200    0.235    0.001    0.235    0.001 {time.sleep}
```

that is, we see many expensive calls within the `type_api` system, and the actual time consuming thing is the `time.sleep()` call.

Make sure to check the Dialect documentation for notes on known performance tuning suggestions at this level, especially for databases like Oracle. There may be systems related to ensuring numeric accuracy or string processing that may not be needed in all cases.

There also may be even more low-level points at which row-fetching performance is suffering; for example, if time spent seems to focus on a call like `socket.receive()`, that could indicate that everything is fast except for the actual network connection, and too much time is spent with data moving over the network.

Result Fetching Slowness - ORM

To detect slowness in ORM fetching of rows (which is the most common area of performance concern), calls like `populate_state()` and `_instance()` will illustrate individual ORM object populations:

```
# the ORM calls _instance for each ORM-loaded row it sees, and
# populate_state for each ORM-loaded row that results in the population
# of an object's attributes
```

```
220/20    0.001    0.000    0.010    0.000 lib/sqlalchemy/orm/loading.py:327(_instance)
220/20    0.000    0.000    0.009    0.000 lib/sqlalchemy/orm/loading.py:284(populate_state)
```

The ORM's slowness in turning rows into ORM-mapped objects is a product of the complexity of this operation combined with the overhead of cPython. Common strategies to mitigate this include:

- fetch individual columns instead of full entities, that is:

  ```
  session.query(User.id, User.name)
  ```

  instead of:

  ```
  session.query(User)
  ```

- Use `Bundle` objects to organize column-based results:

  ```
  u_b = Bundle('user', User.id, User.name)
  a_b = Bundle('address', Address.id, Address.email)

  for user, address in session.query(u_b, a_b).join(User.addresses):
      # ...
  ```

- Use result caching - see examples_caching for an in-depth example of this.
- Consider a faster interpreter like that of Pypy.

The output of a profile can be a little daunting but after some practice they are very easy to read.

See also:

examples_performance - a suite of performance demonstrations with bundled profiling capabilities.

5.5.2 I'm inserting 400,000 rows with the ORM and it's really slow!

The SQLAlchemy ORM uses the unit of work pattern when synchronizing changes to the database. This pattern goes far beyond simple "inserts" of data. It includes that attributes which are assigned on objects are received using an attribute instrumentation system which tracks changes on objects as they are made, includes that all rows inserted are tracked in an identity map which has the effect that for each row SQLAlchemy must retrieve its "last inserted id" if not already given, and also involves that rows to be inserted are scanned and sorted for dependencies as needed. Objects are also subject to a fair degree of bookkeeping in order to keep all of this running, which for a very large number of rows at once can create an inordinate amount of time spent with large data structures, hence it's best to chunk these.

Basically, unit of work is a large degree of automation in order to automate the task of persisting a complex object graph into a relational database with no explicit persistence code, and this automation has a price.

ORMs are basically not intended for high-performance bulk inserts - this is the whole reason SQLAlchemy offers the Core in addition to the ORM as a first-class component.

For the use case of fast bulk inserts, the SQL generation and execution system that the ORM builds on top of is part of the Core. Using this system directly, we can produce an INSERT that is competitive with using the raw database API directly.

Note: When using the psycopg2 dialect, consider making use of the *batch execution helpers* feature of psycopg2, now supported directly by the SQLAlchemy psycopg2 dialect.

Alternatively, the SQLAlchemy ORM offers the bulk_operations suite of methods, which provide hooks into subsections of the unit of work process in order to emit Core-level INSERT and UPDATE constructs with a small degree of ORM-based automation.

The example below illustrates time-based tests for several different methods of inserting rows, going from the most automated to the least. With cPython 2.7, runtimes observed:

```
SQLAlchemy ORM: Total time for 100000 records 6.89754080772 secs
SQLAlchemy ORM pk given: Total time for 100000 records 4.09481811523 secs
SQLAlchemy ORM bulk_save_objects(): Total time for 100000 records 1.65821218491 secs
SQLAlchemy ORM bulk_insert_mappings(): Total time for 100000 records 0.466513156891 secs
SQLAlchemy Core: Total time for 100000 records 0.21024107933 secs
sqlite3: Total time for 100000 records 0.137335062027 sec
```

We can reduce the time by a factor of nearly three using recent versions of Pypy:

```
SQLAlchemy ORM: Total time for 100000 records 2.39429616928 secs
SQLAlchemy ORM pk given: Total time for 100000 records 1.51412987709 secs
SQLAlchemy ORM bulk_save_objects(): Total time for 100000 records 0.568987131119 secs
SQLAlchemy ORM bulk_insert_mappings(): Total time for 100000 records 0.320806980133 secs
SQLAlchemy Core: Total time for 100000 records 0.206904888153 secs
sqlite3: Total time for 100000 records 0.165791988373 sec
```

Script:

```
import time
import sqlite3

from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String,  create_engine
from sqlalchemy.orm import scoped_session, sessionmaker

Base = declarative_base()
DBSession = scoped_session(sessionmaker())
engine = None


class Customer(Base):
    __tablename__ = "customer"
    id = Column(Integer, primary_key=True)
    name = Column(String(255))


def init_sqlalchemy(dbname='sqlite:///sqlalchemy.db'):
    global engine
    engine = create_engine(dbname, echo=False)
    DBSession.remove()
    DBSession.configure(bind=engine, autoflush=False, expire_on_commit=False)
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)


def test_sqlalchemy_orm(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    for i in xrange(n):
        customer = Customer()
        customer.name = 'NAME ' + str(i)
        DBSession.add(customer)
        if i % 1000 == 0:
            DBSession.flush()
    DBSession.commit()
    print(
        "SQLAlchemy ORM: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")
```

```
def test_sqlalchemy_orm_pk_given(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    for i in xrange(n):
        customer = Customer(id=i + 1, name="NAME " + str(i))
        DBSession.add(customer)
        if i % 1000 == 0:
            DBSession.flush()
    DBSession.commit()
    print(
        "SQLAlchemy ORM pk given: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def test_sqlalchemy_orm_bulk_save_objects(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    for chunk in range(0, n, 10000):
        DBSession.bulk_save_objects(
            [
                Customer(name="NAME " + str(i))
                for i in xrange(chunk, min(chunk + 10000, n))
            ]
        )
    DBSession.commit()
    print(
        "SQLAlchemy ORM bulk_save_objects(): Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def test_sqlalchemy_orm_bulk_insert(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    for chunk in range(0, n, 10000):
        DBSession.bulk_insert_mappings(
            Customer,
            [
                dict(name="NAME " + str(i))
                for i in xrange(chunk, min(chunk + 10000, n))
            ]
        )
    DBSession.commit()
    print(
        "SQLAlchemy ORM bulk_insert_mappings(): Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def test_sqlalchemy_core(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    engine.execute(
        Customer.__table__.insert(),
        [{"name": 'NAME ' + str(i)} for i in xrange(n)]
    )
    print(
        "SQLAlchemy Core: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def init_sqlite3(dbname):
    conn = sqlite3.connect(dbname)
    c = conn.cursor()
    c.execute("DROP TABLE IF EXISTS customer")
```

```
    c.execute(
        "CREATE TABLE customer (id INTEGER NOT NULL, "
        "name VARCHAR(255), PRIMARY KEY(id))")
    conn.commit()
    return conn


def test_sqlite3(n=100000, dbname='sqlite3.db'):
    conn = init_sqlite3(dbname)
    c = conn.cursor()
    t0 = time.time()
    for i in xrange(n):
        row = ('NAME ' + str(i),)
        c.execute("INSERT INTO customer (name) VALUES (?)", row)
    conn.commit()
    print(
        "sqlite3: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " sec")

if __name__ == '__main__':
    test_sqlalchemy_orm(100000)
    test_sqlalchemy_orm_pk_given(100000)
    test_sqlalchemy_orm_bulk_save_objects(100000)
    test_sqlalchemy_orm_bulk_insert(100000)
    test_sqlalchemy_core(100000)
    test_sqlite3(100000)
```

5.6 Sessions / Queries

- *I'm re-loading data with my Session but it isn't seeing changes that I committed elsewhere*
- *"This Session's transaction has been rolled back due to a previous exception during flush." (or similar)*
  - *But why does flush() insist on issuing a ROLLBACK?*
  - *But why isn't the one automatic call to ROLLBACK enough? Why must I ROLLBACK again?*
- *How do I make a Query that always adds a certain filter to every query?*
- *I've created a mapping against an Outer Join, and while the query returns rows, no objects are returned. Why not?*
- *I'm using `joinedload()` or `lazy=False` to create a JOIN/OUTER JOIN and SQLAlchemy is not constructing the correct query when I try to add a WHERE, ORDER BY, LIMIT, etc. (which relies upon the (OUTER) JOIN)*
- *Query has no `__len__()`, why not?*
- *How Do I use Textual SQL with ORM Queries?*
- *I'm calling `Session.delete(myobject)` and it isn't removed from the parent collection!*
- *why isn't my `__init__()` called when I load objects?*
- *how do I use ON DELETE CASCADE with SA's ORM?*
- *I set the "foo_id" attribute on my instance to "7", but the "foo" attribute is still `None` - shouldn't it have loaded Foo with id #7?*
- *How do I walk all objects that are related to a given object?*

- *Is there a way to automagically have only unique keywords (or other kinds of objects) without doing a query for the keyword and getting a reference to the row containing that keyword?*
- *Why does post_update emit UPDATE in addition to the first UPDATE?*

5.6.1 I'm re-loading data with my Session but it isn't seeing changes that I committed elsewhere

The main issue regarding this behavior is that the session acts as though the transaction is in the *serializable* isolation state, even if it's not (and it usually is not). In practical terms, this means that the session does not alter any data that it's already read within the scope of a transaction.

If the term "isolation level" is unfamiliar, then you first need to read this link:

Isolation Level

In short, serializable isolation level generally means that once you SELECT a series of rows in a transaction, you will get *the identical data* back each time you re-emit that SELECT. If you are in the next-lower isolation level, "repeatable read", you'll see newly added rows (and no longer see deleted rows), but for rows that you've *already* loaded, you won't see any change. Only if you are in a lower isolation level, e.g. "read committed", does it become possible to see a row of data change its value.

For information on controlling the isolation level when using the SQLAlchemy ORM, see session_transaction_isolation.

To simplify things dramatically, the `Session` itself works in terms of a completely isolated transaction, and doesn't overwrite any mapped attributes it's already read unless you tell it to. The use case of trying to re-read data you've already loaded in an ongoing transaction is an *uncommon* use case that in many cases has no effect, so this is considered to be the exception, not the norm; to work within this exception, several methods are provided to allow specific data to be reloaded within the context of an ongoing transaction.

To understand what we mean by "the transaction" when we talk about the `Session`, your `Session` is intended to only work within a transaction. An overview of this is at unitofwork_transaction.

Once we've figured out what our isolation level is, and we think that our isolation level is set at a low enough level so that if we re-SELECT a row, we should see new data in our `Session`, how do we see it?

Three ways, from most common to least:

1. We simply end our transaction and start a new one on next access with our `Session` by calling `Session.commit()` (note that if the `Session` is in the lesser-used "autocommit" mode, there would be a call to `Session.begin()` as well). The vast majority of applications and use cases do not have any issues with not being able to "see" data in other transactions because they stick to this pattern, which is at the core of the best practice of **short lived transactions**. See session_faq_whentocreate for some thoughts on this.
2. We tell our `Session` to re-read rows that it has already read, either when we next query for them using `Session.expire_all()` or `Session.expire()`, or immediately on an object using `Session.refresh`. See session_expire for detail on this.
3. We can run whole queries while setting them to definitely overwrite already-loaded objects as they read rows by using `Query.populate_existing()`.

But remember, **the ORM cannot see changes in rows if our isolation level is repeatable read or higher, unless we start a new transaction**.

5.6.2 "This Session's transaction has been rolled back due to a previous exception during flush." (or similar)

This is an error that occurs when a `Session.flush()` raises an exception, rolls back the transaction, but further commands upon the *Session* are called without an explicit call to `Session.rollback()` or `Session.close()`.

It usually corresponds to an application that catches an exception upon `Session.flush()` or `Session.commit()` and does not properly handle the exception. For example:

```
from sqlalchemy import create_engine, Column, Integer
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base(create_engine('sqlite://'))

class Foo(Base):
    __tablename__ = 'foo'
    id = Column(Integer, primary_key=True)

Base.metadata.create_all()

session = sessionmaker()()

# constraint violation
session.add_all([Foo(id=1), Foo(id=1)])

try:
    session.commit()
except:
    # ignore error
    pass

# continue using session without rolling back
session.commit()
```

The usage of the `Session` should fit within a structure similar to this:

```
try:
    <use session>
    session.commit()
except:
   session.rollback()
   raise
finally:
   session.close()  # optional, depends on use case
```

Many things can cause a failure within the try/except besides flushes. You should always have some kind of "framing" of your session operations so that connection and transaction resources have a definitive boundary, otherwise your application doesn't really have its usage of resources under control. This is not to say that you need to put try/except blocks all throughout your application - on the contrary, this would be a terrible idea. You should architect your application such that there is one (or few) point(s) of "framing" around session operations.

For a detailed discussion on how to organize usage of the `Session`, please see session_faq_whentocreate.

But why does flush() insist on issuing a ROLLBACK?

It would be great if `Session.flush()` could partially complete and then not roll back, however this is beyond its current capabilities since its internal bookkeeping would have to be modified such that it can be halted at any time and be exactly consistent with what's been flushed to the database. While this

is theoretically possible, the usefulness of the enhancement is greatly decreased by the fact that many database operations require a ROLLBACK in any case. Postgres in particular has operations which, once failed, the transaction is not allowed to continue:

```
test=> create table foo(id integer primary key);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "foo_pkey" for table "foo"
CREATE TABLE
test=> begin;
BEGIN
test=> insert into foo values(1);
INSERT 0 1
test=> commit;
COMMIT
test=> begin;
BEGIN
test=> insert into foo values(1);
ERROR:  duplicate key value violates unique constraint "foo_pkey"
test=> insert into foo values(2);
ERROR:  current transaction is aborted, commands ignored until end of transaction block
```

What SQLAlchemy offers that solves both issues is support of SAVEPOINT, via `Session.begin_nested()`. Using `Session.begin_nested()`, you can frame an operation that may potentially fail within a transaction, and then "roll back" to the point before its failure while maintaining the enclosing transaction.

But why isn't the one automatic call to ROLLBACK enough? Why must I ROLLBACK again?

This is again a matter of the `Session` providing a consistent interface and refusing to guess about what context its being used. For example, the `Session` supports "framing" above within multiple levels. Such as, suppose you had a decorator `@with_session()`, which did this:

```
def with_session(fn):
    def go(*args, **kw):
        session.begin(subtransactions=True)
        try:
            ret = fn(*args, **kw)
            session.commit()
            return ret
        except:
            session.rollback()
            raise
    return go
```

The above decorator begins a transaction if one does not exist already, and then commits it, if it were the creator. The "subtransactions" flag means that if `Session.begin()` were already called by an enclosing function, nothing happens except a counter is incremented - this counter is decremented when `Session.commit()` is called and only when it goes back to zero does the actual COMMIT happen. It allows this usage pattern:

```
@with_session
def one():
    # do stuff
    two()


@with_session
def two():
    # etc.

one()
```

```
two()
```

`one()` can call `two()`, or `two()` can be called by itself, and the `@with_session` decorator ensures the appropriate "framing" - the transaction boundaries stay on the outermost call level. As you can see, if `two()` calls `flush()` which throws an exception and then issues a `rollback()`, there will *always* be a second `rollback()` performed by the decorator, and possibly a third corresponding to two levels of decorator. If the `flush()` pushed the `rollback()` all the way out to the top of the stack, and then we said that all remaining `rollback()` calls are moot, there is some silent behavior going on there. A poorly written enclosing method might suppress the exception, and then call `commit()` assuming nothing is wrong, and then you have a silent failure condition. The main reason people get this error in fact is because they didn't write clean "framing" code and they would have had other problems down the road.

If you think the above use case is a little exotic, the same kind of thing comes into play if you want to SAVEPOINT- you might call `begin_nested()` several times, and the `commit()`/`rollback()` calls each resolve the most recent `begin_nested()`. The meaning of `rollback()` or `commit()` is dependent upon which enclosing block it is called, and you might have any sequence of `rollback()`/`commit()` in any order, and its the level of nesting that determines their behavior.

In both of the above cases, if `flush()` broke the nesting of transaction blocks, the behavior is, depending on scenario, anywhere from "magic" to silent failure to blatant interruption of code flow.

`flush()` makes its own "subtransaction", so that a transaction is started up regardless of the external transactional state, and when complete it calls `commit()`, or `rollback()` upon failure - but that `rollback()` corresponds to its own subtransaction - it doesn't want to guess how you'd like to handle the external "framing" of the transaction, which could be nested many levels with any combination of subtransactions and real SAVEPOINTs. The job of starting/ending the "frame" is kept consistently with the code external to the `flush()`, and we made a decision that this was the most consistent approach.

5.6.3 How do I make a Query that always adds a certain filter to every query?

See the recipe at PreFilteredQuery.

5.6.4 I've created a mapping against an Outer Join, and while the query returns rows, no objects are returned. Why not?

Rows returned by an outer join may contain NULL for part of the primary key, as the primary key is the composite of both tables. The `Query` object ignores incoming rows that don't have an acceptable primary key. Based on the setting of the `allow_partial_pks` flag on `mapper()`, a primary key is accepted if the value has at least one non-NULL value, or alternatively if the value has no NULL values. See `allow_partial_pks` at `mapper()`.

5.6.5 I'm using `joinedload()` or `lazy=False` to create a JOIN/OUTER JOIN and SQLAlchemy is not constructing the correct query when I try to add a WHERE, ORDER BY, LIMIT, etc. (which relies upon the (OUTER) JOIN)

The joins generated by joined eager loading are only used to fully load related collections, and are designed to have no impact on the primary results of the query. Since they are anonymously aliased, they cannot be referenced directly.

For detail on this beahvior, see zen_of_eager_loading.

5.6.6 Query has no `__len__()`, why not?

The Python `__len__()` magic method applied to an object allows the `len()` builtin to be used to determine the length of the collection. It's intuitive that a SQL query object would link `__len__()` to

the `Query.count()` method, which emits a *SELECT COUNT*. The reason this is not possible is because evaluating the query as a list would incur two SQL calls instead of one:

```
class Iterates(object):
    def __len__(self):
        print("LEN!")
        return 5

    def __iter__(self):
        print("ITER!")
        return iter([1, 2, 3, 4, 5])

list(Iterates())
```

output:

```
ITER!
LEN!
```

5.6.7 How Do I use Textual SQL with ORM Queries?

See:

- orm_tutorial_literal_sql - Ad-hoc textual blocks with `Query`
- session_sql_expressions - Using `Session` with textual SQL directly.

5.6.8 I'm calling `Session.delete(myobject)` and it isn't removed from the parent collection!

See session_deleting_from_collections for a description of this behavior.

5.6.9 why isn't my `__init__()` called when I load objects?

See mapping_constructors for a description of this behavior.

5.6.10 how do I use ON DELETE CASCADE with SA's ORM?

SQLAlchemy will always issue UPDATE or DELETE statements for dependent rows which are currently loaded in the `Session`. For rows which are not loaded, it will by default issue SELECT statements to load those rows and update/delete those as well; in other words it assumes there is no ON DELETE CASCADE configured. To configure SQLAlchemy to cooperate with ON DELETE CASCADE, see passive_deletes.

5.6.11 I set the "foo_id" attribute on my instance to "7", but the "foo" attribute is still `None` - shouldn't it have loaded Foo with id #7?

The ORM is not constructed in such a way as to support immediate population of relationships driven from foreign key attribute changes - instead, it is designed to work the other way around - foreign key attributes are handled by the ORM behind the scenes, the end user sets up object relationships naturally. Therefore, the recommended way to set `o.foo` is to do just that - set it!:

```
foo = Session.query(Foo).get(7)
o.foo = foo
Session.commit()
```

Manipulation of foreign key attributes is of course entirely legal. However, setting a foreign-key attribute to a new value currently does not trigger an "expire" event of the `relationship()` in which it's involved. This means that for the following sequence:

```
o = Session.query(SomeClass).first()
assert o.foo is None  # accessing an un-set attribute sets it to None
o.foo_id = 7
```

`o.foo` is initialized to `None` when we first accessed it. Setting `o.foo_id = 7` will have the value of "7" as pending, but no flush has occurred - so `o.foo` is still `None`:

```
# attribute is already set to None, has not been
# reconciled with o.foo_id = 7 yet
assert o.foo is None
```

For `o.foo` to load based on the foreign key mutation is usually achieved naturally after the commit, which both flushes the new foreign key value and expires all state:

```
Session.commit()  # expires all attributes

foo_7 = Session.query(Foo).get(7)

assert o.foo is foo_7  # o.foo lazyloads on access
```

A more minimal operation is to expire the attribute individually - this can be performed for any persistent object using `Session.expire()`:

```
o = Session.query(SomeClass).first()
o.foo_id = 7
Session.expire(o, ['foo'])  # object must be persistent for this

foo_7 = Session.query(Foo).get(7)

assert o.foo is foo_7  # o.foo lazyloads on access
```

Note that if the object is not persistent but present in the `Session`, it's known as pending. This means the row for the object has not been INSERTed into the database yet. For such an object, setting `foo_id` does not have meaning until the row is inserted; otherwise there is no row yet:

```
new_obj = SomeClass()
new_obj.foo_id = 7

Session.add(new_obj)

# accessing an un-set attribute sets it to None
assert new_obj.foo is None

Session.flush()  # emits INSERT

# expire this because we already set .foo to None
Session.expire(o, ['foo'])

assert new_obj.foo is foo_7  # now it loads
```

Attribute loading for non-persistent objects

One variant on the "pending" behavior above is if we use the flag `load_on_pending` on `relationship()`. When this flag is set, the lazy loader will emit for `new_obj.foo` before the INSERT proceeds; another variant of this is to use the `Session.enable_relationship_loading()` method, which can "attach" an object to a `Session` in such a way that many-to-one relationships

load as according to foreign key attributes regardless of the object being in any particular state. Both techniques are **not recommended for general use**; they were added to suit specific programming scenarios encountered by users which involve the repurposing of the ORM's usual object states.

The recipe ExpireRelationshipOnFKChange features an example using SQLAlchemy events in order to coordinate the setting of foreign key attributes with many-to-one relationships.

5.6.12 How do I walk all objects that are related to a given object?

An object that has other objects related to it will correspond to the `relationship()` constructs set up between mappers. This code fragment will iterate all the objects, correcting for cycles as well:

```
from sqlalchemy import inspect


def walk(obj):
    deque = [obj]

    seen = set()

    while deque:
        obj = deque.pop(0)
        if obj in seen:
            continue
        else:
            seen.add(obj)
            yield obj
        insp = inspect(obj)
        for relationship in insp.mapper.relationships:
            related = getattr(obj, relationship.key)
            if relationship.uselist:
                deque.extend(related)
            elif related is not None:
                deque.append(related)
```

The function can be demonstrated as follows:

```
Base = declarative_base()


class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    bs = relationship("B", backref="a")


class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))
    c_id = Column(ForeignKey('c.id'))
    c = relationship("C", backref="bs")


class C(Base):
    __tablename__ = 'c'
    id = Column(Integer, primary_key=True)


a1 = A(bs=[B(), B(c=C())])
```

```
for obj in walk(a1):
    print(obj)
```

Output:

```
<__main__.A object at 0x10303b190>
<__main__.B object at 0x103025210>
<__main__.B object at 0x10303b0d0>
<__main__.C object at 0x103025490>
```

5.6.13 Is there a way to automagically have only unique keywords (or other kinds of objects) without doing a query for the keyword and getting a reference to the row containing that keyword?

When people read the many-to-many example in the docs, they get hit with the fact that if you create the same `Keyword` twice, it gets put in the DB twice. Which is somewhat inconvenient.

This UniqueObject recipe was created to address this issue.

5.6.14 Why does post_update emit UPDATE in addition to the first UPDATE?

The post_update feature, documented at post_update, involves that an UPDATE statement is emitted in response to changes to a particular relationship-bound foreign key, in addition to the INSERT/UPDATE/DELETE that would normally be emitted for the target row. While the primary purpose of this UPDATE statement is that it pairs up with an INSERT or DELETE of that row, so that it can post-set or pre-unset a foreign key reference in order to break a cycle with a mutually dependent foreign key, it currently is also bundled as a second UPDATE that emits when the target row itself is subject to an UPDATE. In this case, the UPDATE emitted by post_update is *usually* unnecessary and will often appear wasteful.

However, some research into trying to remove this "UPDATE / UPDATE" behavior reveals that major changes to the unit of work process would need to occur not just throughout the post_update implementation, but also in areas that aren't related to post_update for this to work, in that the order of operations would need to be reversed on the non-post_update side in some cases, which in turn can impact other cases, such as correctly handling an UPDATE of a referenced primary key value (see **:ticket:'1063'** for a proof of concept).

The answer is that "post_update" is used to break a cycle between two mutually dependent foreign keys, and to have this cycle breaking be limited to just INSERT/DELETE of the target table implies that the ordering of UPDATE statements elsewhere would need to be liberalized, leading to breakage in other edge cases.

CHAPTER

SIX

ERROR MESSAGES

This section lists descriptions and background for common error messages and warnings raised or emitted by SQLAlchemy.

SQLAlchemy normally raises errors within the context of a SQLAlchemy-specific exception class. For details on these classes, see core_exceptions_toplevel and orm_exceptions_toplevel.

SQLAlchemy errors can roughly be separated into two categories, the **programming-time error** and the **runtime error**. Programming-time errors are raised as a result of functions or methods being called with incorrect arguments, or from other configuration-oriented methods such as mapper configurations that can't be resolved. The programming-time error is typically immediate and deterministic. The runtime error on the other hand represents a failure that occurs as a program runs in response to some condition that occurs arbitrarily, such as database connections being exhausted or some data-related issue occurring. Runtime errors are more likely to be seen in the logs of a running application as the program encounters these states in response to load and data being encountered.

Since runtime errors are not as easy to reproduce and often occur in response to some arbitrary condition as the program runs, they are more difficult to debug and also affect programs that have already been put into production.

Within this section, the goal is to try to provide background on some of the most common runtime errors as well as programming time errors.

6.1 Connections and Transactions

6.1.1 QueuePool limit of size <x> overflow <y> reached, connection timed out, timeout <z>

This is possibly the most common runtime error experienced, as it directly involves the work load of the application surpassing a configured limit, one which typically applies to nearly all SQLAlchemy applications.

The following points summarize what this error means, beginning with the most fundamental points that most SQLAlchemy users should already be familiar with.

- **The SQLAlchemy Engine object uses a pool of connections by default** - What this means is that when one makes use of a SQL database connection resource of an `Engine` object, and then releases that resource, the database connection itself remains connected to the database and is returned to an internal queue where it can be used again. Even though the code may appear to be ending its conversation with the database, in many cases the application will still maintain a fixed number of database connections that persist until the application ends or the pool is explicitly disposed.
- Because of the pool, when an application makes use of a SQL database connection, most typically from either making use of `Engine.connect()` or when making queries using an ORM `Session`, this activity does not necessarily establish a new connection to the database at the moment the connection object is acquired; it instead consults the connection pool for a connection, which will often retrieve an existing connection from the pool to be re-used. If no connections are available,

the pool will create a new database connection, but only if the pool has not surpassed a configured capacity.

- The default pool used in most cases is called `QueuePool`. When you ask this pool to give you a connection and none are available, it will create a new connection **if the total number of connections in play are less than a configured value**. This value is equal to the **pool size plus the max overflow**. That means if you have configured your engine as:

```
engine = create_engine("mysql://u:p@host/db", pool_size=10, max_overflow=20)
```

  The above `Engine` will allow **at most 30 connections** to be in play at any time, not including connections that were detached from the engine or invalidated. If a request for a new connection arrives and 30 connections are already in use by other parts of the application, the connection pool will block for a fixed period of time, before timing out and raising this error message.

  In order to allow for a higher number of connections be in use at once, the pool can be adjusted using the `create_engine.pool_size` and `create_engine.max_overflow` parameters as passed to the `create_engine()` function. The timeout to wait for a connection to be available is configured using the `create_engine.pool_timeout` parameter.

- The pool can be configured to have unlimited overflow by setting `create_engine.max_overflow` to the value "-1". With this setting, the pool will still maintain a fixed pool of connections, however it will never block upon a new connection being requested; it will instead unconditionally make a new connection if none are available.

  However, when running in this way, if the application has an issue where it is using up all available connectivity resources, it will eventually hit the configured limit of available connections on the database itself, which will again return an error. More seriously, when the application exhausts the database of connections, it usually will have caused a great amount of resources to be used up before failing, and can also interfere with other applications and database status mechanisms that rely upon being able to connect to the database.

  Given the above, the connection pool can be looked at as a **safety valve for connection use**, providing a critical layer of protection against a rogue application causing the entire database to become unavailable to all other applications. When receiving this error message, it is vastly preferable to repair the issue using up too many connections and/or configure the limits appropriately, rather than allowing for unlimited overflow which does not actually solve the underlying issue.

What causes an application to use up all the connections that it has available?

- **The application is fielding too many concurrent requests to do work based on the configured value for the pool** - This is the most straightforward cause. If you have an application that runs in a thread pool that allows for 30 concurrent threads, with one connection in use per thread, if your pool is not configured to allow at least 30 connections checked out at once, you will get this error once your application receives enough concurrent requests. Solution is to raise the limits on the pool or lower the number of concurrent threads.

- **The application is not returning connections to the pool** - This is the next most common reason, which is that the application is making use of the connection pool, but the program is failing to release these connections and is instead leaving them open. The connection pool as well as the ORM `Session` do have logic such that when the session and/or connection object is garbage collected, it results in the underlying connection resources being released, however this behavior cannot be relied upon to release resources in a timely manner.

  A common reason this can occur is that the application uses ORM sessions and does not call `Session.close()` upon them one the work involving that session is complete. Solution is to make sure ORM sessions if using the ORM, or engine-bound `Connection` objects if using Core, are explicitly closed at the end of the work being done, either via the appropriate `.close()` method, or by using one of the available context managers (e.g. "with:" statement) to properly release the resource.

- **The application is attempting to run long-running transactions** - A database transaction is a very expensive resource, and should **never be left idle waiting for some event to occur**.

If an application is waiting for a user to push a button, or a result to come off of a long running job queue, or is holding a persistent connection open to a browser, **don't keep a database transaction open for the whole time**. As the application needs to work with the database and interact with an event, open a short-lived transaction at that point and then close it.

- **The application is deadlocking** - Also a common cause of this error and more difficult to grasp, if an application is not able to complete its use of a connection either due to an application-side or database-side deadlock, the application can use up all the available connections which then leads to additional requests receiving this error. Reasons for deadlocks include:
  - Using an implicit async system such as gevent or eventlet without properly monkeypatching all socket libraries and drivers, or which has bugs in not fully covering for all monkeypatched driver methods, or less commonly when the async system is being used against CPU-bound workloads and greenlets making use of database resources are simply waiting too long to attend to them. Neither implicit nor explicit async programming frameworks are typically necessary or appropriate for the vast majority of relational database operations; if an application must use an async system for some area of functionality, it's best that database-oriented business methods run within traditional threads that pass messages to the async part of the application.
  - A database side deadlock, e.g. rows are mutually deadlocked
  - Threading errors, such as mutexes in a mutual deadlock, or calling upon an already locked mutex in the same thread

Keep in mind an alternative to using pooling is to turn off pooling entirely. See the section pool_switching for background on this. However, note that when this error message is occurring, it is **always** due to a bigger problem in the application itself; the pool just helps to reveal the problem sooner.

See also:

pooling_toplevel

connections_toplevel

6.2 DBAPI Errors

The Python database API, or DBAPI, is a specification for database drivers which can be located at Pep-249. This API specifies a set of exception classes that accommodate the full range of failure modes of the database.

SQLAlchemy does not generate these exceptions directly. Instead, they are intercepted from the database driver and wrapped by the SQLAlchemy-provided exception `DBAPIError`, however the messaging within the exception is **generated by the driver, not SQLAlchemy**.

6.2.1 InterfaceError

Exception raised for errors that are related to the database interface rather than the database itself.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

The `InterfaceError` is sometimes raised by drivers in the context of the database connection being dropped, or not being able to connect to the database. For tips on how to deal with this, see the section pool_disconnects.

6.2.2 DatabaseError

Exception raised for errors that are related to the database itself, and not the interface or data being passed.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

6.2.3 DataError

Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

6.2.4 OperationalError

Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

The `OperationalError` is the most common (but not the only) error class used by drivers in the context of the database connection being dropped, or not being able to connect to the database. For tips on how to deal with this, see the section pool_disconnects.

6.2.5 IntegrityError

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

6.2.6 InternalError

Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

The `InternalError` is sometimes raised by drivers in the context of the database connection being dropped, or not being able to connect to the database. For tips on how to deal with this, see the section pool_disconnects.

6.2.7 ProgrammingError

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

The `ProgrammingError` is sometimes raised by drivers in the context of the database connection being dropped, or not being able to connect to the database. For tips on how to deal with this, see the section pool_disconnects.

6.2.8 NotSupportedError

Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a .rollback() on a connection that does not support transaction or has transactions turned off.

This error is a DBAPI Error and originates from the database driver (DBAPI), not SQLAlchemy itself.

6.3 SQL Expression Language

6.3.1 This Compiled object is not bound to any Engine or Connection

This error refers to the concept of "bound metadata", described at dbengine_implicit. The issue occurs when one invokes the `Executable.execute()` method directly off of a Core expression object that is not associated with any `Engine`:

```
metadata = MetaData()
table = Table('t', metadata, Column('q', Integer))

stmt = select([table])
result = stmt.execute()   # <--- raises
```

What the logic is expecting is that the `MetaData` object has been **bound** to a `Engine`:

```
engine = create_engine("mysql+pymysql://user:pass@host/db")
metadata = MetaData(bind=engine)
```

Where above, any statement that derives from a `Table` which in turn derives from that `MetaData` will implicitly make use of the given `Engine` in order to invoke the statement.

Note that the concept of bound metadata is a **legacy pattern** and in most cases is **highly discouraged**. The best way to invoke the statement is to pass it to the `Connection.execute()` method of a `Connection`:

```
with engine.connect() as conn:
  result = conn.execute(stmt)
```

When using the ORM, a similar facility is available via the `Session`:

```
result = session.exxecute(stmt)
```

See also:

dbengine_implicit

6.3.2 A value is required for bind parameter <x> (in parameter group <y>)

This error occurs when a statement makes use of `bindparam()` either implicitly or explicitly and does not provide a value when the statement is executed:

```
stmt = select([table.c.column]).where(table.c.id == bindparam('my_param'))

result = conn.execute(stmt)
```

Above, no value has been provided for the parameter "my_param". The correct approach is to provide a value:

```
result = conn.execute(stmt, my_param=12)
```

When the message takes the form "a value is required for bind parameter <x> in parameter group <y>", the message is referring to the "executemany" stye of execution. In this case, the statement is typically an INSERT, UPDATE, or DELETE and a list of parameters is being passed. In this format, the statement may be generated dynamically to include parameter positions for every parameter given in the argument list, where it will use the **first set of parameters** to determine what these should be.

For example, the statement below is calculated based on the first parameter set to require the parameters, "a", "b", and "c" - these names determine the final string format of the statement which will be used for each set of parameters in the list. As the second entry does not contain "b", this error is generated:

```
m = MetaData()
t = Table(
    't', m,
    Column('a', Integer),
    Column('b', Integer),
    Column('c', Integer)
)

e.execute(
    t.insert(), [
        {"a": 1, "b": 2, "c": 3},
        {"a": 2, "c": 4},
        {"a": 3, "b": 4, "c": 5},
    ]
)

sqlalchemy.exc.StatementError: (sqlalchemy.exc.InvalidRequestError)
A value is required for bind parameter 'b', in parameter group 1
[SQL: u'INSERT INTO t (a, b, c) VALUES (?, ?, ?)']
[parameters: [{'a': 1, 'c': 3, 'b': 2}, {'a': 2, 'c': 4}, {'a': 3, 'c': 5, 'b': 4}]]
```

Since "b" is required, pass it as `None` so that the INSERT may proceed:

```
e.execute(
    t.insert(), [
        {"a": 1, "b": 2, "c": 3},
        {"a": 2, "b": None, "c": 4},
        {"a": 3, "b": 4, "c": 5},
    ]
)
```

See also:

coretutorial_bind_param

execute_multiple

6.4 Object Relational Mapping

6.4.1 Parent instance <x> is not bound to a Session; (lazy load/deferred load/refresh/etc.) operation cannot proceed

This is likely the most common error message when dealing with the ORM, and it occurs as a result of the nature of a technique the ORM makes wide use of known as lazy loading. Lazy loading is a common object-relational pattern whereby an object that's persisted by the ORM maintains a proxy to the database itself, such that when various attributes upon the object are accessed, their value may be retrieved from the database *lazily*. The advantage to this approach is that objects can be retrieved from the database without having to load all of their attributes or related data at once, and instead only that data which is requested can be delivered at that time. The major disadvantage is basically a mirror image of the advantage, which is that if lots of objects are being loaded which are known to require a certain set of data in all cases, it is wasteful to load that additional data piecemeal.

Another caveat of lazy loading beyond the usual efficiency concerns is that in order for lazy loading to proceed, the object has to **remain associated with a Session** in order to be able to retrieve its state. This error message means that an object has become de-associated with its `Session` and is being asked to lazy load data from the database.

The most common reason that objects become detached from their `Session` is that the session itself was closed, typically via the `Session.close()` method. The objects will then live on to be accessed further,

very often within web applications where they are delivered to a server-side templating engine and are asked for further attributes which they cannot load.

Mitigation of this error is via two general techniques:

- **Don't close the session prematurely** - Often, applications will close out a transaction before passing off related objects to some other system which then fails due to this error. Sometimes the transaction doesn't need to be closed so soon; an example is the web application closes out the transaction before the view is rendered. This is often done in the name of "correctness", but may be seen as a mis-application of "encapsulation", as this term refers to code organization, not actual actions. The template that uses an ORM object is making use of the proxy pattern which keeps database logic encapsulated from the caller. If the `Session` can be held open until the lifespan of the objects are done, this is the best approach.
- **Load everything that's needed up front** - It is very often impossible to keep the transaction open, especially in more complex applications that need to pass objects off to other systems that can't run in the same context even though they're in the same process. In this case, the application should try to make appropriate use of eager loading to ensure that objects have what they need up front. As an additional measure, special directives like the `raiseload()` option can ensure that systems don't call upon lazy loading when its not expected.

  See also:

  loading_toplevel - detailed documentation on eager loading and other relationship-oriented loading techniques

6.5 Core Exception Classes

See core_exceptions_toplevel for Core exception classes.

6.6 ORM Exception Classes

See orm_exceptions_toplevel for ORM exception classes.

CHAPTER

SEVEN

GLOSSARY

ACID

ACID model An acronym for "Atomicity, Consistency, Isolation, Durability"; a set of properties that guarantee that database transactions are processed reliably. (via Wikipedia)

See also:

atomicity

consistency

isolation

durability

http://en.wikipedia.org/wiki/ACID_Model

annotations Annotations are a concept used internally by SQLAlchemy in order to store additional information along with `ClauseElement` objects. A Python dictionary is associated with a copy of the object, which contains key/value pairs significant to various internal systems, mostly within the ORM:

```
some_column = Column('some_column', Integer)
some_column_annotated = some_column._annotate({"entity": User})
```

The annotation system differs from the public dictionary `Column.info` in that the above annotation operation creates a *copy* of the new `Column`, rather than considering all annotation values to be part of a single unit. The ORM creates copies of expression objects in order to apply annotations that are specific to their context, such as to differentiate columns that should render themselves as relative to a joined-inheritance entity versus those which should render relative to their immediate parent table alone, as well as to differentiate columns within the "join condition" of a relationship where the column in some cases needs to be expressed in terms of one particular table alias or another, based on its position within the join expression.

association relationship A two-tiered relationship which links two tables together using an association table in the middle. The association relationship differs from a many to many relationship in that the many-to-many table is mapped by a full class, rather than invisibly handled by the `sqlalchemy.orm.relationship()` construct as in the case with many-to-many, so that additional attributes are explicitly available.

For example, if we wanted to associate employees with projects, also storing the specific role for that employee with the project, the relational schema might look like:

```
CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE project (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
```

```
)

CREATE TABLE employee_project (
    employee_id INTEGER PRIMARY KEY,
    project_id INTEGER PRIMARY KEY,
    role_name VARCHAR(30),
    FOREIGN KEY employee_id REFERENCES employee(id),
    FOREIGN KEY project_id REFERENCES project(id)
)
```

A SQLAlchemy declarative mapping for the above might look like:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key)
    name = Column(String(30))


class Project(Base):
    __tablename__ = 'project'

    id = Column(Integer, primary_key)
    name = Column(String(30))


class EmployeeProject(Base):
    __tablename__ = 'employee_project'

    employee_id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    project_id = Column(Integer, ForeignKey('project.id'), primary_key=True)
    role_name = Column(String(30))

    project = relationship("Project", backref="project_employees")
    employee = relationship("Employee", backref="employee_projects")
```

Employees can be added to a project given a role name:

```
proj = Project(name="Client A")

emp1 = Employee(name="emp1")
emp2 = Employee(name="emp2")

proj.project_employees.extend([
    EmployeeProject(employee=emp1, role="tech lead"),
    EmployeeProject(employee=emp2, role="account executive")
])
```

See also:

many to many

atomicity Atomicity is one of the components of the ACID model, and requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. (via Wikipedia)

See also:

ACID

http://en.wikipedia.org/wiki/Atomicity_(database_systems)

backref

bidirectional relationship An extension to the relationship system whereby two distinct `relationship()` objects can be mutually associated with each other, such that they coordinate in memory as changes occur to either side. The most common way these two relationships are constructed is by using the `relationship()` function explicitly for one side and specifying the `backref` keyword to it so that the other `relationship()` is created automatically. We can illustrate this against the example we've used in one to many as follows:

```
class Department(Base):
    __tablename__ = 'department'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    employees = relationship("Employee", backref="department")

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    dep_id = Column(Integer, ForeignKey('department.id'))
```

A backref can be applied to any relationship, including one to many, many to one, and many to many.

See also:

relationship

one to many

many to one

many to many

candidate key A relational algebra term referring to an attribute or set of attributes that form a uniquely identifying key for a row. A row may have more than one candidate key, each of which is suitable for use as the primary key of that row. The primary key of a table is always a candidate key.

See also:

primary key

http://en.wikipedia.org/wiki/Candidate_key

check constraint A check constraint is a condition that defines valid data when adding or updating an entry in a table of a relational database. A check constraint is applied to each row in the table.

(via Wikipedia)

A check constraint can be added to a table in standard SQL using DDL like the following:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

See also:

http://en.wikipedia.org/wiki/Check_constraint

columns clause The portion of the `SELECT` statement which enumerates the SQL expressions to be returned in the result set. The expressions follow the `SELECT` keyword directly and are a comma-separated list of individual expressions.

E.g.:

```
SELECT user_account.name, user_account.email
FROM user_account WHERE user_account.name = 'fred'
```

Above, the list of columns `user_acount.name`, `user_account.email` is the columns clause of the `SELECT`.

consistency Consistency is one of the components of the ACID model, and ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints, cascades, triggers, and any combination thereof. (via Wikipedia)

See also:

ACID

http://en.wikipedia.org/wiki/Consistency_(database_systems)

constraint

constraints

constrained Rules established within a relational database that ensure the validity and consistency of data. Common forms of constraint include primary key constraint, foreign key constraint, and check constraint.

correlates

correlated subquery

correlated subqueries A subquery is correlated if it depends on data in the enclosing `SELECT`.

Below, a subquery selects the aggregate value `MIN(a.id)` from the `email_address` table, such that it will be invoked for each value of `user_account.id`, correlating the value of this column against the `email_address.user_account_id` column:

```
SELECT user_account.name, email_address.email
 FROM user_account
 JOIN email_address ON user_account.id=email_address.user_account_id
 WHERE email_address.id = (
    SELECT MIN(a.id) FROM email_address AS a
    WHERE a.user_account_id=user_account.id
 )
```

The above subquery refers to the `user_account` table, which is not itself in the `FROM` clause of this nested query. Instead, the `user_account` table is received from the enclosing query, where each row selected from `user_account` results in a distinct execution of the subquery.

A correlated subquery is in most cases present in the WHERE clause or columns clause of the immediately enclosing `SELECT` statement, as well as in the ORDER BY or HAVING clause.

In less common cases, a correlated subquery may be present in the FROM clause of an enclosing `SELECT`; in these cases the correlation is typically due to the enclosing `SELECT` itself being enclosed in the WHERE, ORDER BY, columns or HAVING clause of another `SELECT`, such as:

```
SELECT parent.id FROM parent
WHERE EXISTS (
    SELECT * FROM (
        SELECT child.id AS id, child.parent_id AS parent_id, child.pos AS pos
        FROM child
        WHERE child.parent_id = parent.id ORDER BY child.pos
    LIMIT 3)
WHERE id = 7)
```

Correlation from one `SELECT` directly to one which encloses the correlated query via its `FROM` clause is not possible, because the correlation can only proceed once the original source rows from the enclosing statement's FROM clause are available.

crud An acronym meaning "Create, Update, Delete". The term in SQL refers to the set of operations that create, modify and delete data from the database, also known as DML, and typically refers to the `INSERT`, `UPDATE`, and `DELETE` statements.

DBAPI DBAPI is shorthand for the phrase "Python Database API Specification". This is a widely used specification within Python to define common usage patterns for all database connection packages.

The DBAPI is a "low level" API which is typically the lowest level system used in a Python application to talk to a database. SQLAlchemy's dialect system is constructed around the operation of the DBAPI, providing individual dialect classes which service a specific DBAPI on top of a specific database engine; for example, the `create_engine()` URL `postgresql+psycopg2://@localhost/test` refers to the *psycopg2* DBAPI/dialect combination, whereas the URL `mysql+mysqldb://@localhost/test` refers to the *MySQL for Python* DBAPI DBAPI/dialect combination.

See also:

PEP 249 - Python Database API Specification v2.0

DDL An acronym for *Data Definition Language*. DDL is the subset of SQL that relational databases use to configure tables, constraints, and other permanent objects within a database schema. SQLAlchemy provides a rich API for constructing and emitting DDL expressions.

See also:

metadata_toplevel

DDL (via Wikipedia)

deleted This describes one of the major object states which an object can have within a session; a deleted object is an object that was formerly persistent and has had a DELETE statement emitted to the database within a flush to delete its row. The object will move to the detached state once the session's transaction is committed; alternatively, if the session's transaction is rolled back, the DELETE is reverted and the object moves back to the persistent state.

See also:

session_object_states

descriptor

descriptors In Python, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are ___get___(), ___set___(), and ___delete___(). If any of those methods are defined for an object, it is said to be a descriptor.

In SQLAlchemy, descriptors are used heavily in order to provide attribute behavior on mapped classes. When a class is mapped as such:

```
class MyClass(Base):
    __tablename__ = 'foo'

    id = Column(Integer, primary_key=True)
    data = Column(String)
```

The `MyClass` class will be mapped when its definition is complete, at which point the `id` and `data` attributes, starting out as `Column` objects, will be replaced by the instrumentation system with instances of `InstrumentedAttribute`, which are descriptors that provide the above mentioned `__get__()`, `__set__()` and `__delete__()` methods. The `InstrumentedAttribute` will generate a SQL expression when used at the class level:

```
>>> print(MyClass.data == 5)
data = :data_1
```

and at the instance level, keeps track of changes to values, and also lazy loads unloaded attributes from the database:

```
>>> m1 = MyClass()
>>> m1.id = 5
>>> m1.data = "some data"

>>> from sqlalchemy import inspect
>>> inspect(m1).attrs.data.history.added
"some data"
```

detached This describes one of the major object states which an object can have within a session; a detached object is an object that has a database identity (i.e. a primary key) but is not associated with any session. An object that was previously persistent and was removed from its session either because it was expunged, or the owning session was closed, moves into the detached state. The detached state is generally used when objects are being moved between sessions or when being moved to/from an external object cache.

See also:

session_object_states

discriminator A result-set column which is used during polymorphic loading to determine what kind of mapped class should be applied to a particular incoming result row. In SQLAlchemy, the classes are always part of a hierarchy mapping using inheritance mapping.

See also:

inheritance_toplevel

domain model A domain model in problem solving and software engineering is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain.

(via Wikipedia)

See also:

Domain Model (wikipedia)

durability Durability is a property of the ACID model which means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). (via Wikipedia)

See also:

ACID

http://en.wikipedia.org/wiki/Durability_(database_systems)

expire

expires

expiring In the SQLAlchemy ORM, refers to when the data in a persistent or sometimes detached object is erased, such that when the object's attributes are next accessed, a lazy load SQL query will be emitted in order to refresh the data for this object as stored in the current ongoing transaction.

See also:

session_expire

foreign key constraint A referential constraint between two tables. A foreign key is a field or set of fields in a relational table that matches a candidate key of another table. The foreign key can be used to cross-reference tables. (via Wikipedia)

A foreign key constraint can be added to a table in standard SQL using DDL like the following:

```
ALTER TABLE employee ADD CONSTRAINT dep_id_fk
FOREIGN KEY (employee) REFERENCES department (dep_id)
```

See also:

http://en.wikipedia.org/wiki/Foreign_key_constraint

FROM clause The portion of the `SELECT` statement which indicates the initial source of rows.

A simple SELECT will feature one or more table names in its FROM clause. Multiple sources are separated by a comma:

```
SELECT user.name, address.email_address
FROM user, address
WHERE user.id=address.user_id
```

The FROM clause is also where explicit joins are specified. We can rewrite the above SELECT using a single FROM element which consists of a JOIN of the two tables:

```
SELECT user.name, address.email_address
FROM user JOIN address ON user.id=address.user_id
```

generative A term that SQLAlchemy uses to refer what's normally known as method chaining; see that term for details.

identity map A mapping between Python objects and their database identities. The identity map is a collection that's associated with an ORM session object, and maintains a single instance of every database object keyed to its identity. The advantage to this pattern is that all operations which occur for a particular database identity are transparently coordinated onto a single object instance. When using an identity map in conjunction with an isolated transaction, having a reference to an object that's known to have a particular primary key can be considered from a practical standpoint to be a proxy to the actual database row.

See also:

Martin Fowler - Identity Map - http://martinfowler.com/eaaCatalog/identityMap.html

instrumentation

instrumented

instrumenting Instrumentation refers to the process of augmenting the functionality and attribute set of a particular class. Ideally, the behavior of the class should remain close to a regular class, except that additional behaviors and features are made available. The SQLAlchemy mapping process, among other things, adds database-enabled descriptors to a mapped class which each represent a particular database column or relationship to a related class.

isolation

isolated The isolation property of the ACID model ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other. Each transaction must execute in total isolation i.e. if T1 and T2 execute concurrently then each should remain independent of the other. (via Wikipedia)

See also:

ACID

http://en.wikipedia.org/wiki/Isolation_(database_systems)

lazy load

lazy loads

lazy loaded

lazy loading In object relational mapping, a "lazy load" refers to an attribute that does not contain its database-side value for some period of time, typically when the object is first loaded. Instead, the attribute receives a *memoization* that causes it to go out to the database and load its data when it's first used. Using this pattern, the complexity and time spent within object fetches can sometimes be reduced, in that attributes for related tables don't need to be addressed immediately.

See also:

Lazy Load (on Martin Fowler)

N plus one problem

Relationship Loading Techniques

many to many A style of `sqlalchemy.orm.relationship()` which links two tables together via an intermediary table in the middle. Using this configuration, any number of rows on the left side may refer to any number of rows on the right, and vice versa.

A schema where employees can be associated with projects:

```
CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE project (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE employee_project (
    employee_id INTEGER PRIMARY KEY,
    project_id INTEGER PRIMARY KEY,
    FOREIGN KEY employee_id REFERENCES employee(id),
    FOREIGN KEY project_id REFERENCES project(id)
)
```

Above, the `employee_project` table is the many-to-many table, which naturally forms a composite primary key consisting of the primary key from each related table.

In SQLAlchemy, the `sqlalchemy.orm.relationship()` function can represent this style of relationship in a mostly transparent fashion, where the many-to-many table is specified using plain table metadata:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key)
    name = Column(String(30))

    projects = relationship(
        "Project",
        secondary=Table('employee_project', Base.metadata,
                    Column("employee_id", Integer, ForeignKey('employee.id'),
                                primary_key=True),
                    Column("project_id", Integer, ForeignKey('project.id'),
                                primary_key=True)
                ),
        backref="employees"
        )

class Project(Base):
    __tablename__ = 'project'

    id = Column(Integer, primary_key)
    name = Column(String(30))
```

Above, the `Employee.projects` and back-referencing `Project.employees` collections are defined:

```
proj = Project(name="Client A")

emp1 = Employee(name="emp1")
emp2 = Employee(name="emp2")

proj.employees.extend([emp1, emp2])
```

See also:

association relationship

relationship

one to many

many to one

many to one A style of `relationship()` which links a foreign key in the parent mapper's table to the primary key of a related table. Each parent object can then refer to exactly zero or one related object.

The related objects in turn will have an implicit or explicit one to many relationship to any number of parent objects that refer to them.

An example many to one schema (which, note, is identical to the one to many schema):

```
CREATE TABLE department (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30),
    dep_id INTEGER REFERENCES department(id)
)
```

The relationship from `employee` to `department` is many to one, since many employee records can be associated with a single department. A SQLAlchemy mapping might look like:

```
class Department(Base):
    __tablename__ = 'department'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    dep_id = Column(Integer, ForeignKey('department.id'))
    department = relationship("Department")
```

See also:

relationship

one to many

backref

mapping

mapped We say a class is "mapped" when it has been passed through the `orm.mapper()` function. This process associates the class with a database table or other selectable construct, so that instances of it can be persisted using a `Session` as well as loaded using a `Query`.

method chaining An object-oriented technique whereby the state of an object is constructed by calling methods on the object. The object features any number of methods, each of which return a new object (or in some cases the same object) with additional state added to the object.

The two SQLAlchemy objects that make the most use of method chaining are the `Select` object and the `Query` object. For example, a `Select` object can be assigned two expressions to its WHERE clause as well as an ORDER BY clause by calling upon the `where()` and `order_by()` methods:

```
stmt = select([user.c.name]).\
            where(user.c.id > 5).\
            where(user.c.name.like('e%').\
            order_by(user.c.name)
```

Each method call above returns a copy of the original `Select` object with additional qualifiers added.

See also:

generative

N plus one problem The N plus one problem is a common side effect of the lazy load pattern, whereby an application wishes to iterate through a related attribute or collection on each member of a result set of objects, where that attribute or collection is set to be loaded via the lazy load pattern. The net result is that a SELECT statement is emitted to load the initial result set of parent objects; then, as the application iterates through each member, an additional SELECT statement is emitted for each member in order to load the related attribute or collection for that member. The end result is that for a result set of N parent objects, there will be N + 1 SELECT statements emitted.

The N plus one problem is alleviated using eager loading.

See also:

Relationship Loading Techniques

one to many A style of `relationship()` which links the primary key of the parent mapper's table to the foreign key of a related table. Each unique parent object can then refer to zero or more unique related objects.

The related objects in turn will have an implicit or explicit many to one relationship to their parent object.

An example one to many schema (which, note, is identical to the many to one schema):

```
CREATE TABLE department (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30)
)

CREATE TABLE employee (
    id INTEGER PRIMARY KEY,
    name VARCHAR(30),
    dep_id INTEGER REFERENCES department(id)
)
```

The relationship from `department` to `employee` is one to many, since many employee records can be associated with a single department. A SQLAlchemy mapping might look like:

```
class Department(Base):
    __tablename__ = 'department'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    employees = relationship("Employee")

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    dep_id = Column(Integer, ForeignKey('department.id'))
```

See also:

relationship

many to one

backref

pending This describes one of the major object states which an object can have within a session; a pending object is a new object that doesn't have any database identity, but has been recently associated with a session. When the session emits a flush and the row is inserted, the object moves to the persistent state.

See also:

session_object_states

persistent This describes one of the major object states which an object can have within a session; a persistent object is an object that has a database identity (i.e. a primary key) and is currently associated with a session. Any object that was previously pending and has now been inserted is in the persistent state, as is any object that's been loaded by the session from the database. When a persistent object is removed from a session, it is known as detached.

See also:

session_object_states

polymorphic

polymorphically Refers to a function that handles several types at once. In SQLAlchemy, the term is usually applied to the concept of an ORM mapped class whereby a query operation will return different subclasses based on information in the result set, typically by checking the value of a particular column in the result known as the discriminator.

Polymorphic loading in SQLAlchemy implies that a one or a combination of three different schemes are used to map a hierarchy of classes; "joined", "single", and "concrete". The section inheritance_toplevel describes inheritance mapping fully.

primary key

primary key constraint A constraint that uniquely defines the characteristics of each row. The primary key has to consist of characteristics that cannot be duplicated by any other row. The primary key may consist of a single attribute or multiple attributes in combination. (via Wikipedia)

The primary key of a table is typically, though not always, defined within the `CREATE TABLE` DDL:

```
CREATE TABLE employee (
     emp_id INTEGER,
     emp_name VARCHAR(30),
     dep_id INTEGER,
     PRIMARY KEY (emp_id)
)
```

See also:

http://en.wikipedia.org/wiki/Primary_Key

relationship

relationships A connecting unit between two mapped classes, corresponding to some relationship between the two tables in the database.

The relationship is defined using the SQLAlchemy function `relationship()`. Once created, SQLAlchemy inspects the arguments and underlying mappings involved in order to classify the relationship as one of three types: one to many, many to one, or many to many. With this classification, the relationship construct handles the task of persisting the appropriate linkages in the database in response to in-memory object associations, as well as the job of loading object references and collections into memory based on the current linkages in the database.

See also:

relationship_config_toplevel

release

releases

released In the context of SQLAlchemy, the term "released" refers to the process of ending the usage of a particular database connection. SQLAlchemy features the usage of connection pools, which allows configurability as to the lifespan of database connections. When using a pooled connection, the process of "closing" it, i.e. invoking a statement like `connection.close()`, may have the effect of the connection being returned to an existing pool, or it may have the effect of actually shutting down the underlying TCP/IP connection referred to by that connection - which one takes place depends on configuration as well as the current state of the pool. So we used the term *released* instead, to mean "do whatever it is you do with connections when we're done using them".

The term will sometimes be used in the phrase, "release transactional resources", to indicate more explicitly that what we are actually "releasing" is any transactional state which as accumulated upon the connection. In most situations, the process of selecting from tables, emitting updates, etc. acquires isolated state upon that connection as well as potential row or table locks. This state is all local to a particular transaction on the connection, and is released when we emit a rollback. An important feature of the connection pool is that when we return a connection to the pool, the `connection.rollback()` method of the DBAPI is called as well, so that as the connection is set up to be used again, it's in a "clean" state with no references held to the previous series of operations.

See also:

pooling_toplevel

RETURNING This is a non-SQL standard clause provided in various forms by certain backends, which provides the service of returning a result set upon execution of an INSERT, UPDATE or DELETE statement. Any set of columns from the matched rows can be returned, as though they were produced from a SELECT statement.

The RETURNING clause provides both a dramatic performance boost to common update/select scenarios, including retrieval of inline- or default- generated primary key values and defaults at the moment they were created, as well as a way to get at server-generated default values in an atomic way.

An example of RETURNING, idiomatic to PostgreSQL, looks like:

```
INSERT INTO user_account (name) VALUES ('new name') RETURNING id, timestamp
```

Above, the INSERT statement will provide upon execution a result set which includes the values of the columns `user_account.id` and `user_account.timestamp`, which above should have been generated as default values as they are not included otherwise (but note any series of columns or SQL expressions can be placed into RETURNING, not just default-value columns).

The backends that currently support RETURNING or a similar construct are PostgreSQL, SQL Server, Oracle, and Firebird. The PostgreSQL and Firebird implementations are generally full featured, whereas the implementations of SQL Server and Oracle have caveats. On SQL Server, the clause is known as "OUTPUT INSERTED" for INSERT and UPDATE statements and "OUTPUT DELETED" for DELETE statements; the key caveat is that triggers are not supported in conjunction with this keyword. On Oracle, it is known as "RETURNING…INTO", and requires that the value be placed into an OUT parameter, meaning not only is the syntax awkward, but it can also only be used for one row at a time.

SQLAlchemy's `UpdateBase.returning()` system provides a layer of abstraction on top of the RETURNING systems of these backends to provide a consistent interface for returning columns. The ORM also includes many optimizations that make use of RETURNING when available.

Session The container or scope for ORM database operations. Sessions load instances from the database, track changes to mapped instances and persist changes in a single unit of work when flushed.

See also:

Using the Session

subquery Refers to a `SELECT` statement that is embedded within an enclosing `SELECT`.

A subquery comes in two general flavors, one known as a "scalar select" which specifically must return exactly one row and one column, and the other form which acts as a "derived table" and serves as a source of rows for the FROM clause of another select. A scalar select is eligible to be placed in the WHERE clause, columns clause, ORDER BY clause or HAVING clause of the enclosing select, whereas the derived table form is eligible to be placed in the FROM clause of the enclosing `SELECT`.

Examples:

1. a scalar subquery placed in the columns clause of an enclosing `SELECT`. The subquery in this example is a correlated subquery because part of the rows which it selects from are given via the enclosing statement.

```
SELECT id, (SELECT name FROM address WHERE address.user_id=user.id)
FROM user
```

2. a scalar subquery placed in the WHERE clause of an enclosing `SELECT`. This subquery in this example is not correlated as it selects a fixed result.

```
SELECT id, name FROM user
WHERE status=(SELECT status_id FROM status_code WHERE code='C')
```

3. a derived table subquery placed in the FROM clause of an enclosing `SELECT`. Such a subquery is almost always given an alias name.

```
SELECT user.id, user.name, ad_subq.email_address
FROM
    user JOIN
    (select user_id, email_address FROM address WHERE address_type='Q') AS ad_subq
    ON user.id = ad_subq.user_id
```

transient This describes one of the major object states which an object can have within a session; a transient object is a new object that doesn't have any database identity and has not been associated with a session yet. When the object is added to the session, it moves to the pending state.

See also:

session_object_states

unique constraint

unique key index A unique key index can uniquely identify each row of data values in a database table. A unique key index comprises a single column or a set of columns in a single database table. No two distinct rows or data records in a database table can have the same data value (or combination of data values) in those unique key index columns if NULL values are not used. Depending on its design, a database table may have many unique key indexes but at most one primary key index.

(via Wikipedia)

See also:

http://en.wikipedia.org/wiki/Unique_key#Defining_unique_keys

unit of work This pattern is where the system transparently keeps track of changes to objects and periodically flushes all those pending changes out to the database. SQLAlchemy's Session implements this pattern fully in a manner similar to that of Hibernate.

See also:

Unit of Work by Martin Fowler

Using the Session

WHERE clause The portion of the `SELECT` statement which indicates criteria by which rows should be filtered. It is a single SQL expression which follows the keyword `WHERE`.

```
SELECT user_account.name, user_account.email
FROM user_account
WHERE user_account.name = 'fred' AND user_account.status = 'E'
```

Above, the phrase `WHERE user_account.name = 'fred' AND user_account.status = 'E'` comprises the WHERE clause of the `SELECT`.

PYTHON MODULE INDEX

S