SQLAlchemy 1.1 Documentation

Contents | Index

Search

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 Object Relational Tutorial, presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language, the Expression Language presents a system of representing the primitive constructs of the relational database directly without opinion.

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

A successful application may be constructed using the Expression Language exclusively, though the application will need to define its own system of translating application concepts into individual database messages and from individual database result sets. Alternatively, an application constructed with the ORM may, in advanced scenarios, make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

The following tutorial is in doctest format, meaning each `>` line represents something you can type at a Python command prompt, and the following text represents the expected return value. The tutorial has no prerequisites.

Version Check

A quick check to verify that we are on at least **version 1.1** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__  # doctest: +SKIP
1.1.0
```

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 `c`:

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

The `e` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard `l` 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 `F`. 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 `c` is an instance of `E`, 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 `s` module.

The first time a method like `E` or `E` is called, the `E` establishes a

Lazy Connecting

real DBAPI connection to the database, which is then used to emit the SQL.

The E , when first returned by c , 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.

See also

Database Urls - includes examples of c connecting to several kinds of databases with links to more information.

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 C , and in all cases a C is associated with a T . A collection of T objects and their associated child objects is referred to as **database metadata**. In this tutorial we will explicitly lay out several T objects, but note that SA can also "import" whole sets of T objects automatically from an existing database (this process is called **table reflection**).

We define our tables all within a catalog called M , using the T 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 T objects, as well as how to create them from an existing database automatically, is described in Describing Databases with MetaData.

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

```
>>> metadata.create_all(engine)
SE...
```

SQL

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 S type as below:

```
Column('name', String(50))
```

The length field on S , as well as similar precision/scale fields available on I , N , 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 S construct:

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

A full, foolproof T 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 `T` 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.

Insert Expressions

The first SQL expression we'll create is the `I` 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 `s` function:

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

Notice above that the INSERT statement names every column in the `u` table. This can be limited by using the `v` 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 `v` method limited the VALUES clause to just two columns, the actual data we placed in `v` didn't get rendered into the string; instead we got named bind parameters. As it turns out, our data *is* stored within our `I` 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'}
```

Executing

The interesting part of an `I` 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 `e` object we created is a repository for database connections capable of issuing SQL to the database. To acquire a connection, we use the `c` method:

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

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

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

So the INSERT statement was now issued to the database. Although we got positional "qmark" bind parameters instead of "named" bind parameters in the output. How come ? Because when executed, the `C` used the SQLite **dialect** to help generate the statement; when we use

the s 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 r variable we got when we called e ? As the SQLAlchemy C object references a DBAPI connection, the result, known as a R 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 R :

```
>>> result.inserted_primary_key
[1]
```

The value of 1 was automatically generated by SQLite, but only because we did not specify the i column in our I 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 D knows the specific steps needed to determine the correct value (or values; note that R returns a list so that it supports composite primary keys). Methods here range from using c , to selecting from a database-specific function, to using I syntax; this all occurs transparently.

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 I statement is usually compiled against the parameters sent to the e method on C , so that there's no need to use the v keyword with I . Lets create a generic I statement again and use it in the "normal" way:

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

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

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

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

Above, we again relied upon SQLite's automatic generation of primary key identifiers for each a 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 I 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 `i`, `u` and `d` constructs.

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 `s` function:

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

Above, we issued a basic `s` call, placing the `u` table within the COLUMNS clause of the select, and then executing. SQLAlchemy expanded the `u` table into the set of each of its columns, and also generated a FROM clause for us. The result returned is again a `R` object, which acts much like a DBAPI cursor, including methods such as `f` and `f`. 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:

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

SQL

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 `C` objects directly as keys:

```
>>> for row in conn.execute(s):
...     print("name:", row[users.c.name], "; fullname:", row[users.c.fullname])
name: jack ; fullname: Jack Jones
name: wendy ; fullname: Wendy Williams
```

SQL

Result sets which have pending rows remaining should be explicitly closed before discarding. While the cursor and connection resources referenced by the `R` 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 `C` objects from our `T`. These are available as named attributes off the `c` attribute of the `T` object:

```
>>> s = select([users.c.name, users.c.fullname])
>>> result = conn.execute(s)
```

SQL

```
>>> 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 `s` construct only has a list containing columns. How does this work ? Let's try putting *two* tables into our `s` statement:

SQL

```
>>> for row in conn.execute(select([users, addresses])):
...     print(row)
(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 `u` table is produced against each row from the `a` table. So to put some sanity into this statement, we need a WHERE clause. We do that using `S`:

SQL

```
>>> s = select([users, addresses]).where(users.c.id == addresses.c.user_id)
>>> for row in conn.execute(s):
...     print(row)
(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 `s` which had the effect of adding `W u = a` to our statement, and our results were managed down so that the join of `u` and `a` rows made sense. But let's look at that expression? It's using just a Python equality operator between two different `C` objects. It should be clear that something is up. Saying `1 = 1` produces `T`, and `1 = 2` produces `F`, 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 `T` nor a `F`. 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 `I` and `s` objects we've made so far, thanks to Python's `_` builtin; you call `s` 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 `C`.

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 `C`; we can use the same trick we did with the `I` 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 `C` is important! If we use `+` with two string based columns (recall we put types like `I` and `S` on our `C` 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 `E` that's connected to a MySQL database; the `|` operator now compiles as MySQL's `c` function.

If you have come across an operator which really isn't available, you can always use the `C` 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*.

Operator Customization

While `C` 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 Redefining and Creating New Operators for a description.

Conjunctions

We'd like to show off some of our operators inside of `s` 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. `a`, `o`, and

`n` can work from the corresponding functions SQLAlchemy provides (notice we also throw in a `l`):

```
>>> 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, `b` and `l`. `b` produces a BETWEEN clause, and `l` is used in a column expression to produce labels using the `A` 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 `a` is to chain together multiple `w` 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 `s` construct through successive method calls is called method chaining.

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 `t` construct is used to compose a textual statement that is passed to the database mostly unchanged. Below, we create a `t` 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)")
>>> conn.execute(s, x='m', y='z', e1='%@aol.com', e2='%@msn.com').fetchall()
[(u'Wendy Williams, wendy@aol.com',)]
```

SQL

Above, we can see that bound parameters are specified in `t` 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 `e` method as additional arguments.

Specifying Bound Parameter Behaviors

The `t` construct supports pre-established bound values using the `T` 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

`T` - full method description

Specifying Result-Column Behaviors

We may also specify information about the result columns using the `T` 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 `T` method, we get back a `T` object that supports the full suite of `T` 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 `T` 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
...     )
>>> result = conn.execute(stmt)
```

SQL

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 `e` 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 `i` 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 `C` objects may seem unusual, it is in fact the only system used by the ORM, which occurs transparently beneath the facade of the `Q` object; in this way, the `T` method is typically very applicable to textual statements to be used in an ORM context. The example at Using Textual SQL illustrates a simple usage.

New in version 1.1: The `T` 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

`T` - full method description

Using Textual SQL - integrating ORM-level queries with `t`

Using text() fragments inside bigger statements

`t` can also be used to produce fragments of SQL that can be freely within a `s` object, which accepts `t` objects as an argument for most of its builder functions. Below, we combine

the usage of `t` within a `s` object. The `s` construct provides the "geometry" of the statement, and the `t` construct provides the textual content within this form. We can build a statement without the need to refer to any pre-established `T` 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'))
>>> conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()
[(u'Wendy Williams, wendy@aol.com',)]
```

SQL

Changed in version 1.0.0: The `s` construct emits warnings when string SQL fragments are coerced to `t`, and `t` should be used explicitly. See Warnings emitted when coercing full SQL fragments into text() for background.

Using More Specific Text with `t`, `l`, and `C`

We can move our level of structure back in the other direction too, by using `c`, `l`, and `t` for some of the key elements of our statement. Using these constructs, we can get some more expression capabilities than if we used `t` 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 `T` based metadata. Below, we also specify the `S` datatype for two of the key `l` objects, so that the string-specific concatenation operator becomes available. We also use `l` in order to use table-qualified expressions, e.g. `u`, that will be rendered as is; using `c` 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'))

>>> conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()
[(u'Wendy Williams, wendy@aol.com',)]
```

SQL

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 `s` construct, we can refer to it directly by passing the string straight into `s` or `s`, 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")
```

```
>>> conn.execute(stmt).fetchall()
[(2, 4)]
```
SQL

We can use modifiers like `a` or `d` 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"))

>>> conn.execute(stmt).fetchall()
[(2, 4)]
```
SQL

Note that the string feature here is very much tailored to when we have already used the `l` method to create a specifically-named label. In other cases, we always want to refer to the `C` object directly so that the expression system can make the most effective choices for rendering. Below, we illustrate how using the `C` 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

>>> conn.execute(stmt).fetchall()
[(2, u'wendy', u'Wendy Williams', 1, u'jack', u'Jack Jones')]
```
SQL

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 `A` 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 `T`, `s` construct, or other selectable can be turned into an alias using the `F` method, which produces a `A` construct. As an example, suppose we know that our user `j` 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 `a` table, once for each address. We create two `A` constructs against `a`, and then use them both within a `s` 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'
...         ))
>>> conn.execute(s).fetchall()
[(1, u'jack', u'Jack Jones')]
```
SQL

Note that the `A` construct generated the names `a` and `a` 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 `a` alias, the name would come out as `a`. 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 `A` 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 `F` 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 `u` table back to the `s` we've created by making an alias of the entire statement. The `c` directive is to avoid SQLAlchemy's attempt to "correlate" the inner `u` table with the outer one:

```
>>> a1 = s.correlate(None).alias()
>>> s = select([users.c.name]).where(users.c.id == a1.c.id)
>>> conn.execute(s).fetchall()
[(u'jack',)]
```
SQL

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 `s` construct. But if we want to make a real "JOIN" or "OUTERJOIN" construct, we use the `j` and `o` 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 `F` object which we placed on the `a` table way at the beginning of this tutorial. Already the `j` 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 `s` 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 `s` method:

```
>>> s = select([users.c.fullname]).select_from(
...    users.join(addresses,
...              addresses.c.email_address.like(users.c.name + '%'))
...    )
>>> conn.execute(s).fetchall()
[(u'Jack Jones',), (u'Jack Jones',), (u'Wendy Williams',)]
```
SQL

The `o` method creates `L O J` constructs, and is used in the same way as `j`:

```
>>> 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 `o` 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 `O`) 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

e

e

J

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 b 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'))
>>> conn.execute(s, username='wendy').fetchall()
[(2, u'wendy', u'Wendy Williams')]
```

SQL

Another important aspect of b 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("'%'"
>>> conn.execute(s, username='wendy').fetchall()
[(2, u'wendy', u'Wendy Williams')]
```

SQL

b 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)
>>> conn.execute(s, name='jack').fetchall()
[(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com'), (1, u'jack', u'Jack Jones', 2, 1
```

SQL

See also

b

Functions

SQL functions are created using the f 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 `s` 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()
SELECT max(addresses.email_address) AS maxemail
FROM addresses
()
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 `c` 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 `c` statement twice with different bind parameters, the `u` 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

f

Window Functions

Any F, including functions generated by f, can be turned into a "window function", that is an OVER clause, using the F 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
```

See also

o

F

Unions and Other Set Operations

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

```
>>> 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)

>>> conn.execute(u).fetchall()
[(1, 1, u'jack@yahoo.com')]
```

SQL

Also available, though not supported on all databases, are i, i, e, and e:

```
>>> 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'))
... )

>>> conn.execute(u).fetchall()
[(1, 1, u'jack@yahoo.com'), (4, 2, u'wendy@aol.com')]
```

SQL

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 . 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'))
... )
>>> conn.execute(u).fetchall()
[(1, 1, u'jack@yahoo.com')]
```

SQL

See also

`u`

`u`

`i`

`i`

`e`

`e`

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 `s` construct can be modified to act as a column expression by calling either the `a` or `l` method:

```
>>> stmt = select([func.count(addresses.c.id)]).\
...             where(users.c.id == addresses.c.user_id).\
...             as_scalar()
```

The above construct is now a `S` object, and is no longer part of the `F` hierarchy; it instead is within the `C` family of expression constructs. We can place this construct the same as any other column within another `s`:

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

To apply a non-anonymous column name to our scalar select, we create it using `S` 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()
SELECT users.name, (SELECT count(addresses.id) AS count_1
FROM addresses
WHERE users.id = addresses.user_id) AS address_count
FROM users
()
[(u'jack', 2), (u'wendy', 2)]
```

See also

`S`

`S`

Correlated Subqueries

Notice in the examples on Scalar Selects, the FROM clause of each embedded select did not contain the `u` 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()
```

```
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',)
```

```
[(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 `a` table but not the `u` table, even if both were present in the enclosing SELECT, we use the `c` 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()
```

```
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',)
```

```
[(u'jack', u'jack@yahoo.com'), (u'jack', u'jack@msn.com')]
```

To entirely disable a statement from correlating, we can pass `N` 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()
```

```
SELECT users.name
 FROM users
 WHERE users.id = (SELECT users.id
  FROM users
  WHERE users.name = ?)
('wendy',)
```

```
[(u'wendy',)]
```

We can also control correlation via exclusion, using the `S` method. Such as, we can write our SELECT for the `u` table by telling it to correlate all FROM clauses except for `u`:

```
>>> 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()
```

```
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',)
```

```
[(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 `t J ( S A s` 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 `S` 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 `S` method acts a lot like the `S` method, including that we can specify an optional name. However the construct is the `L` construct instead of an `A` which provides for the LATERAL keyword as well as special instructions to allow correlation from inside the FROM clause of the enclosing statement.

The `S` method interacts normally with the `S` and `S` 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 `S`, and is explicit to all tables except those specified to `S`.

New in version 1.1: Support for the LATERAL keyword and lateral correlation.

See also

`L`

`S`

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

Ordering is done by passing column expressions to the `o` method:

```
>>> stmt = select([users.c.name]).order_by(users.c.name)
>>> conn.execute(stmt).fetchall()
```

```
SELECT users.name
FROM users ORDER BY users.name
()
```

```
[(u'jack',), (u'wendy',)]
```

Ascending or descending can be controlled using the `a` and `d` modifiers:

```
>>> stmt = select([users.c.name]).order_by(users.c.name.desc())
>>> conn.execute(stmt).fetchall()
```

```
SELECT users.name
FROM users ORDER BY users.name DESC
()
```

```
[(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 `g` 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()
```

```
SELECT users.name, count(addresses.id) AS count_1
FROM users JOIN addresses
    ON users.id = addresses.user_id
GROUP BY users.name
()
```

```
[(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 `h` 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()
```

```
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,)
```

```
[(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 `S` method:

```
>>> stmt = select([users.c.name]).\
...             where(addresses.c.email_address.
...                     contains(users.c.name)).\
...             distinct()
>>> conn.execute(stmt).fetchall()
```

```
SELECT DISTINCT users.name
FROM users, addresses
WHERE (addresses.email_address LIKE '%%' || users.name || '%%')
()
```

```
[(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 `l` and `o` 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()
```

```
SELECT users.name, addresses.email_address
FROM users JOIN addresses ON users.id = addresses.user_id
 LIMIT ? OFFSET ?
(1, 1)
```

```
[(u'jack', u'jack@msn.com')]
```

Inserts, Updates and Deletes

We've seen `i` demonstrated earlier in this tutorial. Where `i` produces INSERT, the `u` method produces UPDATE. Both of these constructs feature a method called `v` which specifies the VALUES or SET clause of the statement.

The `v` method accommodates any column expression as a value:

```
>>> stmt = users.update().\
...             values(fullname="Fullname: " + users.c.name)
>>> conn.execute(stmt)
```

```
UPDATE users SET fullname=(? || users.name)
('Fullname: ',)
COMMIT
```

```
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

When using `i` or `u` 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 `e` at execution time. However, if we wish to use explicitly targeted named parameters with composed expressions, we need to use the `b` construct. When using `b` with `i` or `u`, 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'},
...     ])
```

```
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 `u` 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)
```

```
UPDATE users SET name=? WHERE users.name = ?
('ed', 'jack')
COMMIT
```

```
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

When using `u` in an "executemany" context, we may wish to also use explicitly named bound parameters in the WHERE clause. Again, `b` 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'},
...     ])
UPDATE users SET name=? WHERE users.name = ?
(('ed', 'jack'), ('mary', 'wendy'), ('jake', 'jim'))
COMMIT
<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))
UPDATE users SET fullname=(SELECT addresses.email_address
    FROM addresses
    WHERE addresses.user_id = users.id
    LIMIT ? OFFSET ?)
(1, 0)
COMMIT
<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 `u` 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 `U`:

```
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, '%%')
```

SQLAlchemy doesn't do anything special when these constructs are used on a non-supporting

database. The `U F` syntax generates by default when multiple tables are present, and the statement will be rejected by the database if this syntax is not supported.

Parameter-Ordered Updates

The default behavior of the `u` construct when rendering the SET clauses is to render them using the column ordering given in the originating `T` 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 `U` 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 `p` flag may be used. When using this flag, we supply a **Python list of 2-tuples** as the argument to the `U` 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 `p` flag.

Deletes

Finally, a delete. This is accomplished easily enough using the `d` construct:

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

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

Matched Row Counts

Both of `u` and `d` 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 `r`:

```
>>> result = conn.execute(users.delete())
DELETE FROM users
```

```
()
COMMIT
```

```
>>> result.rowcount
1
```

Further Reference

Expression Language Reference: SQL Statements and Expressions API

Database Metadata Reference: Describing Databases with MetaData

Engine Reference: *Engine Configuration*

Connection Reference: Working with Engines and Connections

Types Reference: Column and Data Types