SQLAlchemy 1.0 Documentation

LEGACY VERSION

Contents | Index

Search terms: search...

ads via Carbon

SQLAlchemy ORM

Object Relational Tutorial
Mapper Configuration
Relationship Configuration
Loading Objects
- Loading Columns
- Relationship Loading Techniques
- Loading Inheritance Hierarchies
- Constructors and Object Initialization
- **Query API**
  - The Query Object
  - ORM-Specific Query Constructs

Using the Session

Query API

This section presents the API reference for the ORM `Query` object. For a walkthrough of how to use this object, see Object Relational Tutorial.

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 Object Relational Tutorial.

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

Construct a `Query` directly.

E.g.:

```
q = Query([User, Address], session=some_session)
```

The above is equivalent to:

```
q = some_session.query(User, Address)
```

Parameters:

- **entities** – a sequence of entities and/or SQL expressions.
- **session** – a `Session` with which the `Query` will be associated. Optional; a `Query` can be associated with a `Session` generatively via the `Query.with_session()` method as well.

See also

`Session.query()`

`Query.with_session()`

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Pending deprecation: `add_column()` will be superseded by `add_columns()`.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).

New in version 0.7.6.

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

`SelectBase.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 accomodate 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, Updates and Deletes - 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.

`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

Using Textual 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`

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 XML Persistence 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.

- **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

Querying with Joins in the ORM tutorial.

Mapping Class Inheritance Hierarchies 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 Dogpile 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 Column Loading and Relationship Loading Techniques for reference documentation.

order_by(**criterion*)

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

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

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

outerjoin(**props*, ***kwargs*)

Create a left outer join against this `Query` object's criterion and apply generatively, 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_nam
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.

This method is similar to the `Query.select_from()` method, in that it sets the FROM clause of the query. However, where `Query.select_from()` only affects what is placed in the FROM, this method also applies the given selectable to replace the FROM which the selected entities would normally select from.

The given `from_obj` must be an instance of a `FromClause`, e.g. a `select()` or `Alias` construct.

An example would be a `Query` that selects `User` entities, but uses `Query.select_entity_from()` to have the entities selected from a `select()` construct instead of the base `user` table:

```
select_stmt = select([User]).where(User.id == 7)

q = session.query(User).\
        select_entity_from(select_stmt).\
        filter(User.name == 'ed')
```

The query generated will select `User` entities directly from the given `select()` construct, and will be:

```
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
```

Notice above that even the WHERE criterion was “adapted” such that the `anon_1` subquery effectively replaces all references to the `user` table, except for the one that it refers to internally.

Compare this to `Query.select_from()`, which as of version 0.9, does not affect existing entities. The statement below:

```
q = session.query(User).\
        select_from(select_stmt).\
        filter(User.name == 'ed')
```

Produces SQL where both the `user` table as well as the `select_stmt` construct are present as separate elements in the FROM clause. No “adaptation” of the `user` table is applied:

```
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
```

`Query.select_entity_from()` maintains an older behavior of `Query.select_from()`. In modern usage, similar results can also be achieved using `aliased()`:

```
select_stmt = select([User]).where(User.id == 7)
user_from_select = aliased(User, select_stmt.alias())

q = session.query(user_from_select)
```

Parameters:
from_obj – a `FromClause` object that will replace the FROM clause of this `Query`.

See also

`Query.select_from()`

New in version 0.8: `Query.select_entity_from()` was added to specify the specific behavior of entity replacement, however the `Query.select_from()` maintains this behavior as well until 0.9.

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 Multiple 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, Updates and Deletes - 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*)

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().state
```

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*)

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

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

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

with_polymorphic(*cls_or_mappers*, *selectable=None*, *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 Using 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 Building 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 with most eager loading schemes, including subqueryload and joinedload with collections**. For this reason, 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` dialect 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 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 Hybrid Attributes).

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*)

Bases: `sqlalchemy.orm.base.InspectionAttr`

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

Runtime Inspection API

class sqlalchemy.orm.query.**Bundle**(*name*, **exprs*, ***kw*)

Bases: sqlalchemy.orm.base.InspectionAttr

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

Column Bundles

__init__(*name*, **exprs*, ***kw*)

Construct a new Bundle.

e.g.:

```
bn = Bundle("mybundle", MyClass.x, MyClass.y)

for row in session.query(bn).filter(
        bn.c.x == 5).filter(bn.c.y == 4):
    print(row.mybundle.x, row.mybundle.y)
```

Parameters:

- **name** – name of the bundle.
- ***exprs** – columns or SQL expressions comprising the bundle.
- **single_entity=False** – if True, rows for this Bundle can be returned as a "single entity" outside of any enclosing tuple in the same manner as a mapped entity.

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

Column 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**

Bases: `sqlalchemy.util._collections.AbstractKeyedTuple`

`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

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`()

inherited from the `keys()` *method of* `AbstractKeyedTuple`

Return a list of string key names for this `KeyedTuple`.

See also

`KeyedTuple._fields`

class sqlalchemy.orm.strategy_options.**Load**(*entity*)

Bases: sqlalchemy.sql.expression.Generative, sqlalchemy.orm.interfaces.MapperOption

Represents loader options which modify the state of a Query in order to affect how various mapped attributes are loaded.

> *New in version 0.9.0:* The Load() system is a new foundation for the existing system of loader options, including options such as orm.joinedload(), orm.defer(), and others. In particular, it introduces a new method-chained system that replaces the need for dot-separated paths as well as "_all()" options such as orm.joinedload_all().

A Load object can be used directly or indirectly. To use one directly, instantiate given the parent class. This style of usage is useful when dealing with a Query that has multiple entities, or when producing a loader option that can be applied generically to any style of query:

```
myopt = Load(MyClass).joinedload("widgets")
```

The above myopt can now be used with Query.options():

```
session.query(MyClass).options(myopt)
```

The Load construct is invoked indirectly whenever one makes use of the various loader options that are present in sqlalchemy.orm, including options such as orm.joinedload(), orm.defer(), orm.subqueryload(), and all the rest. These constructs produce an "anonymous" form of the Load object which tracks attributes and options, but is not linked to a parent class until it is associated with a parent Query:

```
# produce "unbound" Load object
myopt = joinedload("widgets")

# when applied using options(), the option is "bound" to the
# class observed in the given query, e.g. MyClass
session.query(MyClass).options(myopt)
```

Whether the direct or indirect style is used, the Load object returned now represents a specific "path" along the entities of a Query. This path can be traversed using a standard method-chaining approach. Supposing a class hierarchy such as User, User.addresses -> Address, User.orders -> Order and Order.items -> Item, we can specify a variety of loader options along each element in the "path":

```
session.query(User).options(
            joinedload("addresses"),
            subqueryload("orders").joinedload("items")
        )
```

Where above, the addresses collection will be joined-loaded, the orders collection will be subquery-loaded, and within that subquery load the items collection will be joined-loaded.

baked_lazyload(*attr*)

Produce a new `Load` object with the `orm.baked_lazyload()` option applied.

See `orm.baked_lazyload()` for usage examples.

contains_eager(*attr*, *alias=None*)

Produce a new `Load` object with the `orm.contains_eager()` option applied.

See `orm.contains_eager()` for usage examples.

defaultload(*attr*)

Produce a new `Load` object with the `orm.defaultload()` option applied.

See `orm.defaultload()` for usage examples.

defer(*key*)

Produce a new `Load` object with the `orm.defer()` option applied.

See `orm.defer()` for usage examples.

immediateload(*attr*)

Produce a new `Load` object with the `orm.immediateload()` option applied.

See `orm.immediateload()` for usage examples.

joinedload(*attr*, *innerjoin=None*)

Produce a new `Load` object with the `orm.joinedload()` option applied.

See `orm.joinedload()` for usage examples.

lazyload(*attr*)

Produce a new `Load` object with the `orm.lazyload()` option applied.

See `orm.lazyload()` for usage examples.

load_only(**attrs*)

Produce a new `Load` object with the `orm.load_only()` option applied.

See `orm.load_only()` for usage examples.

noload(*attr*)

Produce a new `Load` object with the `orm.noload()` option applied.

See `orm.noload()` for usage examples.

process_query(*query*)

Apply a modification to the given `Query`.

process_query_conditionally(*query*)

same as process_query(), except that this option may not apply to the given query.

This is typically used during a lazy load or scalar refresh operation to propagate options stated in the original Query to the new Query being used for the load. It occurs for those options that specify propagate_to_loaders=True.

subqueryload(*attr*)

Produce a new `Load` object with the `orm.subqueryload()` option applied.

See `orm.subqueryload()` for usage examples.

undefer(*key*)

Produce a new `Load` object with the `orm.undefer()` option applied.

See `orm.undefer()` for usage examples.

undefer_group(*name*)

Produce a new `Load` object with the `orm.undefer_group()` option applied.

See `orm.undefer_group()` for usage examples.

`sqlalchemy.orm.` **join**(*left*, *right*, *onclause=None*, *isouter=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*, *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*)

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.

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

Parameters:

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

Website generation by zeekofile, with huge thanks to the Blogofile project.