ORM Examples

The SQLAlchemy distribution includes a variety of code examples illustrating a select set of patterns, some typical and some not so typical. All are runnable and can be found in the `/` directory of the distribution. Descriptions and source code for all can be found here.

Additional SQLAlchemy examples, some user contributed, are available on the wiki at http://www.sqlalchemy.org/trac/wiki/UsageRecipes.

Mapping Recipes

Adjacency List

An example of a dictionary-of-dictionaries structure mapped using an adjacency list model.

E.g.:

```
node = TreeNode('rootnode')
node.append('node1')
node.append('node3')
session.add(node)
session.commit()

dump_tree(node)
```

Listing of files:

- adjacency_list.py

Associations

Examples illustrating the usage of the "association object" pattern, where an intermediary class mediates the relationship between two classes that are associated in a many-to-many pattern.

Listing of files:

- proxied_association.py - same example as basic_association, adding in usage of `s` to make explicit references to `O` optional.
- basic_association.py - illustrate a many-to-many relationship between an "Order" and a collection of "Item" objects, associating a purchase price with each via an association object called "OrderItem"
- dict_of_sets_with_default.py - an advanced association proxy example which illustrates nesting of association proxies to produce multi-level Python collections, in this case a dictionary with string keys and sets of integers as values, which conceal the underlying mapped classes.

Directed Graphs

An example of persistence for a directed graph structure. The graph is stored as a collection of edges, each referencing both a "lower" and an "upper" node in a table of nodes. Basic persistence and querying for lower- and upper- neighbors are illustrated:

```
n2 = Node(2)
n5 = Node(5)
n2.add_neighbor(n5)
print n2.higher_neighbors()
```

Listing of files:

- directed_graph.py - a directed graph example.

Dynamic Relations as Dictionaries

Illustrates how to place a dictionary-like facade on top of a "dynamic" relation, so that dictionary operations (assuming simple string keys) can operate upon a large collection without loading the full collection at once.

Listing of files:

- dynamic_dict.py

Generic Associations

Illustrates various methods of associating multiple types of parents with a particular child object.

The examples all use the declarative extension along with declarative mixins. Each one presents the identical use case at the end - two classes, `C` and `S`, both subclassing the `H` mixin, which ensures that the parent class is provided with an `a` collection which contains `A` objects.

The discriminator_on_association.py and generic_fk.py scripts are modernized versions of recipes presented in the 2007 blog post Polymorphic Associations with SQLAlchemy.

Listing of files:

- table_per_association.py - Illustrates a mixin which provides a generic association via a individually generated association tables for each parent class. The associated objects themselves are persisted in a single table shared among all parents.
- generic_fk.py - Illustrates a so-called "generic foreign key", in a similar fashion to that of popular frameworks such as Django, ROR, etc. This approach bypasses standard referential integrity practices, in that the "foreign key" column is not actually constrained to refer to any particular table; instead, in-application logic is used to determine which table is referenced.
- discriminator_on_association.py - Illustrates a mixin which provides a generic association using a single target table and a single association table, referred to by all parent tables. The association table contains a "discriminator" column which determines what type of parent object associates to each particular row in the association table.
- table_per_related.py - Illustrates a generic association which persists association objects within individual tables, each one generated to persist those objects on behalf of a particular parent class.

Large Collections

Large collection example.

Illustrates the options to use with `r` when the list of related objects is very large, including:

- "dynamic" relationships which query slices of data as accessed
- how to use ON DELETE CASCADE in conjunction with `p` to greatly improve the performance of related collection deletion.

Listing of files:

- large_collection.py

Materialized Paths

Illustrates the "materialized paths" pattern for hierarchical data using the SQLAlchemy ORM.

Listing of files:

- materialized_paths.py - Illustrates the "materialized paths" pattern.

Nested Sets

Illustrates a rudimentary way to implement the "nested sets" pattern for hierarchical data using the SQLAlchemy ORM.

Listing of files:

- nested_sets.py - Celko's "Nested Sets" Tree Structure.

Performance

A performance profiling suite for a variety of SQLAlchemy use cases.

Each suite focuses on a specific use case with a particular performance profile and associated implications:

- bulk inserts

- individual inserts, with or without transactions
- fetching large numbers of rows
- running lots of short queries

All suites include a variety of use patterns illustrating both Core and ORM use, and are generally sorted in order of performance from worst to greatest, inversely based on amount of functionality provided by SQLAlchemy, greatest to least (these two things generally correspond perfectly).

A command line tool is presented at the package level which allows individual suites to be run:

```
$ python -m examples.performance --help
usage: python -m examples.performance [-h] [--test TEST] [--dburl DBURL]
                                      [--num NUM] [--profile] [--dump]
                                      [--runsnake] [--echo]

                                      {bulk_inserts,large_resultsets,single_inserts}

positional arguments:
  {bulk_inserts,large_resultsets,single_inserts}
                        suite to run

optional arguments:
  -h, --help            show this help message and exit
  --test TEST           run specific test name
  --dburl DBURL         database URL, default sqlite:///profile.db
  --num NUM             Number of iterations/items/etc for tests; default is 0
                        module-specific
  --profile             run profiling and dump call counts
  --dump                dump full call profile (implies --profile)
  --runsnake            invoke runsnakerun (implies --profile)
  --echo                Echo SQL output
```

An example run looks like:

```
$ python -m examples.performance bulk_inserts
```

Or with options:

```
$ python -m examples.performance bulk_inserts \
    --dburl mysql+mysqldb://scott:tiger@localhost/test \
    --profile --num 1000
```

See also

How can I profile a SQLAlchemy powered application?

File Listing

Listing of files:

- large_resultsets.py - In this series of tests, we are looking at time to load a large number of very small and simple rows.
- bulk_updates.py - This series of tests illustrates different ways to UPDATE a large number of rows in bulk.
- short_selects.py - This series of tests illustrates different ways to SELECT a single record by primary key
- bulk_inserts.py - This series of tests illustrates different ways to INSERT a large number of rows in bulk.
- __main__.py - Allows the examples/performance package to be run as a script.
- single_inserts.py - In this series of tests, we’re looking at a method that inserts a row within a distinct transaction, and afterwards returns to essentially a “closed” state. This would be analogous to an API call that starts up a database connection, inserts the row, commits and closes.

Running all tests with time

This is the default form of run:

```
$ python -m examples.performance single_inserts
Tests to run: test_orm_commit, test_bulk_save,
              test_bulk_insert_dictionaries, test_core,
                test_core_query_caching, test_dbapi_raw_w_connect,
                test_dbapi_raw_w_pool

test_orm_commit : Individual INSERT/COMMIT pairs via the
    ORM (10000 iterations); total time 13.690218 sec
test_bulk_save : Individual INSERT/COMMIT pairs using
    the "bulk" API  (10000 iterations); total time 11.290371 sec
test_bulk_insert_dictionaries : Individual INSERT/COMMIT pairs using
    the "bulk" API with dictionaries (10000 iterations);
    total time 10.814626 sec
test_core : Individual INSERT/COMMIT pairs using Core.
    (10000 iterations); total time 9.665620 sec
test_core_query_caching : Individual INSERT/COMMIT pairs using Core
    with query caching (10000 iterations); total time 9.209010 sec
test_dbapi_raw_w_connect : Individual INSERT/COMMIT pairs w/ DBAPI +
    connection each time (10000 iterations); total time 9.551103 sec
test_dbapi_raw_w_pool : Individual INSERT/COMMIT pairs w/ DBAPI +
    connection pool (10000 iterations); total time 8.001813 sec
```

Dumping Profiles for Individual Tests

A Python profile output can be dumped for all tests, or more commonly individual tests:

```
$ python -m examples.performance single_inserts --test test_core --num 1000 --dump
Tests to run: test_core
test_core : Individual INSERT/COMMIT pairs using Core. (1000 iterations); total fn call
         186109 function calls (186102 primitive calls) in 1.089 seconds

   Ordered by: internal time, call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.634    0.001    0.634    0.001 {method 'commit' of 'sqlite3.Connection'
     1000    0.154    0.000    0.154    0.000 {method 'execute' of 'sqlite3.Cursor' obj
     1000    0.021    0.000    0.074    0.000 /Users/classic/dev/sqlalchemy/lib/sqlalch
     1000    0.015    0.000    0.034    0.000 /Users/classic/dev/sqlalchemy/lib/sqlalch
        1    0.012    0.012    1.091    1.091 examples/performance/single_inserts.py:79

    ...
```

Using RunSnake

This option requires the RunSnake command line tool be installed:

```
$ python -m examples.performance single_inserts --test test_core --num 1000 --runsnake
```

A graphical RunSnake output will be displayed.

Writing your Own Suites

The profiler suite system is extensible, and can be applied to your own set of tests. This is a valuable technique to use in deciding upon the proper approach for some performance-critical set of routines. For example, if we wanted to profile the difference between several kinds of loading, we can create a file `t`, with the following content:

```
from examples.performance import Profiler
from sqlalchemy import Integer, Column, create_engine, ForeignKey
from sqlalchemy.orm import relationship, joinedload, subqueryload, Session
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
engine = None
session = None


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


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


# Init with name of file, default number of items
Profiler.init("test_loads", 1000)


@Profiler.setup_once
def setup_once(dburl, echo, num):
    "setup once.  create an engine, insert fixture data"
    global engine
    engine = create_engine(dburl, echo=echo)
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)
    sess = Session(engine)
    sess.add_all([
        Parent(children=[Child() for j in range(100)])
        for i in range(num)
    ])
    sess.commit()


@Profiler.setup
def setup(dburl, echo, num):
    "setup per test.  create a new Session."
    global session
    session = Session(engine)
    # pre-connect so this part isn't profiled (if we choose)
    session.connection()


@Profiler.profile
def test_lazyload(n):
    "load everything, no eager loading."

    for parent in session.query(Parent):
        parent.children


@Profiler.profile
def test_joinedload(n):
    "load everything, joined eager loading."

    for parent in session.query(Parent).options(joinedload("children")):
        parent.children


@Profiler.profile
def test_subqueryload(n):
    "load everything, subquery eager loading."

    for parent in session.query(Parent).options(subqueryload("children")):
        parent.children

if __name__ == '__main__':
    Profiler.main()
```

We can run our new script directly:

```
$ python test_loads.py  --dburl postgresql+psycopg2://scott:tiger@localhost/test
Running setup once...
Tests to run: test_lazyload, test_joinedload, test_subqueryload
test_lazyload : load everything, no eager loading. (1000 iterations); total time 11.971
test_joinedload : load everything, joined eager loading. (1000 iterations); total time
test_subqueryload : load everything, subquery eager loading. (1000 iterations); total t
```

As well as see RunSnake output for an individual test:

```
$ python test_loads.py  --num 100 --runsnake --test test_joinedload
```

Relationship Join Conditions

Examples of various `o` configurations, which make use of the `p` argument to compose special types of join conditions.

Listing of files:

- cast.py - Illustrate a `r` that joins two columns where those columns are not of the same type, and a CAST must be used on the SQL side in order to match them.
- threeway.py - Illustrate a "three way join" - where a primary table joins to a remote table via an association table, but then the primary table also needs to refer to some columns in the remote table directly.

XML Persistence

Illustrates three strategies for persisting and querying XML documents as represented by ElementTree in a relational database. The techniques do not apply any mappings to the ElementTree objects directly, so are compatible with the native cElementTree as well as lxml, and can be adapted to suit any kind of DOM representation system. Querying along xpath-like strings is illustrated as well.

E.g.:

```
# parse an XML file and persist in the database
doc = ElementTree.parse("test.xml")
session.add(Document(file, doc))
session.commit()

# locate documents with a certain path/attribute structure
for document in find_document('/somefile/header/field2[@attr=foo]'):
    # dump the XML
    print document
```

Listing of files:

- pickle.py - illustrates a quick and dirty way to persist an XML document expressed using ElementTree and pickle.
- adjacency_list.py - Illustrates an explicit way to persist an XML document expressed using ElementTree.
- optimized_al.py - Uses the same strategy as `a`, but associates each DOM row with its owning document row, so that a full document of DOM nodes can be loaded using O(1) queries - the construction of the "hierarchy" is performed after the load in a non-recursive fashion and is more efficient.

Versioning Objects

Versioning with a History Table

Illustrates an extension which creates version tables for entities and stores records for each change. The given extensions generate an anonymous "history" class which represents historical versions of the target object.

Usage is illustrated via a unit test module `t`, which can be run via nose:

```
cd examples/versioning
nosetests -v
```

A fragment of example usage, using declarative:

```
from history_meta import Versioned, versioned_session

Base = declarative_base()

class SomeClass(Versioned, Base):
    __tablename__ = 'sometable'

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

    def __eq__(self, other):
        assert type(other) is SomeClass and other.id == self.id

Session = sessionmaker(bind=engine)
versioned_session(Session)

sess = Session()
sc = SomeClass(name='sc1')
```

```
sess.add(sc)
sess.commit()

sc.name = 'sc1modified'
sess.commit()

assert sc.version == 2

SomeClassHistory = SomeClass.__history_mapper__.class_

assert sess.query(SomeClassHistory).\
            filter(SomeClassHistory.version == 1).\
            all() \
            == [SomeClassHistory(version=1, name='sc1')]
```

The `V` mixin is designed to work with declarative. To use the extension with classical mappers, the `_` function can be applied:

```
from history_meta import _history_mapper

m = mapper(SomeClass, sometable)
_history_mapper(m)

SomeHistoryClass = SomeClass.__history_mapper__.class_
```

Listing of files:

- history_meta.py - Versioned mixin class and other utilities.
- test_versioning.py - Unit tests illustrating usage of the `h` module functions.

Versioning using Temporal Rows

Illustrates an extension which versions data by storing new rows for each change; that is, what would normally be an UPDATE becomes an INSERT.

Listing of files:

- versioned_map.py - A variant of the versioned_rows example. Here we store a dictionary of key/value pairs, storing the k/v's in a "vertical" fashion where each key gets a row. The value is split out into two separate datatypes, string and int - the range of datatype storage can be adjusted for individual needs.
- versioned_rows.py - Illustrates a method to intercept changes on objects, turning an UPDATE statement on a single row into an INSERT statement, so that a new row is inserted with the new data, keeping the old row intact.

Vertical Attribute Mapping

Illustrates "vertical table" mappings.

A "vertical table" refers to a technique where individual attributes of an object are stored as distinct rows in a table. The "vertical table" technique is used to persist objects which can have a varied set of attributes, at the expense of simple query control and brevity. It is commonly found in content/document management systems in order to represent user-created structures flexibly.

Two variants on the approach are given. In the second, each row references a "datatype" which contains information about the type of information stored in the attribute, such as integer, string, or date.

Example:

```
shrew = Animal(u'shrew')
shrew[u'cuteness'] = 5
shrew[u'weasel-like'] = False
shrew[u'poisonous'] = True

session.add(shrew)
session.flush()

q = (session.query(Animal).
     filter(Animal.facts.any(
       and_(AnimalFact.key == u'weasel-like',
            AnimalFact.value == True))))
print 'weasel-like animals', q.all()
```

Listing of files:

- dictlike-polymorphic.py - Mapping a polymorphic-valued vertical table as a dictionary.
- dictlike.py - Mapping a vertical table as a dictionary.

Inheritance Mapping Recipes

Basic Inheritance Mappings

Working examples of single-table, joined-table, and concrete-table inheritance as described in datamapping_inheritance.

Listing of files:

- joined.py - Joined-table (table-per-subclass) inheritance example.
- single.py - Single-table inheritance example.
- concrete.py - Concrete (table-per-class) inheritance example.

Special APIs

Attribute Instrumentation

Two examples illustrating modifications to SQLAlchemy's attribute management system.

Listing of files:

- custom_management.py - Illustrates customized class instrumentation, using the `s` extension package.
- listen_for_events.py - Illustrates how to attach events to all instrumented attributes and listen for change events.

Horizontal Sharding

A basic example of using the SQLAlchemy Sharding API. Sharding refers to horizontally scaling data across multiple databases.

The basic components of a "sharded" mapping are:

- multiple databases, each assigned a 'shard id'
- a function which can return a single shard id, given an instance to be saved; this is called "shard_chooser"
- a function which can return a list of shard ids which apply to a particular instance identifier; this is called "id_chooser". If it returns all shard ids, all shards will be searched.
- a function which can return a list of shard ids to try, given a particular Query ("query_chooser"). If it returns all shard ids, all shards will be queried and the results joined together.

In this example, four sqlite databases will store information about weather data on a database-per-continent basis. We provide example shard_chooser, id_chooser and query_chooser functions. The query_chooser illustrates inspection of the SQL expression element in order to attempt to determine a single shard being requested.

The construction of generic sharding routines is an ambitious approach to the issue of organizing instances among multiple databases. For a more plain-spoken alternative, the "distinct entity" approach is a simple method of assigning objects to different tables (and potentially database nodes) in an explicit way - described on the wiki at EntityName.

Listing of files:

- attribute_shard.py

Extending the ORM

Dogpile Caching

Illustrates how to embed dogpile.cache functionality within the `Q` object, allowing full cache control as well as the ability to pull "lazy loaded" attributes from long term cache as well.

Changed in version 0.8: The example was modernized to use dogpile.cache, replacing Beaker as the caching library in use.

In this demo, the following techniques are illustrated:

- Using custom subclasses of `Q`
- Basic technique of circumventing Query to pull from a custom cache source instead of the database.
- Rudimental caching with dogpile.cache, using "regions" which allow global control over a fixed set of configurations.
- Using custom `M` objects to configure options on a Query, including the ability to invoke the options deep within an object graph when lazy loads occur.

E.g.:

```
# query for Person objects, specifying cache
q = Session.query(Person).options(FromCache("default"))

# specify that each Person's "addresses" collection comes from
# cache too
q = q.options(RelationshipCache(Person.addresses, "default"))

# query
print q.all()
```

To run, both SQLAlchemy and dogpile.cache must be installed or on the current PYTHONPATH. The demo will create a local directory for datafiles, insert initial data, and run. Running the demo a second time will utilize the cache files already present, and exactly one SQL statement against two tables will be emitted - the displayed result however will utilize dozens of lazyloads that all pull from cache.

The demo scripts themselves, in order of complexity, are run as Python modules so that relative imports work:

```
python -m examples.dogpile_caching.helloworld

python -m examples.dogpile_caching.relationship_caching

python -m examples.dogpile_caching.advanced

python -m examples.dogpile_caching.local_session_caching
```

Listing of files:

- environment.py - Establish data / cache file paths, and configurations, bootstrap fixture data if necessary.
- caching_query.py - Represent functions and classes which allow the usage of Dogpile caching with SQLAlchemy. Introduces a query option called FromCache.
- model.py - The datamodel, which represents Person that has multiple Address objects, each with PostalCode, City, Country.
- fixture_data.py - Installs some sample data. Here we have a handful of postal codes for a few US/ Canadian cities. Then, 100 Person records are installed, each with a randomly selected postal code.
- helloworld.py - Illustrate how to load some data, and cache the results.
- relationship_caching.py - Illustrates how to add cache options on relationship endpoints, so that lazyloads load from cache.
- advanced.py - Illustrate usage of Query combined with the FromCache option, including front-end loading, cache invalidation and collection caching.
- local_session_caching.py - Grok everything so far ? This example creates a new dogpile.cache backend that will persist data in a dictionary which is local to the current session. remove() the session and the cache is gone.

PostGIS Integration

A naive example illustrating techniques to help embed PostGIS functionality.

This example was originally developed in the hopes that it would be extrapolated into a comprehensive PostGIS integration layer. We are pleased to announce that this has come to fruition as GeoAlchemy.

The example illustrates:

- a DDL extension which allows CREATE/DROP to work in conjunction with AddGeometryColumn/DropGeometryColumn
- a Geometry type, as well as a few subtypes, which convert result row values to a GIS-aware object, and also integrates with the DDL extension.
- a GIS-aware object which stores a raw geometry value and provides a factory for functions such as AsText().
- an ORM comparator which can override standard column methods on mapped objects to produce GIS operators.
- an attribute event listener that intercepts strings and converts to GeomFromText().
- a standalone operator example.

The implementation is limited to only public, well known and simple to use extension points.

E.g.:

```
print session.query(Road).filter(Road.road_geom.intersects(r1.road_geom)).all()
```

Listing of files:

- postgis.py