SQLAlchemy Core

March 31, 2021

0.1 Init

[2]:
```
import sys
!{sys.executable} -m pip install SQLAlchemy

import sqlalchemy
```

```
WARNING: You are using pip version 20.2.3; however, version 21.0.1 is available.
You should consider upgrading via the
'c:\users\vicktree\appdata\local\programs\python\python39\python.exe -m pip
install --upgrade pip' command.
WARNING: You are using pip version 20.2.3; however, version 21.0.1 is available.
You should consider upgrading via the
'c:\users\vicktree\appdata\local\programs\python\python39\python.exe -m pip
install --upgrade pip' command.
```

```
Collecting SQLAlchemy
  Downloading SQLAlchemy-1.4.4-cp39-cp39-win_amd64.whl (1.5 MB)
Collecting greenlet!=0.4.17; python_version >= "3"
  Downloading greenlet-1.0.0-cp39-cp39-win_amd64.whl (95 kB)
Installing collected packages: greenlet, SQLAlchemy
Successfully installed SQLAlchemy-1.4.4 greenlet-1.0.0
Collecting SQLAlchemy
  Downloading SQLAlchemy-1.4.4-cp39-cp39-win_amd64.whl (1.5 MB)
Collecting greenlet!=0.4.17; python_version >= "3"
  Downloading greenlet-1.0.0-cp39-cp39-win_amd64.whl (95 kB)
Installing collected packages: greenlet, SQLAlchemy
Successfully installed SQLAlchemy-1.4.4 greenlet-1.0.0
```

0.2 Schema and Types

- inorder to provide access to the underlying database, SQLALchemy needs a representation of the tables that should be present in the database
- this can be done in three ways
  - using user-defined Table objects
  - using declarative classes that represent your tables
  - interferring them from the database
- we will be using user-defined `Table` objects

- the 'Table object contain a list of typed columns and their attributes, which are associated with a common metadata container

0.2.1 Types

- there are four categories of types we can use inside of SQLAlchemy
- Generic
- SQL standard
- Vendor specific
- User defined

Table 1-1. Generic type representations

SQLAlchemy	Python	SQL
BigInteger	int	BIGINT
Boolean	bool	BOOLEAN or SMALLINT
Date	datetime.date	DATE (SQLite: STRING)
DateTime	datetime.datetime	DATETIME (SQLite: STRING)
Enum	str	ENUM or VARCHAR
Float	float or Decimal	FLOAT or REAL
Integer	int	INTEGER
Interval	datetime.timedelta	INTERVAL or DATE from epoch
LargeBinary	byte	BLOB or BYTEA
Numeric	decimal.Decimal	NUMERIC or DECIMAL
Unicode	unicode	UNICODE or VARCHAR
Text	str	CLOB or TEXT
Time	datetime.time	DATETIME

- There are standard types which we can import from `sqlchemy.dialects`
- they are usually captial letters
- `from sqlalchemy.dialects.postgresql iport JSON`
- this allows us to use things like `array_to_json`
- we can also use `VARCHAR` when working with legacy codebase

0.2.2 Metadata

- Metadata is used to tie together the database structure so it can be quickly accessed inside SQLAlchemy
- metadata is a catalog of Table objects with optional information about the engine and co-nenction
- these tables can be accessed via a dictionary, `MetaData.tables`

[2]:
```
from sqlalchemy import MetaData
metadata = MetaData()
```

0.2.3 Tables

- Tables objects are initialized in SQLAlchemy Core in supplied `MetaData` object by calling the `Table` construvtor with the table name and metadata
- any additional arguments are assumed to be column objects
- `Column` objects represent each field in the table
- Columns are constructed by calling `Column` with a name, type and then arguemnt that represent any additional `SQL` constructs and constraints

[3]:
```
from sqlalchemy import Table, Column, Integer, Numeric, String, ForeignKey

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('unit_cost', Numeric(12, 2))
)
```

- note `index(=True)` is being used to speed up queires on that column

[4]:
```
from datetime import datetime
from sqlalchemy import DateTime

user = Table('user', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_adress', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
```

```
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.
 ↪now)

)
```

0.2.4 Keys and Constraints

- keys and constraints ensure that the data meets certain requirements prior to being stored in the database
- `from sqlalchemy import PrimaryKeyConstraint, UniqueConstraint, CheckConstraints`
- the most common key type is the primary key which is used to ensure a proper relationship between two pieces of related data in different tables
- if your `primary_key=True` on multiple columns, then the keys are treated as a tuple
-
- we can add multiple columns seprated by commas to create a composite key
- we can explicitly define a key with the following code:
  - `PrimaryKeyConstraints('user_id', name='user_pk')`
- another popular constraint is the unique constraint, which ensures that no two values are duplicated in a given field
  - `UniqueConstraints('username', name='uix_username')`
  - we are insuring that each customer had a unique username to log
- another one is the check constraint, which is used to ensure that the data supplied for a column matches a set of user-defined criteria
  - `CheckConstraint('unit_cost >= 0.00', name='unit_cost_postive')`

0.2.5 indexes

- used to accelerate lookups for field values
- we create index on `cookie_name` column because we know we will be searching by that often
- we can also define an index using an explicit construction type
- multiple columns can be designated by separating them by a comma
- you can also add a keyword argument of `unique=True` require the index to be unique

[5]:
```
from sqlalchemy import Index

Index('ix_cookies_cookie_name', 'cookie _name')
```

[5]:
```
Index('ix_cookies_cookie_name', 'cookie _name')
```

- we can also create functional indexes that vary a bit by the backend database being used
- this lets you create an index for situations where you often need to query based on some unusual context
- for example if we want to select by cookie SKU and name as a joined item, such as `SKU001 Chocolate Chip`
- `Index('ix_test, mytable.c.cookie_sku, mytable.c.cookie_name)`

0.2.6 Relationships and ForeignKeyConstraints

- one way to implement a relationshp is to use `ForeignKeyConstraint`

[6]:
```
# More tables with relationships

from sqlalchemy import ForeignKey, Boolean

orders = Table('orders', metadata,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('user.user_id')),
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('order.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)
```

- in the `ForeignKey` you want to use string and not actual reference because using reference could actually cause it to fail

0.2.7 Persisting The Tables

- persisting the schema to the database is simply a matter of calling the `create_all()` method on our `metadata` instance with the engine where it should create those tables:
  - `metadata.create_all(engine)`
- better to use database migration tools like `Alembic` to handle any changes to existing tables or additional schema than to try to handcode changes directly in your application code

[7]:
```
from datetime import datetime
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
 DateTime, ForeignKey, create_engine)

metadata = MetaData()

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2))
)
```

```
users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('customer_number', Integer(), autoincrement=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.
 ↪now)
)

orders = Table('orders', metadata,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('users.user_id'))
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)

engine = create_engine('sqlite:///:memory:')
metadata.create_all(engine)
```

0.3 Working With Data via SQLAlchemy Core

0.3.1 Inserting Data

[8]:
```
ins = cookies.insert().values(
    cookie_name="chocolate chip",
    cookie_recipe_url="http://some.aweso.me/cookie/recipe.html",
    cookie_sku="CC01",
    quantity="12",
    unit_cost="0.50"
)

#print(str(ins)) shows us the actual SQL statement that will be executed
print('The SQL equlivant of our insert statement')
print(str(ins))
print('')

print('the paramaters entered into our SQL statement')
ins.compile().params
```

```
print('')

connection = engine.connect()
result = connection.execute(ins)

print('ID of the recored we just created')
result.inserted_primary_key
```

```
The SQL equlivant of our insert statement
INSERT INTO cookies (cookie_name, cookie_recipe_url, cookie_sku, quantity,
unit_cost) VALUES (:cookie_name, :cookie_recipe_url, :cookie_sku, :quantity,
:unit_cost)

the paramaters entered into our SQL statement

ID of the recored we just created
```

[8]:
```
[1]
```

- alternatively you could also insert by importing `insert`
- more adviced to just use the `Table` method insert

[9]:
```
'''from sqlalchemy import insert

ins = insert(cookies).values(
    cookie_name="plain",
    cookie_recipe_url="https://some.aweso.me/cookie/recipe.html",
    cookie_sku="CC01",
    quantity="12",
    unit_cost="0.50"
)'''
```

[9]:
```
'from sqlalchemy import insert\n\nins = insert(cookies).values(\n
cookie_name="plain",\n
cookie_recipe_url="https://some.aweso.me/cookie/recipe.html",\n
cookie_sku="CC01",\n    quantity="12",\n    unit_cost="0.50"\n)'
```

- `execute` method of the connection object can take more than just statements
- it is also possible to provide the values as keyword arguemnts to the `execute` method after our statement

[10]:
```
ins = cookies.insert()
result = connection.execute(
        ins,
        cookie_name='dark chocolate chip',
        cookie_recipe_url='https://some.aweso.me/cookie/recipe_dark.html',
        cookie_sku="CC02",
        quantity="1",
        unit_cost="0.75"
```

```
)
result.inserted_primary_key
```

[10]: [2]

- we can insert multiple objects into the table by using an array

[11]:
```
inventory_list = [
 {
    'cookie_name': 'peanut butter',
    'cookie_recipe_url': 'http://some.aweso.me/cookie/peanut.html',
    'cookie_sku': 'PB01',
    'quantity': '24',
    'unit_cost': '0.25'
 },
 {
    'cookie_name': 'oatmeal raisin',
    'cookie_recipe_url': 'http://some.okay.me/cookie/raisin.html',
    'cookie_sku': 'EWW01',
    'quantity': '100',
    'unit_cost': '1.00'
 }
]
result = connection.execute(ins, inventory_list)
```

0.3.2 Querying Data

[12]:
```
from sqlalchemy.sql import select

s = select([cookies])
rp = connection.execute(s)
results = rp.fetchall()

for cookie in results:
    print(f'{cookie} \n')
```

```
(1, 'chocolate chip', 'http://some.aweso.me/cookie/recipe.html', 'CC01', 12,
Decimal('0.50'))

(2, 'dark chocolate chip', 'https://some.aweso.me/cookie/recipe_dark.html',
'CC02', 1, Decimal('0.75'))

(3, 'peanut butter', 'http://some.aweso.me/cookie/peanut.html', 'PB01', 24,
Decimal('0.25'))

(4, 'oatmeal raisin', 'http://some.okay.me/cookie/raisin.html', 'EWW01', 100,
Decimal('1.00'))
```

```
c:\users\vicktree\appdata\local\programs\python\python38-32\lib\site-
packages\sqlalchemy\sql\sqltypes.py:661: SAWarning: Dialect sqlite+pysqlite does
*not* support Decimal objects natively, and SQLAlchemy must convert from
floating point - rounding errors and other issues may occur. Please consider
storing Decimal numbers as strings or integers on this platform for lossless
storage.
  util.warn(
```

[13]:
```
from sqlalchemy.sql import select
s = cookies.select()
rp = connection.execute(s)
result = rp.fetchall()


for cookie in results:
    print(f'{cookie} \n')
```

```
(1, 'chocolate chip', 'http://some.aweso.me/cookie/recipe.html', 'CC01', 12,
Decimal('0.50'))

(2, 'dark chocolate chip', 'https://some.aweso.me/cookie/recipe_dark.html',
'CC02', 1, Decimal('0.75'))

(3, 'peanut butter', 'http://some.aweso.me/cookie/peanut.html', 'PB01', 24,
Decimal('0.25'))

(4, 'oatmeal raisin', 'http://some.okay.me/cookie/raisin.html', 'EWW01', 100,
Decimal('1.00'))
```

0.3.3 ResultProxy

- a wrapper around a DBAPI cursor object and the goal is to make it easier to use and manipulate the results of a statement
- it could make handling query results easier by allowing access using an index, name or Column object
- all of the following statmeents result in the same thing: 'choclate chip'
- they each refrence the exact same data element in the first record of our results variable

[14]:
```
# get the first row of the resultproxy
first_row = results[0]

# access column by index
first_row[1]

# access column by name
first_row.cookie_name
```

```
# access column by Column object
first_row[cookies.c.cookie_name]
```

[14]: 'chocolate chip'

- we can also leverage the ResultProxy as an iterable and perform an action on each record returned without creatingt another variable to hold the results

[15]:
```
# print the name of each cookie in our database

rp = connection.execute(s)

for record in rp:
    print(record.cookie_name)
```

```
chocolate chip
dark chocolate chip
peanut butter
oatmeal raisin
```

- you can use the following methods to fetch results
- `first()`
  - returns the first record if there is one and closes the connection
- `fetchone()`
  - returns one row and leaves the cursor open for you to make additional fetch calls
- `scalar()`
  - returns a signle value if a query results in a single record with one column
- if you want to see the columns that are available in a result set you can use the `keys()` method to get a list of the column names

Tips for Good Production Code

- use the `first` method for getting a single record over both the `fetchone` and `scalar` methods
- use the iterable version of the `ResultProxy` over the `fetchall` and `fetchone` methods. It is more memory efficient and we tend to operate on the data one record at a time
- avoid the `fetchone` method, as it leaves connections open if you are not careful
- use the `scalar` method sparingly, as it raises errors if a query every returns more than one row and one column, which often gets missed during testing

0.3.4 Controlling The Columns in The Query

- to limit the fields that are returned from a query, we need to pass the columns we want into the `select()` method constructor as a list

[16]:
```
s = select([cookies.c.cookie_name, cookies.c.quantity])
rp = connection.execute(s)

# returns the list of columns which is [u'cookie_name', u'quantity']
print('getting the list of columns')
```

```
print(rp.keys())
print('')
# returns only the first field
print('getting the first field')
result = rp.first()
print(result)
```

```
getting the list of columns
['cookie_name', 'quantity']

getting the first field
('chocolate chip', 12)
```

0.3.5 Ordering

- if we want the list to be returned in a particular order, we can chain an `order_by()` statement to our select
- `desc()` can be used to reverse the order

[17]:
```
s = select([cookies.c.cookie_name, cookies.c.quantity])
s = s.order_by(cookies.c.quantity)
rp = connection.execute(s)

for cookie in rp:
    print(f'{cookie.quantity} - {cookie.cookie_name}')
```

```
1 - dark chocolate chip
12 - chocolate chip
24 - peanut butter
100 - oatmeal raisin
```

0.3.6 Limiting

- we have to use `limit()` method to limit

[18]:
```
s = select([cookies.c.cookie_name, cookies.c.quantity])
s = s.order_by(cookies.c.quantity)
s = s.limit(2)
rp = connection.execute(s)

print([result.cookie_name for result in rp])
```

```
['dark chocolate chip', 'chocolate chip']
```

0.3.7 Built-In SQL Functions and Labels

- two of the most common database functions are `SUM()` and `COUNT()`

[19]:
```
from sqlalchemy.sql import func
```

```
s = select([func.sum(cookies.c.quantity)])
rp = connection.execute(s)

# scalar is returning the left most column in the first record
print("returning the left most colum in the first record")
print(rp.scalar())
print("")

s2 = select([func.count(cookies.c.cookie_name)])
rp = connection.execute(s2)
record = rp.first()

# this will show us the columns in the ResultProxy
print('these are the columns in the ResultProxy')
print(record.keys())
print("")

# the column name is autogenerated and commonly `<func_name>_<position>`
print("this is the record count")
print(record.count_1)
print("")
```

```
returning the left most colum in the first record
137

these are the columns in the ResultProxy
['count_1']

this is the record count
4
```

- `label()` function helps us label better so that the column name is not annoying and cumbersome
- you dont really want to be using `count_1`

[20]:
```
s = select([func.count(cookies.c.cookie_name).label('inventory_count')])
rp = connection.execute(s)
record = rp.first()


print('these are the columns in the ResultProxy')
print(record.keys())
print("")

print(print("this is the record count"))
print(record.inventory_count)
```

```
these are the columns in the ResultProxy
```

```
['inventory_count']

this is the record count
None
4
```

0.3.8 Filtering

- filtering queries is done by adding a `where()` statement just like in SQL
- we can chain multiple `where()` clauses

[21]:
```
s = select([cookies]).where(cookies.c.cookie_name == 'chocolate chip')
rp = connection.execute(s)

record = rp.first()

#
print('a list ofcolumns and values \n')
for record in record.items():
    print(record)
    print('')
```

```
a list ofcolumns and values

('cookie_id', 1)

('cookie_name', 'chocolate chip')

('cookie_recipe_url', 'http://some.aweso.me/cookie/recipe.html')

('cookie_sku', 'CC01')

('quantity', 12)

('unit_cost', Decimal('0.50'))
```

[22]:
```
# we can also find statements that contain particular words in them
s = select([cookies]).where(cookies.c.cookie_name.like('%chocolate%'))
rp = connection.execute(s)
print('here are cookies types that begin with %chocolate% \n')
for record in rp.fetchall():
    print(record.cookie_name)
```

```
here are cookies types that begin with %chocolate%

chocolate chip
dark chocolate chip
```

0.3.9 Clause Elements

- `ClauseElements` are just an entity we use in a clause, and they are typically columns in a table
- unlike columns, ClauseElements come with many additional capabilities
- `like` method is an example of this
- there are also negation versions of those methods called `isnot()`
- instead of using these, we can just use Operators

[23]:
```
![](images/02.png)
```

```
'[]' is not recognized as an internal or external command,
operable program or batch file.
```

0.3.10 Operators

- you can use most operators
- you can also use `/+, -, *, /, %` for sting contraction

[24]:
```
s = select([cookies.c.cookie_name, 'SKU-' + cookies.c.cookie_sku])

print('showing the cookies along with their SKU \n')
for row in connection.execute(s):
    print(row)
```

```
showing the cookies along with their SKU

('chocolate chip', 'SKU-CC01')
('dark chocolate chip', 'SKU-CC02')
('peanut butter', 'SKU-PB01')
('oatmeal raisin', 'SKU-EWW01')
```

- we can also use operators to compute values

[25]:
```
# Cast() is another function that allows us to convert types
from sqlalchemy import cast

s = select([cookies.c.cookie_name,
        cast((cookies.c.quantity * cookies.c.unit_cost),
            Numeric(12,2)).label('inv_cost')])


print('showing inventory value by cookie \n')
for row in connection.execute(s):
    print(f'{row.cookie_name} - {row.inv_cost}')
```

```
showing inventory value by cookie

chocolate chip - 6.00
dark chocolate chip - 0.75
```

```
peanut butter - 6.00
oatmeal raisin - 100.00
```

0.3.11 Boolean Operators

- be careful because `&` binds more closely than `<`
- you want to use `Conjuction` instead of using `&` or `<` because it is more expressive

0.3.12 Conjuction

- when it is possible to chain multiple `where()` clauses together, it's often more readable and functional to use conjuctions to accomplish the desired effect
- conjuctions are `and_()`, `or_()`, `not_()`

[26]:
```
from sqlalchemy import and_, or_, not_

s = select([cookies]).where(
    and_(cookies.c.quantity > 23,
         cookies.c.unit_cost < 0.40
    )
)

print('using and() conjuction to show filtered data \n')
for row in connection.execute(s):
    print(row.cookie_name)
```

```
using and() conjuction to show filtered data

peanut butter
```

0.4 Updating Data

- `update` statements can be created by either the `update()` function or the `update()` method on the table being updated
- if you dont have a where clause, you can update everything

[27]:
```
from sqlalchemy import update

u = update(cookies).where(cookies.c.cookie_name == 'chocolate chip')
u = u.values(quantity=(cookies.c.quantity + 120))
result = connection.execute(u)

print('showing how many rows were updated')
print(result.rowcount, "\n")

s = select([cookies]).where(cookies.c.cookie_name == 'chocolate chip')

result = connection.execute(s).first()
```

```
print('showing the new updated results')
for key in result.keys():
    print(f'{key}: {result[key]}')
```

```
showing how many rows were updated
1

showing the new updated results
cookie_id: 1
cookie_name: chocolate chip
cookie_recipe_url: http://some.aweso.me/cookie/recipe.html
cookie_sku: CC01
quantity: 132
unit_cost: 0.50
```

0.4.1 Deleting Data

- we can either use the `delete()` function or method on the table
- unlike `insert` or `update`, delete takes no values, only optional where clause

[28]:
```
from sqlalchemy import delete

u = delete(cookies).where(cookies.c.cookie_name == "dark chocolate chip")
result = connection.execute(u)

print('how many results have been found')
print(result.rowcount)
print("")

s = select([cookies]).where(cookies.c.cookie_name == "dark chocolate chip")
result = connection.execute(s).fetchall()

print('fetching all instances of dark choclate chip')
print(len(result))
```

```
how many results have been found
1

fetching all instances of dark choclate chip
0
```

[29]:
```
# we are loading in some customer data


customer_list = [
    {
```

```
    'username': 'cookiemon',
    'email_address': 'mon@cookie.com',
    'phone': '111-111-1111',
    'password': 'password'
    },
    {
    'username': 'cakeeater',
    'email_address': 'cakeeater@cake.com',
    'phone': '222-222-2222',
    'password': 'password'
    },
    {
    'username': 'pieguy',
    'email_address': 'guy@pie.com',
    'phone': '333-333-3333',
    'password': 'password'
    }
]

ins = users.insert()
result = connection.execute(ins, customer_list)
```

[30]:
```
from sqlalchemy import insert

# enter customer data
ins = insert(orders).values(user_id=1, order_id=1)
result = connection.execute(ins)
ins = insert(line_items)
order_items = [
    {
    'order_id': 1,
    'cookie_id': 1,
    'quantity': 2,
    'extended_cost': 1.00
    },
    {
    'order_id': 1,
    'cookie_id': 3,
    'quantity': 12,
    'extended_cost': 3.00
    }
]

result = connection.execute(ins, order_items)
ins = insert(orders).values(user_id=2, order_id=2)
result = connection.execute(ins)
ins = insert(line_items)
```

```
order_items = [
    {
    'order_id': 2,
    'cookie_id': 1,
    'quantity': 24,
    'extended_cost': 12.00
    },
    {
    'order_id': 2,
    'cookie_id': 4,
    'quantity': 6,
    'extended_cost': 6.00
    }
]
result = connection.execute(ins, order_items)
```

0.4.2 Joins

- `select_from()` allows us to replace the entire `from` clause with a more specific one

[31]:
```
columns = [orders.c.order_id, users.c.username, users.c.phone,
           cookies.c.cookie_name, line_items.c.quantity,
           line_items.c.extended_cost]

cookiemon_orders = select(columns)
cookiemon_orders = cookiemon_orders.select_from(orders.join(users).join(
                       line_items).join(cookies)).where(users.c.username ==
                            'cookiemon')

result = connection.execute(cookiemon_orders).fetchall()
for row in result:
    print(row)
```

```
(1, 'cookiemon', '111-111-1111', 'chocolate chip', 2, Decimal('1.00'))
(1, 'cookiemon', '111-111-1111', 'peanut butter', 12, Decimal('3.00'))
```

- the sqlalchemy statement above looks like this

  SELECT orders.order_id, users.username, users.phone, cookies.cookie_name, line_items.quantity, line_items.extended_cost FROM users JOIN orders ON users.user_id = orders.user_id JOIN line_items ON orders.order_id = line_items.order_id JOIN cookies ON cookies.cookie_id = line_items.cookie_id WHERE users.username = :username_1

- we can also use outerjoin to select from multiple tables
- sqlalchemy knows how to join the users and order tables because of the forgein key defined in the orders table

[32]:
```
columns = [users.c.username, func.count(orders.c.order_id)]
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username)
result = connection.execute(all_orders).fetchall()

for row in result:
    print(row)
```

```
('cakeeater', 1)
('cookiemon', 1)
('pieguy', 0)
```

0.4.3 Aliases

- SQLAlchemy allows the use of aliasing with the use of `alias` function or methods

```
employee_table = Table(
    'employee', metadata,
    Column('id', Integer, primary_key=True),
    Column('manager', None, ForeignKey('employee.id')),
    Column('name', String(255)))

'''
SELECT employee.name
FROM employee, employee AS manager
WHERE employee.manager_id = manager.id
    AND manager.name = 'Fred'
'''



manager = employee_table.alias('mgr')
stmt = select([employee_table.c.name],
        and_(employee_table.c.manager_id==manager.c.id,
            manager.c.name=='Fred'))

print(smtmt)
```

```
manager = employee_table.alias()
stmt = select([employee_table.c.name],
    and_(employee_table.c.manager_id==manager.c.id,
    manager.c.name=='Fred'))
    print(stmt)

SELECT employee.name
FROM employee, employee AS employee_1
WHERE employee.manager_id = employee_1.id AND employee_1.name = ?
```

0.4.4 Grouping

- when grouping, you need one or more columns to group on and one or more columns that it makes sense to aggrate with count, sum, etc

[33]:
```
columns = [users.c.username, func.count(orders.c.order_id)]
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username)
result = connection.execute(all_orders).fetchall()

for row in result:
    print(row)
```

```
('cakeeater', 1)
('cookiemon', 1)
('pieguy', 0)
```

0.4.5 Chainig

[34]:
```
def get_orders_by_customer(cust_name):
    columns = [orders.c.order_id, users.c.username, users.c.phone,
               cookies.c.cookie_name, line_items.c.quantity,
               line_items.c.extended_cost]
    cust_orders = select(columns)

    cust_orders = cust_orders.select_from(
         users.join(orders).join(line_items).join(cookies))
    cust_orders = cust_orders.where(users.c.username == cust_name)

    result = connection.execute(cust_orders).fetchall()

    return result
get_orders_by_customer('cakeeater')
```

[34]:
```
[(2, 'cakeeater', '222-222-2222', 'chocolate chip', 24, Decimal('12.00')),
 (2, 'cakeeater', '222-222-2222', 'oatmeal raisin', 6, Decimal('6.00'))]
```

- we can also do some form of conditional chaining

```
'''
def get_orders_by_customer(cust_name, shipped=None, details=False):
    columns = [orders.c.order_id, users.c.username, users.c.phone]
    joins = users.join(orders)
    if details:
        columns.extend([cookies.c.cookie_name, line_items.c.quantity,
            line_items.c.extended_cost])
        joins = joins.join(line_items).join(cookies)
    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(joins)
```

```
    cust_orders = cust_orders.where(users.c.username == cust_name)
    if shipped is not None:
        cust_orders = cust_orders.where(orders.c.shipped == shipped)
    result = connection.execute(cust_orders).fetchall()
    return result


get_orders_by_customer('cakeeater')
get_orders_by_customer('cakeeater', details=True)
get_orders_by_customer('cakeeater', shipped=True)
get_orders_by_customer('cakeeater', shipped=False)
get_orders_by_customer('cakeeater', shipped=False, details=True)
'''
```

0.5 Raw Queries

[35]:
```
result = connection.execute("select * from orders").fetchall()
print(result)
```

```
[(1, 1), (2, 2)]
```

[36]:
```
from sqlalchemy import text

stmt = select([users]).where(text("username='cookiemon'"))
print(connection.execute(stmt).fetchall())
```

```
[(1, None, 'cookiemon', 'mon@cookie.com', '111-111-1111', 'password',
datetime.datetime(2021, 2, 22, 8, 38, 51, 536158), datetime.datetime(2021, 2,
22, 8, 38, 51, 536158))]
```

0.6 Exceptions and Transactions

- the most common errors are `AttributeErrors` and `IntegrityErrors`

[6]:
```
from datetime import datetime
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine,
                        CheckConstraint)

metadata = MetaData()

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2)),
```

```
    CheckConstraint('quantity > 0', name='quantity_positive')
)

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now,
            onupdate=datetime.now)
)

orders = Table('orders', metadata,
    Column('order_id', Integer()),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)

engine = create_engine('sqlite:///:memory:')
metadata.create_all(engine)
connection = engine.connect()
```

0.6.1 AttributeError

- occurs when you attempt to access a column on a `ResultProxy` that isn't present

[5]:
```
# this is an example of an attributeError

from sqlalchemy import select, insert

ins = insert(users).values(
    username="cookiemon",
    email_address="mon@cookie.com",
    phone="111-111-1111",
    password="password"
)
```

```
result = connection.execute(ins)
s = select([users.c.username])
results = connection.execute(s)

for result in results:
    print(result.username)
    print(result.password)
```

```
---------------------------------------------------------------------------
IntegrityError                                Traceback (most recent call last)
c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
 ↪*args)
   1275                 if not evt_handled:
-> 1276                     self.dialect.do_execute(

   1277                         cursor, statement, parameters, context

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in do_execute(self, cursor, statement, parameters, context)
    607     def do_execute(self, cursor, statement, parameters, context=None):
--> 608         cursor.execute(statement, parameters)
    609

IntegrityError: UNIQUE constraint failed: users.username

The above exception was the direct cause of the following exception:

IntegrityError                                Traceback (most recent call last)
<ipython-input-5-02115079d03f> in <module>
     10 )
     11
---> 12 result = connection.execute(ins)
     13 s = select([users.c.username])
     14 results = connection.execute(s)

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in execute(self, object_, *multiparams, **params)
   1009             )
   1010         else:
-> 1011             return meth(self, multiparams, params)
   1012
   1013     def _execute_function(self, func, multiparams, params):
```

```
c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\sql\
 ↪py in _execute_on_connection(self, connection, multiparams, params)
    296     def _execute_on_connection(self, connection, multiparams, params):
    297         if self.supports_execution:
--> 298             return connection._execute_clauseelement(self, multiparams,␣
 ↪params)
    299         else:
    300             raise exc.ObjectNotExecutableError(self)

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_clauseelement(self, elem, multiparams, params)
   1122             )
   1123
-> 1124         ret = self._execute_context(
   1125             dialect,
   1126             dialect.execution_ctx_cls._init_compiled,

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
 ↪*args)
   1314
   1315         except BaseException as e:
-> 1316             self._handle_dbapi_exception(
   1317                 e, statement, parameters, cursor, context
   1318             )

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _handle_dbapi_exception(self, e, statement, parameters, cursor, context
   1508                 util.raise_(newraise, with_traceback=exc_info[2],␣
 ↪from_=e)
   1509             elif should_wrap:
-> 1510                 util.raise_(
   1511                     sqlalchemy_exception, with_traceback=exc_info[2],␣
 ↪from_=e
   1512                 )

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\util
 ↪py in raise_(***failed resolving arguments***)
    180
    181         try:
--> 182             raise exception
    183         finally:
    184             # credit to
```

```
c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
 ↪*args)
   1274                                 break
   1275                 if not evt_handled:
-> 1276                     self.dialect.do_execute(

   1277                         cursor, statement, parameters, context
   1278                     )

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in do_execute(self, cursor, statement, parameters, context)
    606
    607     def do_execute(self, cursor, statement, parameters, context=None):
--> 608         cursor.execute(statement, parameters)
    609
    610     def do_execute_no_params(self, cursor, statement, context=None):

IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed: users.usernam e
[SQL: INSERT INTO users (username, email_address, phone, password, created_on,␣
 ↪updated_on) VALUES (?, ?, ?, ?, ?, ?)]
[parameters: ('cookiemon', 'mon@cookie.com', '111-111-1111', 'password',␣
 ↪'2021-02-22 09:31:24.772878', '2021-02-22 09:31:24.772878')]
(Background on this error at: http://sqlalche.me/e/13/gkpj)
```

0.6.2 IntegrityError

- occurs wheen we try to do something that would violate the constraints configured on a `Column` or `Table`
- this type of error is commonly encountered in cases where you require something to be unique -for example, if you attempt to create two users with the same username, an `IntegrityError` will be thrown because usernames in our `users` table must be unique

[9]:
```
s = select([users.c.username])
connection.execute(s).fetchall()

[(u'cookiemon',)]

ins = insert(users).values(
    username="cookiemon",
    email_address="damon@cookie.com",
    phone="111-111-1111",
    password="password"
)
```

```
result = connection.execute(ins)
```

```
---------------------------------------------------------------------------
IntegrityError                                Traceback (most recent call last)
c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s␣lalchemy\engi
 ↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
 ↪*args)
   1275                 if not evt_handled:
-> 1276                     self.dialect.do_execute(
   1277                         cursor, statement, parameters, context

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s␣lalchemy\engi
 ↪py in do_execute(self, cursor, statement, parameters, context)
    607     def do_execute(self, cursor, statement, parameters, context=None):
--> 608         cursor.execute(statement, parameters)
    609

IntegrityError: UNIQUE constraint failed: users.username

The above exception was the direct cause of the following exception:

IntegrityError                                Traceback (most recent call last)
<ipython-input-9-db414867bc94> in <module>
     11 )
     12
---> 13 result = connection.execute(ins)

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s␣lalchemy\engi
 ↪py in execute(self, object_, *multiparams, **params)
   1009             )
   1010         else:
-> 1011             return meth(self, multiparams, params)
   1012
   1013     def _execute_function(self, func, multiparams, params):

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s␣lalchemy\sql\
 ↪py in _execute_on_connection(self, connection, multiparams, params)
    296     def _execute_on_connection(self, connection, multiparams, params):
    297         if self.supports_execution:
--> 298             return connection._execute_clauseelement(self, multiparams,␣
 ↪params)
    299         else:
    300             raise exc.ObjectNotExecutableError(self)
```

```
c:
↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
↪py in _execute_clauseelement(self, elem, multiparams, params)
   1122             )
   1123
-> 1124         ret = self._execute_context(

   1125             dialect,
   1126             dialect.execution_ctx_cls._init_compiled,


c:
↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
↪*args)
   1314
   1315         except BaseException as e:
-> 1316             self._handle_dbapi_exception(

   1317                 e, statement, parameters, cursor, context
   1318             )


c:
↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
↪py in _handle_dbapi_exception(self, e, statement, parameters, cursor, context
   1508                 util.raise_(newraise, with_traceback=exc_info[2],␣
↪from_=e)
   1509             elif should_wrap:
-> 1510                 util.raise_(

   1511                     sqlalchemy_exception, with_traceback=exc_info[2],␣
↪from_=e
   1512                 )


c:
↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\util
↪py in raise_(***failed resolving arguments***)
    180
    181         try:
--> 182             raise exception
    183         finally:
    184             # credit to


c:
↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
↪*args)
   1274                             break
   1275                 if not evt_handled:
-> 1276                     self.dialect.do_execute(

   1277                         cursor, statement, parameters, context
   1278                     )
```

```
c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in do_execute(self, cursor, statement, parameters, context)
    606
    607     def do_execute(self, cursor, statement, parameters, context=None):
--> 608         cursor.execute(statement, parameters)
    609
    610     def do_execute_no_params(self, cursor, statement, context=None):

IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed: users.usernam e
[SQL: INSERT INTO users (username, email_address, phone, password, created_on,␣
 ↪updated_on) VALUES (?, ?, ?, ?, ?, ?)]
[parameters: ('cookiemon', 'damon@cookie.com', '111-111-1111', 'password',␣
 ↪'2021-02-22 09:34:48.127762', '2021-02-22 09:34:48.127762')]
(Background on this error at: http://sqlalche.me/e/13/gkpj)
```

0.6.3 Handling Errors

- wrap as little code as possible in a `try/except` block

[12]:
```
from sqlalchemy.exc import IntegrityError
ins = insert(users).values(
    username="cookiemon",
    email_address="damon@cookie.com",
    phone="111-111-1111",
    password="password"
)

try:
    result = connection.execute(ins)
except IntegrityError as error:
    print(error.orig.message, error.params)
```

```
---------------------------------------------------------------------------
IntegrityError                             Traceback (most recent call last)
c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
 ↪*args)
   1275                 if not evt_handled:
-> 1276                     self.dialect.do_execute(
   1277                         cursor, statement, parameters, context

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in do_execute(self, cursor, statement, parameters, context)
```

```
    607     def do_execute(self, cursor, statement, parameters, context=None):
--> 608         cursor.execute(statement, parameters)
    609

IntegrityError: UNIQUE constraint failed: users.username

The above exception was the direct cause of the following exception:

IntegrityError                            Traceback (most recent call last)
<ipython-input-12-b821eef8ec75> in <module>
      9 try:
---> 10     result = connection.execute(ins)
     11 except IntegrityError as error:

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in execute(self, object_, *multiparams, **params)
   1010         else:
-> 1011             return meth(self, multiparams, params)
   1012

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\sql\
 ↪py in _execute_on_connection(self, connection, multiparams, params)
    297         if self.supports_execution:
--> 298             return connection._execute_clauseelement(self, multiparams,
 ↪params)
    299         else:

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_clauseelement(self, elem, multiparams, params)
   1123
-> 1124         ret = self._execute_context(
   1125             dialect,

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
 ↪*args)
   1315         except BaseException as e:
-> 1316             self._handle_dbapi_exception(
   1317                 e, statement, parameters, cursor, context

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _handle_dbapi_exception(self, e, statement, parameters, cursor, context
   1509             elif should_wrap:
```

```
-> 1510                 util.raise_(

   1511                     sqlalchemy_exception, with_traceback=exc_info[2],␣
 ↪from_=e

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\uti
 ↪py in raise_(***failed resolving arguments***)
    181         try:
--> 182             raise exception
    183         finally:

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in _execute_context(self, dialect, constructor, statement, parameters,␣
 ↪*args)
   1275                 if not evt_handled:
-> 1276                     self.dialect.do_execute(

   1277                         cursor, statement, parameters, context

c:
 ↪\users\vicktree\appdata\local\programs\python\python38-32\lib\site-packages\s lalchemy\engi
 ↪py in do_execute(self, cursor, statement, parameters, context)
    607     def do_execute(self, cursor, statement, parameters, context=None):
--> 608         cursor.execute(statement, parameters)
    609

IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed: users.usernam e
[SQL: INSERT INTO users (username, email_address, phone, password, created_on,␣
 ↪updated_on) VALUES (?, ?, ?, ?, ?, ?)]
[parameters: ('cookiemon', 'damon@cookie.com', '111-111-1111', 'password',␣
 ↪'2021-02-22 09:38:06.519785', '2021-02-22 09:38:06.519785')]
(Background on this error at: http://sqlalche.me/e/13/gkpj)

During handling of the above exception, another exception occurred:

AttributeError                            Traceback (most recent call last)
<ipython-input-12-b821eef8ec75> in <module>
     10     result = connection.execute(ins)
     11 except IntegrityError as error:
---> 12     print(error.orig.message, error.params)

AttributeError: 'IntegrityError' object has no attribute 'message'
```

0.6.4 Transactions

- transactions are a way to ensure that multiple database statements succeed or fail as a group
- when we start a transaction, we record the current state of our databas

- then we can execute multiple SQL statements
- if all the SQL statements in the transaction succeed, the database continues on normally and we discard the prior database state

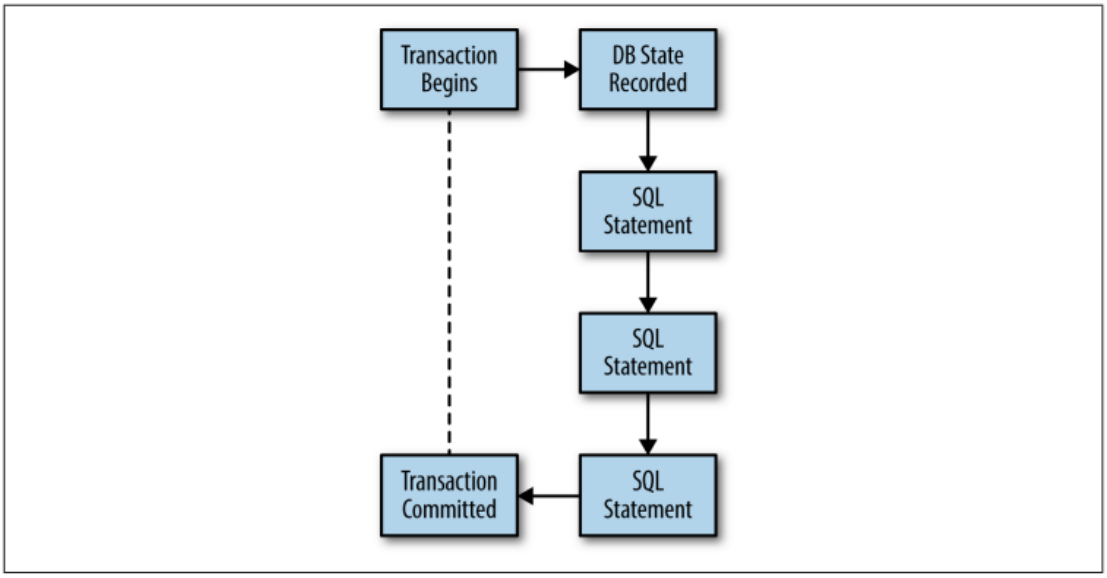

Figure 3-1. Successful transaction flow

- howerver if one or more of these statements fail,
- we can catch that error and use the prior state to roll back any statements that succeedded

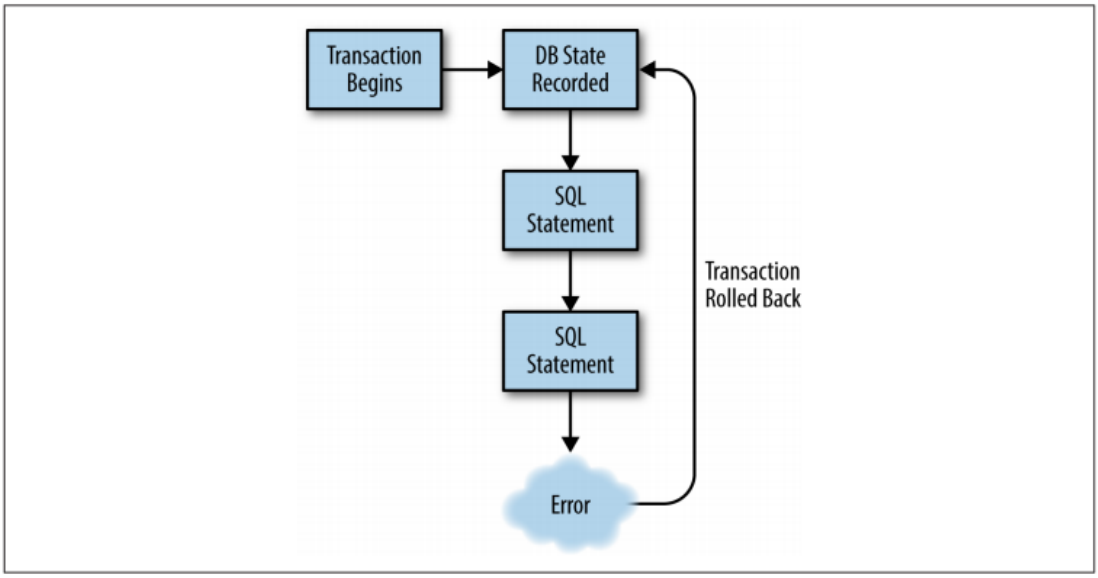

Figure 3-2. Failed transaction flow

[3]:
```
# Setting up the Transaction enviorment

from datetime import datetime
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
```

```
                        DateTime, ForeignKey, Boolean, create_engine,
                        CheckConstraint)

metadata = MetaData()

cookies = Table('cookies', metadata,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2)),

    # Note the check we aee doing here, before we actually send out shipment
    # cookies, we need to check that we have that quantity
    CheckConstraint('quantity >= 0', name='quantity_positive')
)

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False),
    Column('phone', String(20), nullable=False),
    Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now,
           onupdate=datetime.now)
)

orders = Table('orders', metadata,
    Column('order_id', Integer()),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadata,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey('cookies.cookie_id')),
    Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)

engine = create_engine('sqlite:///:memory:')
metadata.create_all(engine)
connection = engine.connect()
```

```
from sqlalchemy import select, insert, update

ins = insert(users).values(
    username="cookiemon",
    email_address="mon@cookie.com",
    phone="111-111-1111",
    password="password"
)

result = connection.execute(ins)
ins = cookies.insert()

inventory_list = [
    {
        'cookie_name': 'chocolate chip',
        'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe.html',
        'cookie_sku': 'CC01',
        'quantity': '12',
        'unit_cost': '0.50'
    },
    {
        'cookie_name': 'dark chocolate chip',
        'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe_dark.html',
        'cookie_sku': 'CC02',
        'quantity': '1',
        'unit_cost': '0.75'
    }
]

result = connection.execute(ins, inventory_list)

ins = insert(orders).values(user_id=1, order_id='1')
result = connection.execute(ins)
ins = insert(line_items)
order_items = [
     {
         'order_id': 1,
         'cookie_id': 1,
         'quantity': 9,
         'extended_cost': 4.50
     }
]
result = connection.execute(ins, order_items)

ins = insert(orders).values(user_id=1, order_id='2')
result = connection.execute(ins)
ins = insert(line_items)
```

```
order_items = [
     {
         'order_id': 2,
         'cookie_id': 1,
         'quantity': 4,
         'extended_cost': 1.50
     },
     {
         'order_id': 2,
         'cookie_id': 2,
         'quantity': 1,
         'extended_cost': 4.50
     }
]
result = connection.execute(ins, order_items)
```

[4]:
```
def ship_it(order_id):
    s = select([line_items.c.cookie_id, line_items.c.quantity])
    s = s.where(line_items.c.order_id == order_id)
    cookies_to_ship = connection.execute(s)
    for cookie in cookies_to_ship:
        u = update(cookies).where(cookies.c.cookie_id==cookie.cookie_id)
        u = u.values(quantity = cookies.c.quantity - cookie.quantity)
        result = connection.execute(u)
    u = update(orders).where(orders.c.order_id == order_id)
    u = u.values(shipped=True)
    result = connection.execute(u)
    print("Shipped order ID: {}".format(order_id))
```

[5]:
```
# running ship_it on the first order
ship_it(1)
s = select([cookies.c.cookie_name, cookies.c.quantity])
connection.execute(s).fetchall()
```

```
Shipped order ID: 1
```

[5]:
```
[('chocolate chip', 3), ('dark chocolate chip', 1)]
```

- if we tried to run the second order, we would get IntegrityError because we would not have enough chocolate chip cookies
- the problem is howerver, it did change the database value for one of the cookies before throwing an exception
- we need to use a `transaction`

[7]:
```
from sqlalchemy.exc import IntegrityError

def ship_it(order_id):
    s = select([line_items.c.cookie_id, line_items.c.quantity])
```

```
    s = s.where(line_items.c.order_id == order_id)
    transaction = connection.begin()
    cookies_to_ship = conenction.execute(s).fetchall()

    try:
        for cookie in cookies_to_ship:
            u = update(cookies).where(cookies.c.cookie_id == cookie.cookie_id)
            u = u.values(quantity == cookies.c.quantity - cookie.quantity)
            result = connection.execute(u)
        u = update(orders).where(orders.c.order_id == order_id)
        u = u.values(shipped=True)
        print("Shipped order ID: {}".format(order_id))
        transaction.commit()
    except IntegrityError as error:
        transaction.rollback()
        print(error)
```

- we import `IntegrityError` so we can handle its exception
- we start the tranasction by using `begin()`
- we use `commit()` to commit the transactions if no error occurs
- we use `rollback()` incase an error occurs

0.7 Testing

0.7.1 Testing with a Database

- we will have an `app.py` for the application logic and an `db.py` file that contains our database tables and connections
- in the `db.py` file, we have our database set up via the `DataAccessLayer` class
- the class is used to initialize a database cehema and connect it to an engine
- the `DataAccessLayer` class is initialized without an engine and a connection in the `dal` variable

[5]:
```
# db.py

from datetime import datetime
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric,
                        String, DateTime, ForeignKey, Boolean, create_engine)

class DataAccessLayer:
    connection = None
    engine = None
    conn_string = None
    metadata = MetaData()
    cookies = Table('cookies',
                    metadata,
                    Column('cookie_id', Integer(), primary_key=True),
                    Column('cookie_name', String(50), index=True),
```

```
                    Column('cookie_recipe_url', String(255)),
                    Column('cookie_sku', String(55)),
                    Column('quantity', Integer()),
                    Column('unit_cost', Numeric(12, 2))
                   )

'''provides a way to initialize a connection with a
    specific connection string like a factory
'''
def db_init(self, conn_string):
    self.engine = create_engine(conn_string or self.conn_string)
    self.metadata.create_all(self.engine)
    self.connection = self.engine.connect()

'''
provides an instance of the DataAccess Layer that can be
imported throughout our application
'''
dal = DataAccessLayer()
```

[8]:
```
# app.py

# this is our DataAccessLayer instance from the db.py file
#from db import dal
from sqlalchemy.sql import select

def get_orders_by_customers(cust_name, shipped=None, details=False):
    columns = [dal.orders.c.order_id, dal.users.c.username, dal.users.c.phone]

    # because our tables are inside of the dal object, we access them from here
    joins = dal.users.join(dal.orders)


    if details:
        columns.extend([dal.cookies.c.cookie_name,
                        dal.line_items.c.quantity,
                        dal.line_items.c.extended_cost])
    joins = joins.joins(dal.line_items).join(dal.cookies)

    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(joins).where(
    dal.users.c.username == cust_name)

    if shipped is not None:
        cust_orders = cust_orders.where(dal.orders.c.shipped == shipped)

    return dal.connection.execute(cust_orders).fetchall()
```

- notice in the file above we have three inputs `cust_name`, `shipped` and `details`
  - `cust_name` can be blank, a string containing the name of a valid customer, or a string that does not contain the name of a valid customeer
  - `shipped` can be None, True, False
  - `details` can be True or False
- inorder to test all possible combinations, we need 12 (3 * 2 * 2) tests

[13]:
```
import unittest

# unittest requires test classes inherited from unittest.TestCase
class TestApp(unittest.TestCase):

    # setupClass method is run once for the entire test class
    @classmethod
    def setUpClass(cls):

        # a connection is initialized for an in-memory db
        dal._db_init('sqlite:///:memory:')

        # this function fills database with dummy data
        # prep_db()
```

- below we are loading in data
- note that each method begins with a `test_` in its name
- also note that we are using an `assertEqual` keyword

[12]:
```
def test_orders_by_customer_blank(self):
    results = get_orders_by_customer('')
    self.assertEqual(results, [])

def test_orders_by_customer_blank_shipped(self):
    results = get_orders_by_customer('', True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped(self):
    results = get_orders_by_customer('', False)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_details(self):
    results = get_orders_by_customer('', details=True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_shipped_details(self):
    results = get_orders_by_customer('', True, True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped_details(self):
    results = get_orders_by_customer('', False, True)
```

```
    self.assertEqual(results, [])
```

- run the tests above with `python -m unittest test_app`

0.7.2 Mocking

- Mocking is a powerful tool to use when you have a test environment where creating a test database doesnt make sense or simply isnt feasible
- it could be useful to mock out the `SQLAlchemy` code to return the value you want
- this allows you to focus on the surrounding logic

[14]:
```
import sys
!{sys.executable} -m pip install mock

import mock
```

```
Collecting mock
  Downloading mock-4.0.3-py3-none-any.whl (28 kB)
Installing collected packages: mock
Successfully installed mock-4.0.3
```

```
WARNING: You are using pip version 20.2.3; however, version 21.0.1 is available.
You should consider upgrading via the
'c:\users\vicktree\appdata\local\programs\python\python39\python.exe -m pip
install --upgrade pip' command.
```

- Mock has a patch function that will let us replace a given object in a Python file with a `MagicMock` that we can control from our test
- a `MagicMock` is a special type of python object that tracks how it is used and allows us to define how it behaves based on how it is used

- below we are going to use the `patch` method as a decorator to replace the connection part of the `dal` object
- because the `dal` object is imported, we will need to patch it inside the `app` module
- this will get passed into test function as an argument
- now that we have mock object, we can set a return value for the `execute` method, which in this case should be nothing but a chained `fetchall` method whos return value is the data we want to test with

[17]:
```
from unittest import mock
from decimal import Decimal
import mock

class TestApp(unittest.TestCase):
    cookie_orders = [(u'wlk001', u'cookiemon', u'111-111-1111')]
    cookie_details = [
        (u'wlk001', u'cookiemon', u'111-111-1111',
        u'dark chocolate chip', 2, Decimal('1.00')),
        (u'wlk001', u'cookiemon', u'111-111-1111',
        u'oatmeal raisin', 12, Decimal('3.00'))
```

```
]

'''
patching dal.connection in the app module with a mock
that patch is passed into the test function as `mock_conn`
'''
@mock.patch('app.dal.connection')
def test_orders_by_customer(self, mock_conn):
    '''
    we set the return value of the execute method to the chained returne
    value of the fetchall method, which we set to self.cookie_order
    '''
    mock_conn.execute.return_value.fetchall.return_value = self.cookie_orders

    '''
    now we call the test function where the `dal. conenction will be mocked
    and return the value we set in the prior step
    '''
    result = get_orders_by_customer('cookiemon')
    self.assertEqual(result, self.cookie_orders)
```

- below we will use the `mock.patch` decorator and mock out the `select` object
- we will have to mock out all the chained query element return values, which in this query are the `select`, `select_from` and `where` clauses

[ ]:
```
'''
mocking out the select method, as it starts the query chain
the decorators are passed into the function in order
'''
@mock.patch('app.select')
@mock.patch('app.dal.connection')
def test_orders_by_customer_blank(self, mock_conn, mock_select):

    '''
    we have to mock the return value for all parts of the chained query
    '''
    mock_select.return_value.select_from.return_value \
        where.return_value = ""

    '''
    we still need to mock the connection or the app module SQLAlchemy
    code would try to make the query
    '''
    mock_conn.execute.return_value.fetchall.return_value = []
    self.assertEqual(results, [])
```

0.8 Reflections

- Reflection allows us to handle an existing database with `SQLAlchemy` without the need to re-create all the schema in Python
- Reflection is a technique that allows us to populate a SQLAlchemy object from an existing database
- instead of defining columns by hand, we are going to use the autoload and autoload_with keyword argument this will reflect
- this will reflect the schema information into the `metadata` object and store a refrence to the table in the `artist` variable
-
- you have to be careful about reflecting because what can happen is that if you reflect one table and it relies on another table which has not been reflected, then `SQLAlchemy` will drop that

  Note, the code below will not work because you need the `chinook` database in the sqlite directory

0.8.1 Reflecting Individual Tables

[25]:
```
def reflect_table(self):
    from sqlalchemy import MetaData, create_engine
    metadata = MetaData()

    # you need to have the chinook database which comes with a bunch of data
    engine = create_engine('sqlite://Chinook_Sqlite.sqlite')

    artist = Table('Artist', metadata, autoload=True, autoload_with=engine)

    # this code fetches 10 artists
    artist.columns.keys()
    from sqlalchemy import select
    s = Select([artist]).limit(10)
    engine.execute(s).fetchall()

    album = Table('Album', metadata, autoload=True, autoload_with=engine)

    # this lets you view the meta data
    metadata.tables['album']
    '''
    above statement prints:

    Table('album',
     MetaData(bind=None),
     Column('AlbumId', INTEGER(), table=<album>, primary_key=True,␣
↪nullable=False),
     Column('Title', NVARCHAR(length=160), table=<album>, nullable=False),
```

```
 Column('ArtistId', INTEGER(), table=<album>, nullable=False),
 schema=None)
)
'''
# check if reflection occured
album.forein_keys

# getting back dropped relationships
from sqlalchemy import ForeignKeyConstrain
album.append_constraint(
    ForeignKeyConstraint(['ArtistId'], ['artist.ArtistId'])
)

# this lets you view the meta data
metadata.tables['album']

# we can see database relations by using the `str` keyword
str(artist.join(album))
'''
'artist JOIN album ON artist."ArtistId" = album."ArtistId"'
'''
```

0.8.2 Reflecting a Whole Database

- inorder to reflect a whole database, we can use the reflect method on the metadata object
- the reflect method will scan everything available on the engine supplied and reflect everything it can

[26]:
```
def reflect_db():
    from sqlalchemy import MetaData, create_engine
    metadata = MetaData()

    # you need to have the chinook database which comes with a bunch of data
    engine = create_engine('sqlite://Chinook_Sqlite.sqlite')

    metadata.reflect(bind=engine)

    # check if database reflected correctly
    metadata.tables.keys()

    '''
    dict_keys(['InvoiceLine', 'Employee', 'Invoice', 'album', 'Genre',
        'PlaylistTrack', 'Album', 'Customer', 'MediaType', 'Artist',
         'Track', 'artist', 'Playlist'])
    '''
```

0.8.3 Query Building with Reflected Objects

- we need a way to refer to the test of the tables that were reflected when we reflected the entire database
- we will need a way to refer to them in our query
- we can do this by assigning them to a variable from the `tables` attribute of the `metadata`

[28]:
```
def build_reflected_query():

    # establish a variable to be refrence to the table
    playlist = metadata.tables['Playlist']
    from sqlalchemy import select

    # use the variable in the query
    s = select([playlist]).limit(10)
    engine.execute(s).fetchall()
```

- by assigning the reflected tables we want to use into a variable, we can use them as we normall used