MassonNn

ОТ ЧАЙНИКА К АЛХИМИКУ

перевод и дополнение оф. туториала

Предисловие

Представляю Вам свой перевод официального туториала SQLAlchemy, который носит нескромное название: “От чайника к алхимику”. В ходе перевода мне пришлось внести немалое количество правок и дополнений, поэтому перед вами находится от части независимый текст, который можно и нужно рассматривать в отрыве от оригинального. Также я решил оформить данный туториал в виде книги.

> SQLAlchemy Unified Tutorial — SQLAlchemy 2.0 Documentation
>
> https://docs.sqlalchemy.org/en/20/tutorial/index.html

Оригинальный туториал на английском языке

Официальный туториал распространяется бесплатно и общедоступно и поэтому **данный перевод распространяется также**. По ходу работу я заметил, что в нем было немало повторений, не всегда сохраняется нить повествования, и в конце концов был наполнен сложными для понимания словесными конструкциями и выражениями. Я постарался сгладить углы и максимально структурировал все рассуждения авторов, при этом удалив множественные повторения, лишние фразы и многое другое. Но, к сожалению, ничто не идеально, по ходу перевода, наоборот, текст становился местами сложнее. Возникло немало слов, которые сложно переводятся на русский (например, Mapping, Declarative Base и др.), поэтому для них были использованы наиболее подходящие слова на русском: для Mapping было выбрано “ассоциативный массив”, а для Declarative Base - “декларативный базис”. Использованием англицизмов я постарался пренебречь настолько насколько это было возможно.

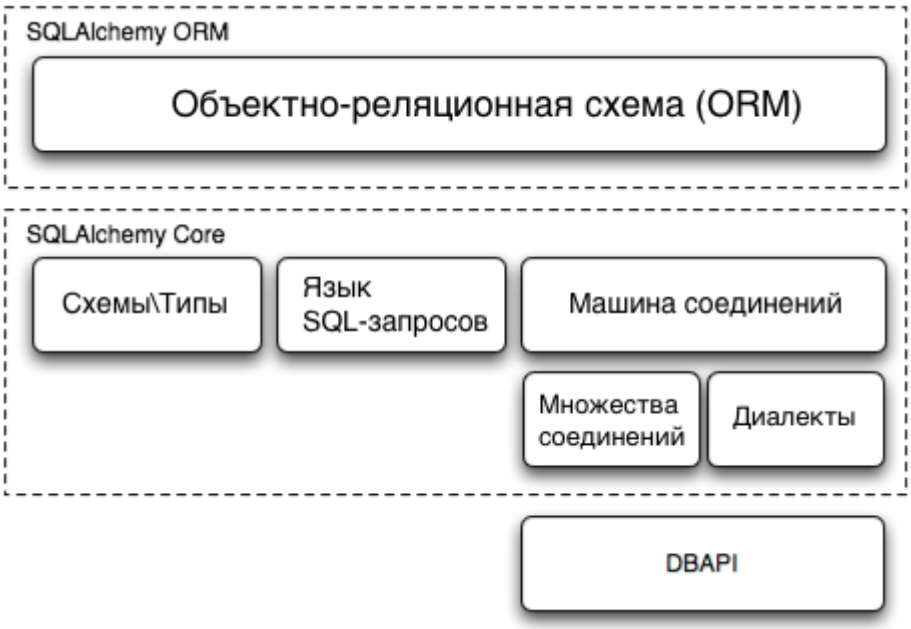
Надеюсь, книга поможет многим разработчикам в освоении такого важного и мощного инструмента как SQLAlchemy, ведь сам туториал написан непосредственно ее разработчиками и охватывает почти все аспекты алхимии. Данный текст будет содержать огромнейшее количество деталей, так что по окончании его прочтения вы можете гордо назвать себя алхимиками.

И в конце необходимо сказать, что данный перевод основан на старой версии 1.4, в новой версии 2.0 (полностью совместимой с 1.4) ORM использует похожий на Core метод создания запросов, в том числе с получением объектов

в виде select() запроса (а не query() как это было в прошлых версиях). Таким образом соединения Core и сессии в ORM в настоящий момент эквивалентны и вы без труда сможете понять принципы работы в 2.0 после освоения 1.4.

1. Введение

SQLAlchemy - это всеобъемлющий набор инструментов для работы с базами данных в рамках языка программирования Python. Она имеет несколько различных сфер функциональности, которые могут быть использованы как отдельно, так и совместно. Самые главные её компоненты изображены ниже:



На схеме две наиболее важные части SQLAlchemy объединены пунктиром в ORM и Core. Core (или ядро) содержит собственный язык SQL запросов, осуществляет поддержку различных диалектов SQL, а также осуществляет общую интеграцию с самой базой данных, в том числе контролирует соединения с ней.

Язык SQL выражений алхимии - это полноценная система, независимая от ORM, которая предоставлят возможность построения реальных SQL запросов к базам данных с помощью Python объектов и функций внутри конкретной транзакции, возвращающей результат. Вставки (Insert), обновления (Updates), и удаления (Deletes), так называемые операции DML (Data Manipulation Language) достигаются с помощью проверки объектов языка SQL алхимии, которые представлены этими выражениями, а также словарями, которые

хранят в себе параметры, необходимые для использования при выполнении каждого выражения.

ORM работает поверх ядра и предоставляет возможность работы с объектами-моделями, связанными со схемами в базе данных. При использовании ORM, SQL выражения создаются практически также, как и в Core, однако задачу DML, которая в данном случае уже представляет собой манипуляции с объектами бизнес-логики, автоматически решает использование паттерна, называемого “объединение работ” (unit of work), который транслирует изменения в состоянии объектов ORM в Insert/Update/Delete конструкции. Выражения SELECT также реализуются с применением объектов ORM, об этом мы подробнее поговорим в главе “Работа с данными”.

Как бы вы ни работали с алхимией: только используя Core или ORM - язык SQL выражений всегда будет представлять собой схематичное отображение базы данных. Разница лишь в том, что в Core он ориентирован на операции, а в ORM на объекты.

2. Соединения и машина соединений

Начало любого приложения с использованием SQLAlchemy начинается с объекта, называемого **машиной соединений** (англ. Engine). Этот объект представляет собой центральный компонент, осуществлящий соединения к отдельным базам данных, предоставляя также фабрику соединений (или пул соединений, англ. connection pool) для работы с ними. Машина является фактически глобальным объектом, создаваемым единожды с заданными настройками для подключения к базе данных, которые чаще всего представляют собой строчку URL.

Для данного туториала мы будем использовать SQLite базу данных, так как это простой способ показать и протестировать простые вещи, без необходимости поднятия полноценного сервера. Машина соединения создается при помощи функции `create_engine()`, при этом если передать в качестве ее параметра `future=True`, то мы получим полный доступ к использованию нового стиля SQLAlchemy 2.0:

```
from sqlalchemy import create_engine
engine = create_engine("sqlite+pysqlite:///:memory:", echo=True, future=True)
```

Главный аргумент для функции `create_engine` это строчка URL, выше мы передали строчку для подключения к SQLite базе данных прямо в оперативной памяти нашего компьютера (то есть даже без создания файла бд). Эта строчка передает три важных факта:

1. С каким типом СУБД мы будем иметь дело? В данном случае это sqlite, который передает SQLAlchemy информацию о том, что необходимо использовать соответствующий **диалект**.
2. Какой DBAPI мы будем использовать? Python DBAPI это сторонний драйвер, который позволяет алхимии взаимодействовать с отдельными базами данных. В данном случае мы используем pysqlite, который в Python3 заменен на sqlite3, входящим в стандартную библиотеку. Если это не будет указано, то SQLAlchemy сам решит какой DBAPI использовать.

3. И в конце концов, где мы собираемся эту базу данных располагать. В данном случае URL включает фразу `/:memory:`, которая говорит sqlite3, что мы будем работать с in-memory-only базой данных.

> ⚠️ Машина соединений, которая возвращается при исполнении функции create_engine(), на самом деле не пытается сразу же подключиться к базе данных по переданному URL, и сделает это лишь при первом обращении к базе данных. Такой паттерн поведения называют ленивой инициализацией (англ. lazy initialization).

Обратите внимание, что мы также передали параметр `echo=True`, который приказывает нашей машине сообщать о всех SQL запросах в логер Python, который в свою очередь будет передавать их в системный вывод (условно консоль). По большому счету, данный параметр всего лишь более простой путь настроить логирование и это может быть очень полезно в некоторых ситуациях.

3. Работа с транзакциями и DBAPI

Когда мы уже настроили объект машины соединений, мы готовы идти дальше. Однако нам может потребоваться углубиться в основы его операций, и в том числе **соединения** (Connection) и **результата обработки запроса** (Result). Нам также необходимо понять как устроена фасада (паттерн объединяющего класса) в ORM, соединяющая эти объекты (соединение и результат) и известная как **сессия**.

> 🌐 При использовании ORM, машина соединений полностью контролируется более высокой абстракцией - сессией. В современной алхимии сессия является менеджером транзакции и выполнения SQL скриптов, подобно тому, как это происходит с соединением (Connection) в Core. Поэтому хотя эта глава предназначена больше для Core, все ее концепции применимы и для ORM. Разница между соединением (Connection) и сессией (Session) будет показана в конце этой главы.

Чтобы познакомиться с основным инструментом алхимии - языком выражений SQL, мы начнем его использование с наиболее простой его конструкции, называемой `text()`, и которая позволяет нам передавать выражения SQL напрямую в виде текста. Остается сказать, что такая практика на сегодняшний день в алхимии используются скорее как исключение, нежели чем правило, хотя эта возможность и остается доступной.

Создание соединения

Единственная цель машины соединений это предоставлять множество соединений к базе данных. При работе с Core напрямую, объект соединения (Connection) представляет полностью всё взаимодействие с ней. При этом нам бы хотелось всегда ограничивать область использования этого объекта до конкретного контекста и лучший способ это сделать - использовать контекстные менеджеры Python, контролируемые ключевым словом `with`. Ниже мы покажем некий Hello World скрипт с использованием текстового SQL:

```
from sqlalchemy import text
with engine.connect() as conn:
result = conn.execute(text("select 'hello world'"))
print(result.all())
```

```
BEGIN (implicit)
select 'hello world'
[...] ()
[('hello world',)]
ROLLBACK
```

Как мы видим в приведенных примерах, контекстный менеджер предоставляет объект соединения, а также обрамляет все операции в рамки одной транзакции. Стандартное поведение Python DBAPI подразумевает, что транзакция всегда активна, когда же контекст с соединением заканчивается отправляется команда ROLLBACK, регистрирующая конец транзакции. Обратите внимание, что по умолчанию транзакции не завершаются автоматически, если вы хотите закрыть ее и выполнить соответствующий SQL скрипт, то вам необходимо вызвать Connection.commit(), либо использовать autocommit, который будет обсужден позже.

Результат выполнения SELECT был также возвращен в качестве объекта, называемого Result (будет также раскрыт подробнее позднее), однако имейте ввиду, что не стоит вытягивать этот объект за пределы контекста соединения.

Сохранение изменений (Commit)

Вы только что узнали, что DBAPI соединения не сохраняют изменения автоматически. Но что, если мы хотим передать определенную информацию и сохранить ее? Мы можем можем расширить предыдущий пример и добавить в него создание таблицы и вставки определенных данных:

```
with engine.connect() as conn:
     conn.execute(text("CREATE TABLE some_table (x int, y int)"))
     conn.execute(
         text("INSERT INTO some_table (x, y) VALUES (:x, :y)"),
         [{"x": 1, "y": 1}, {"x": 2, "y": 4}],
     )
     conn.commit()
```

```
BEGIN (implicit)
CREATE TABLE some_table (x int, y int)
```

```
[...] ()
<sqlalchemy.engine.cursor.CursorResult object at 0x...>
INSERT INTO some_table (x, y) VALUES (?, ?)
[...] ((1, 1), (2, 4))
<sqlalchemy.engine.cursor.CursorResult object at 0x...>
COMMIT
```

Итак, мы выполнили две SQL команды, которые вообще говоря транзакциональны: “CREATE TABLE” и “INSERT”, которые также были параметризованы с помощью соответствующего синтаксиса (о параметризации подробнее поговорим позже). Так как мы хотим, чтобы результат нашей работы не потерялся, то мы должны сохранить (за-commit-тить) наши результаты. В данном примере мы вызываем соответствующий метод `.comit()`, который соответственно закрывает транзакцию, но в отличие от ROLLBACK, не теряет данные, а записывает их в БД. Иногда такой стиль работы с транзакциями называется “сохранение на ходу”.

Однако существует и другой стиль, который как мы увидим будет сильнее связан с контекстом в Python. Для этого стиля сохранения мы будем использовать Engine.begin(), который не только будет открывать соединение, подобно Engine.connect(), но и сохранять изменения по выходу из контекста:

```
with engine.begin() as conn:
     conn.execute(
         text("INSERT INTO some_table (x, y) VALUES (:x, :y)"),
         [{"x": 6, "y": 8}, {"x": 9, "y": 10}],
     )
```

```
BEGIN (implicit)
INSERT INTO some_table (x, y) VALUES (?, ?)
[...] ((6, 8), (9, 10))
<sqlalchemy.engine.cursor.CursorResult object at 0x...>
COMMIT
```

Как мы видим в данном случае мы не вызываем `.commit()` самостоятельно, но тем не менее по закрытии блока with это происходит автоматически.

Данный способ сохранения удобен, но в ходе этого туториала мы будем использовать “сохранение на ходу”, так как оно более подходит для демонстрационных целей.

❓ Что значит BEGIN (implicit)? Вы могли заметить в логах линию с такой фразой в начале каждого SQL скрипта. “implicit” (с англ. подразумеваемый) в данном случае означает, что SQLAlchemy не будет отправлять никаких команд в начале транзакции, а будет это делать только тогда, когда это действительно потребуется.

Основы выполнения выражений

Итак, мы уже посмотрели на пару примеров выполнения SQL запросов к базе данных, используя такой метод как Connection.execute(), в дополнении к нему мы использовали текстовые запросы с помощью text() и получали как результат объект Result. В данном разделе мы подробнее познакомимся с данными механиками и взаимодействиями с ними.

В этом разделе мы будем также использовать метод Session.execute(), который является полным эквивалентом Connection.execute() и используется в ORM.

Итак, давайте в первую очередь познакомимся с объектом Result, мы будем получать те данные, которые вставили в БД ранее используя текстовый SELECT запрос:

```
with engine.connect()as conn:
    result = conn.execute(text("SELECT x, y FROM some_table"))
    for row in result:
        print(f"x:{row.x}  y:{row.y}")
```

```
BEGIN (implicit)
SELECT x, y FROM some_table
[...] ()
x: 1  y: 1
x: 2  y: 4
x: 6  y: 8
x: 9  y: 10
ROLLBACK
```

Итак, выше мы выполнили запрос на выделение всех строк из нашей базы данных. Нам вернулся объект Result, содержащий итератор для всех полученных строчек.

Result имеет множество методов для получения и преобразования строк, один из них - Result.all() - уже был продемонстрирован ранее, когда мы преобразовали данные в список. В дополнении к этому сам объект Result также реализует интерфейс итератора Python, так что мы можем проходить по всем элементам с помощью циклов, как это показано в примере выше.

Строка (или Row) - это объект, который ведет себя как именованный кортеж в Python, у нас есть несколько способов получать данные из него:

- **Получение в виде кортежа** - наиболее привычный для Python способ, который заключается в переборе строк как разложенных кортежей:

```
result = conn.execute(text("select x, y from some_table"))

for x, y in result:
    # ...
```

- **Получение данных по индексу** - кортежи в Python являются последовательностями, поэтому мы можем вытаскивать элементы из них по их индексу:

```
result = conn.execute(text("select x, y from some_table"))

  for row in result:
      x = row[0]
```

- **Получение данных по имени** - мы можем использовать функционал именованных кортежей, и доставать данные по их ключу (имени). Эти имена присваваются автоматически по имени столбцов в базе данных. Поэтому их название легко предугадать, хотя и встречаются редкие случаи, когда поведение определяется непосредственно базой данных.

```
result = conn.execute(text("select x, y from some_table"))

for row in result:
    y = row.y
    print(f"Row: {row.x} {y}")
```

- **Получение в виде ассоциативных массивов (карт)** - для получения строк в качестве ассоциативных массивов Python, которые по сути

являются неизменяемой версией словарей, объект Result имеет такой метод как Result.mappings(), преобразующий все строчки в подобные словарю RowMapping объекты.

```
result = conn.execute(text("select x, y from some_table"))

for dict_row in result.mappings():
    x = dict_row["x"]
    y = dict_row["y"]
```

Параметризация запросов

SQL выражения часто сопровождаются информацией, которую мы бы хотели им передать, как мы уже видели в INSERT примере ранее. Connection.execute() позволяет параметризовать запрос с помощью передачи позиционных аргументов. Самым простым примером может стать запрос SELECT с получением из базы данных только одной строчки, соответствующей следующему критерию: где значение y превышает конкретное значение (которое мы будем передавать в запросе).

Для достижения этого мы будем использовать ключевое слово WHERE, в выражение которого передадим соответствующий параметр. Конструктор text() позволяет сделать подобное с помощью специального синтаксиса :y. SQLAlchemy найдет эту конструкцию и заменит ее на то значение, что мы передадим в будущем:

```
with engine.connect() as conn:
     result = conn.execute(text("SELECT x, y FROM some_table WHERE y > :y"), {"y": 2})
     for row in result:
         print(f"x: {row.x}  y: {row.y}")
```

```
BEGIN (implicit)
SELECT x, y FROM some_table WHERE y > ?
[...] (2,)
x: 2  y: 4
x: 6  y: 8
x: 9  y: 10
ROLLBACK
```

Как вы можете заметить, конструкция :y в самом SQL скрипте заменилась на знак вопроса, который уже является классическим вариантом параметризации (qmark style) для Python DBAPI.

Всегда используйте позиционные аргументы для параметризации. Текстовый конструктор text() не самый обычный способ создания SQL, с которым мы работаем в алхимии. Однако при использовании текстового SQL литеральное значение Python, даже не связанное со строками, такими как целые числа или даты, никогда не должно быть преобразовано в строку SQL напрямую (во избежание SQL-инъекций), всегда следует использовать параметр. Помимо безопасности такой подход еще хорош тем, что позволяет диалектам SQLAlchemy лучше работать с получаемыми данными. Вне использования текстового конструктора, алхимия тем не менее гарантирует, что данные будут передаваться в качестве позиционных аргументов, где это необходимо.

> 🛠️ Стоит отметить, что текст в такой рамке будет иметь отношение только к Core части. Конечно, при использовании ORM эта информация также может быть актуальна, но в гораздо меньшей степени.

Множественная параметризация

В примере с сохранением изменений мы встретили INSERT скрипт, где использовалась вставка сразу нескольких строк в базу данных одновременно. Для выражений, работающих с данными, а не возвращающих результат (и не использующих ключевое слово RETURNING), называемых обобщенно DML (Insert, Upsert, Update), мы можем отправлять сразу множество параметров в качестве аргумента .execute()

```
with engine.connect() as conn:
     conn.execute(
         text("INSERT INTO some_table (x, y) VALUES (:x, :y)"),
         [{"x": 11, "y": 12}, {"x": 13, "y": 14}],
     )
     conn.commit()
```

```
BEGIN (implicit)
INSERT INTO some_table (x, y) VALUES (?, ?)
[...] ((11, 12), (13, 14))
```

```
<sqlalchemy.engine.cursor.CursorResult object at 0x...>
COMMIT
```

За кадром, объект соединения Connection использует функцию DBAPI cursor.executemany(). Данный метод по сути является эквивалентом того, если бы мы выполняли каждый INSERT запрос индивидуально в цикле. При этом DBAPI может оптимизировать эту операцию несколькими путями, используя заготовленные выражения (prepared statements) или с помощью конкатенации множеств параметров в один большой SQL скрипт. Но некоторые SQLAlchemy диалекты могут использовать собственные реализации данной функции, как это сделано в psycopg2 для PostgreSQL.

Вы могли заметить, что данная глава не особо касается темы концепций ORM. Это происходит в том числе потому, что множественные параметры обычно используются для INSERT выражений, которые реализуются в ORM совсем другими способами. Множества параметров также могут использоваться с инструкциями UPDATE и DELETE для создания различных операций UPDATE/DELETE для каждой строки, однако при использовании ORM подобные действия реализуются другими техниками, в том числе обновление и удаление данных может происходить для каждой строчки отдельно.

Выполнение с помощью ORM Session

Как было сказано ранее, многие паттерны и примеры для Core применимы и для ORM, так что теперь пришла пора продемонстрировать как именно мы можем использовать Core и ORM совместно.

Фундаментальный транзакционный объект при использовании ORM называется сессией (Session). В современной алхимии, этот объект во многом похож на соединение (Connection), и вообще говоря при использовании сессии под капотом она обращается к Connection.

Сессия иеет несколько паттернов создания, но сейчас мы продемонстрируем наиболее базовый способ, когда она создается в качестве контекстного менеджера, подобно соединению в Core

```
from sqlalchemy.orm import Session
```

```
stmt = text("SELECT x, y FROM some_table WHERE y > :y ORDER BY x, y")
with Session(engine) as session:
     result = session.execute(stmt, {"y": 6})
     for row in result:
         print(f"x: {row.x}  y: {row.y}")
```

```
BEGIN (implicit)
SELECT x, y FROM some_table WHERE y > ? ORDER BY x, y
[...] (6,)
x: 6  y: 8
x: 9  y: 10
x: 11  y: 12
x: 13  y: 14
ROLLBACK
```

Как вы можете заметить выше, пример полностью аналогичен предыдущему, мы просто заменили конструкцию `with engine.connect() as conn` на `with Session(engine) as session`, и теперь `Session.execute()` будет аналогичен `Connection.execute()`.

Также абсолютно аналогично соединению, сессия также поддерживает “сохранение на ходу”:

```
with Session(engine) as session:
     result = session.execute(
         text("UPDATE some_table SET y=:y WHERE x=:x"),
         [{"x": 9, "y": 11}, {"x": 13, "y": 15}],
     )
     session.commit()
```

```
BEGIN (implicit)
UPDATE some_table SET y=? WHERE x=?
[...] ((11, 9), (15, 13))
COMMIT
```

⚠️ Обратите внимание, что сессия (Session) не держит соединение после завершения транзакции. Она создает новое из машины соединений каждый раз при необходимости.

4. Работа с метаданными

Теперь, когда мы уже научились работать с машиной соединений и выполнять SQL скрипты, мы готовы перейти к магической алхимии. Центральным элементом как Core, так и ORM, является язык выражений SQL, который позволяет просто и быстро конструировать различные SQL скрипты. Фундаментом этих скриптов будут такие объекты как таблицы (Table) и столбцы (Column) в базе данных. Вместе мы будем называть их метаданными (MetaData) базы данных.

> 🌐 Практически вся эта глава посвящена ORM концепциям, поэтому если вы заинтересованы только в Core, то можете пропустить ее

Настройка метаданных с объектами таблиц

Когда мы работаем с реляционными базами данных, базовой структурой, которую мы можем создать и с которой можем работать, является таблица. В алхимии она представлена одноименным Python объектом - Table.

Для начала работы мы должны создать несколько таблиц. Каждая из них должна быть объявлена, что означает непосредственное создание ее напрямую в исходном Python коде. Мы также можем сгенерировать эти объекты на основе тех, что уже существуют в базе данных. Оба этих подхода могут быть реализованы несколькими способами.

Независимо от того, что мы планируем делать: объявлять таблицы или генерировать объекты на основе существующих - нам потребуется некоторая коллекция данных, названная MetaData. Этот объект является по большому счету фасадой вокруг словаря, который хранит в себе всю информацию о таблицах.

Создание метаданных выглядит вполне рутинно:

```
from sqlalchemy import MetaData
metadata_obj = MetaData()
```

Иметь один единственный объект метаданных на всё приложение - это вполне распространённый случай. Но также допускается и несколько коллекций метаданных, хотя, как правило, наиболее полезно, если таблицы, связанные друг с другом, принадлежат к одному объекту метаданных.

Как только у нас уже есть объект метаданных, мы можем объявить несколько таблиц. В данном туториале мы начнем с создания SQLAlchemy моделей пользователей (например, какого-то сайта) и e-mail адрессов, которые так или иначе относятся к пользователям. Обычно мы объявляем каждую таблицу в качестве переменной, на которую мы можем в дальнейшем ссылаться в нашем коде.

```
from sqlalchemy import Table, Column, Integer, String
user_table = Table(
    "user_account",
    metadata_obj,
    Column("id", Integer, primary_key=True),
    Column("name", String(30)),
    Column("fullname", String),
)
```

Мы можем наблюдать, что создание таблицы выглядит практически идентично выражению CREATE TABLE: оно также начинается с названия таблицы, затем происходит перечисление каждого столбца с названиями и типами данных. В примере были использованы такие объекты как:

- **Table** - представляет саму таблицу в базе данных, присоединенную к конкретному объекту метаданных
- **Column** - представляет столбец в базе данных и связывает его с конкретной таблицей. Объект Column обычно включает название в виде строчки и типа данных. Коллекция всех столбцов может быть представлена в коде в виде массива:

```
user_table.c.name
Column('name', String(length=30), table=<user_account>)

user_table.c.keys()
['id', 'name', 'fullname']
```

- **Integer, String** - эти классы помогают определить тип данных для каждого столбца, которые могут быть переданы с или без создания их экземпляра (в случае String мы передаем экземпляр, который помогает нам ограничить длину строчки, а в случае Integer передаем сам класс).

Объявление простых ограничений

Первый столбец в примере выше включает `Column.primary_key` параметр, который является простым способом показать, что данные в этом столбце будут частью первичного ключа для данной таблицы.

Ограничение, которое чаще всего объявляется явно - это ForeignKeyConstraint, которое соответствует ограничению внешнего ключа базы данных. Когда мы объявляем таблицы, связанные друг с другом, SQLAlchemy использует ограничения внешнего ключа не только для того, чтобы они были использованы в инструкции CREATE, но и для помощи в построении SQL-выражений.

Ограничение внешнего ключа, которое объявляется в одной таблице на другую, обычно создается с помощью упрощенной нотации с помощью объекта ForeignKey, ниже мы объявим вторую таблицу address, где будем ссылаться на таблицу user:

```
from sqlalchemy import ForeignKey
address_table = Table(
    "address",
    metadata_obj,
    Column("id", Integer, primary_key=True),
    Column("user_id", ForeignKey("user_account.id"), nullable=False),
    Column("email_address", String, nullable=False),
)
```

В этом примере мы также вводим и третий вид ограничения, который в SQL называют “NOT NULL Constraint”, оно вводится параметром `Column.nullable` и ограничивает возможность данного столбца быть пустым.

⚠️ При использовании внешнего ключа в объявлении столбца нам не нужно указывать тип данных, он будет автоматически определен из таблицы, на которую мы ссылаемся, в нашем примере внешний ключ user_account.id был автоматически определен как Integer.

Генерация DDL для базы данных

Итак, мы уже научились создавать четыре объекта для нашей базы данных: MetaData, Table, Column и ColumnConstraint. Этого уже достаточно для создания самых сложных структур баз данных.

И теперь самое полезное что мы можем со всем этим сделать - это превратить их в выражение CREATE TABLE (или DDL) для отправки в нашу базу данных. И что прекрасно, нам не нужно больше ничего особо делать, наша DDL уже почти создана, осталось вызвать метод MetaData.create_all() в нашем объекте метаданных:

```
BEGIN (implicit)
PRAGMA main.table_...info("user_account")
...
PRAGMA main.table_...info("address")
...
CREATE TABLE user_account (
    id INTEGER NOT NULL,
    name VARCHAR(30),
    fullname VARCHAR,
    PRIMARY KEY (id)
)
...
CREATE TABLE address (
    id INTEGER NOT NULL,
    user_id INTEGER NOT NULL,
    email_address VARCHAR NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(user_id) REFERENCES user_account (id)
)
...
COMMIT
```

```
metadata_obj.create_all(engine)
```

Процесс создания DDL обычно включает некоторые специфичные для SQL выражения PRAGMA, которые проверяют на наличие каждой таблицы перед тем как создавать ее. Полная последовательность шагов также включает в себя пару BEGIN/COMMIT для правильной работы с транзакциями (хотя в случае с SQLite, драйвер sqlite3 вообще работает в режиме “autocommit”).

Процесс создания таблиц также берет на себя ответственность по правильному сортированию выражений CREATE, выше, ограничение внешнего ключа делает

зависимым таблицу address от user, поэтому она создается после. В более сложных сценариях зависимостей ограничения внешнего ключа также могут быть применены к таблицам уже после, используя ALTER.

Объект метаданных также предоставляет возможность удалить все таблицы с помощью метода MetaData.drop_all(), который будет делать это в противоположном созданию порядке.

> ⚠️ **Используйте утилиты для миграций!** Подводя итог, необходимо сказать, что функции CREATE/DROP ALL полезны для тестов, маленьких приложений, либо для приложений с короткоживущей базой данных. Для работы с уже более сложными базами данных строго рекомендуется использовать такую утилиту для миграций как Alembic.

Объявление метаданных с помощью ORM

В данной части мы разберем несколько примеров как метаданные могут быть созданы с помощью объектов ORM. При использовании ORM, процесс объявление таблиц обычно объединен с процессом создания ассоциативных классов (проще сказать, любых классов, которые будут иметь такие же аттрибуты, как и столбцы в базе данных). Для этого опять же существует несколько способов, но самый часто используемый - это декларативный стиль.

Настройка регистра

При использовании ORM, объект метаданных также существует, но при этом содержится внутри объекта ORM, известного как регистр (Registry). Мы можем создать регистр следующим образом:

```
from sqlalchemy.orm import registry
mapper_registry = registry()
```

Объект регистра уже содержит в себе объект метаданных, который будет хранить в себе множество таблиц:

```
mapper_registry.metadata
MetaData()
```

Так что вместо объявления объектов таблиц напрямую, мы теперь можем объявлять их через различные классы-модели. В большинстве случаев каждый класс-модель, наследуется от декларативного базиса (declarative base). Мы можем получить новый декларативный базис из регистра с помощью метода registry.generate_base():

```
Base = mapper_registry.generate_base()
```

⚠️ Все действия по созданию регистра и декларативного базиса могут быть объединены в один простой шаг:

```
from sqlalchemy.orm import declarative_base

Base = declarative_base()
```

⚠️ При этом в новых версиях алхимии допускается и следующий вариант создания декларативного базиса:

```
from sqlalchemy.ext.declarative import as_declarative


@as_declarative()
class Base(object):
    id = Column(Integer, autoincrement=True, primary_key=True)
```

⚠️ В таком случае мы можем задать определенные столбцы для всех таблиц сразу. Это может быть удобно для создания автоинкремент первичного ключа, как это показано в примере выше

Создание классов-моделей

Созданный только что объект базовой модели теперь может быть использован для объявления (декларирования) новых моделей, которые будут соотнесены к соответствующим таблицам в базе данных. В нашем случае мы можем создать соответствующие модели `User` и `Address`:

```
from sqlalchemy.orm import relationship
class User(Base):
    __tablename__ = "user_account"

    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    fullname = Column(String)

    addresses = relationship("Address", back_populates="user")

    def __repr__(self):
        return f"User(id={self.id!r}, name={self.name!r}, fullname={self.fullname!r})"

class Address(Base):
    __tablename__ = "address"

    id = Column(Integer, primary_key=True)
    email_address = Column(String, nullable=False)
    user_id = Column(Integer, ForeignKey("user_account.id"))

    user = relationship("User", back_populates="addresses")

    def __repr__(self):
        return f"Address(id={self.id!r}, email_address={self.email_address!r})"
```

Здесь созданы два класса, которые теперь будут выполнять функции моделей, то есть с их помощью мы сможем как создавать новые записи, так и доставать уже существующие. При этом эти модели также будут иметь внутри объекты Table, знакомые нам по Core API. Мы можем получить этот объект с помощью __table__ аттрибута:

```
>>> User.__table__
Table('user_account', MetaData(),
    Column('id', Integer(), table=<user_account>, primary_key=True, nullable=False),
    Column('name', String(length=30), table=<user_account>),
    Column('fullname', String(), table=<user_account>), schema=None)
```

Данный объект таблицы был сгенерирован автоматически, при этом имена столбцов были получены из названий аттрибутов данной модели.

Другие детали при работе с моделями

Давайте поймем некоторые моменты при работе с моделями, которые вы могли заметить ранее в примерах:

- **все классы-модели имеют автоматически генерируемый __init__() метод**, который вы не должны переопределять в своем коде. Данный метод отвечает за передачу параметров для записи в столбцы таблицы

```
sandy = User(name="sandy", fullname="Sandy Cheeks")
```

- **мы можем переопределить __repr__() метод** - это уже полностью опционально и позволяет нам задать как именно будет выглядеть строковое представление объекта

```
>>> sandy
User(id=None, name='sandy', fullname='Sandy Cheeks')
```

- **мы также задали двустороннюю связь (relationship)** - это также опциональная возможность, которая реализуется с помощью такой ORM функции как relationship(), используемая сразу в двух классах-моделях, между которыми планируется проводить связь. В нашем случае User и Address ссылаются друг на друга в формате связей один-ко-многим и многие-к-одному. Использование связей будет лучше объяснено в рамках данного туториала позднее.

Совмещение Core и ORM таблиц

Вы уже знаете, что мы можем создавать таблицы как напрямую передавая их в MetaData, так и с помощью декларативного базиса. Но вы также можете совмещать сразу два способа создания таблиц.

Такие таблицы будут называться гибридными, и они содержат в качестве аттрибута заданный Table объект по имени __table__:

```
mapper_registry = registry()
Base = mapper_registry.generate_base()


class User(Base):
    __table__ = user_table

    addresses = relationship("Address", back_populates="user")

    def __repr__(self):
        return f"User({self.name!r}, {self.fullname!r})"


class Address(Base):
    __table__ = address_table

    user = relationship("User", back_populates="addresses")

    def __repr__(self):
        return f"Address({self.email_address!r})"
```

Вам скорее всего никогда не потребуется делать что-то подобное, но для исключительных ситуаций такая возможность есть.

Отображение таблиц

Чтобы завершить разбор возможностей по работе с таблицами, мы также должны проиллюстрировать операцию, которая уже была упомянута ранее - отображение таблиц. Данный процесс чаще всего ссылается на процесс создания таблицы и связанных объектов, поскольку читает текущее состояние базы данных. В то время как в предыдущих разделах мы объявляли объекты Table в Python, а затем отправляли соответствующий DDL в базу данных, процесс отображения делает тоже самое, но наоборот.

В качестве простого примера мы создадим таблицу с названием some_table. Опять же есть множество способов сделать это, но самый простой - создать таблицу и сразу же присвоить ее к конкретному объекту метаданных, единственное, чтобы все изменения и отображения происходили на ходу, вместо создания столбцов и ограничений, мы передадим машину соединений в параметр `Table.autoload_with`:

```
some_table = Table("some_table", metadata_obj, autoload_with=engine)
```

```
BEGIN (implicit)
PRAGMA main.table_...info("some_table")
[raw sql] ()
SELECT sql FROM  (SELECT * FROM sqlite_master UNION ALL   SELECT * FROM sqlite_temp_ma
ster) WHERE name = ? AND type = 'table'
[raw sql] ('some_table',)
PRAGMA main.foreign_key_list("some_table")
...
PRAGMA main.index_list("some_table")
...
ROLLBACK
```

В конце данного процесса объект таблицы some_table будет содержать информацию о всех колонках, существующих в таблице с одноименным названием:

```
>>> some_table
Table('some_table', MetaData(),
    Column('x', INTEGER(), table=<some_table>),
    Column('y', INTEGER(), table=<some_table>),
    schema=None)
```

⚠️ Обратите внимание, что вместо создания таблиц, алхимия, имея ввиду параметр autoload_with и при провальных PRAGMA проверках, не завершила транзакцию досрочно, а выполнила SELECT запрос с получением всех данных об этой таблице

5. Работа с данными

В разделе Работа с транзакциями и DBAPI мы изучили основы взаимодействия с Python DBAPI и его транзакционными состояниями. После, в разделе Работа с метаданными мы выяснили как представлены таблицы, столбцы и ограничения в алхимии, как используя объект метаданных, так и другие связанные объекты. В данном разделе мы научимся объединять эти две концепции чтобы создавать, получать и осуществлять манипуляции с данными. Наше взаимодействие с базой данных всегда будет оставаться транзакционным, даже если мы попросим наш драйвер совершать автоматические сохранения (autocommit) под капотом.

Вставка с помощью Core

При использовании Core, команда SQL INSERT генерируется с помощью функции insert().

> 🌐 В ORM этот процесс зациклен на сессии и происходит при работе такого паттерна как “объединение работ” (unit-of-work). Чисто номинально там нет такого понятия как вставка, но при этом она происходит под капотом при работе с Session.add() и Session.commit()

Давайте посмотрим на самый простой пример insert с передачей значений:

```
from sqlalchemy import insert
stmt = insert(user_table).values(name="spongebob", fullname="Spongebob Squarepants")
```

Обратите внимание, что выражение SQL было записано в переменную stmt. Практически все такие объекты можно сразу же превратить в строку:

```
>>> print(stmt)
INSERT INTO user_account (name, fullname) VALUES (:name, :fullname)
```

Строковое представление создается с помощью компилирования. Мы можем и сами скомпилировать объект выражений с помощью ClauseElement.compile()

```
compiled = stmt.compile()
```

Наша с вами конструкция INSERT является также параметризованной, поэтому мы можем посмотреть ее параметры:

```
>>> compiled.params
{'name': 'spongebob', 'fullname': 'Spongebob Squarepants'}
```

Выполнение скриптов

Теперь давайте выполним наш SQL скрипт, это мы уже умеем делать

```
with engine.connect() as conn:
    result = conn.execute(stmt)
    conn.commit()
```

```
BEGIN (implicit)
INSERT INTO user_account (name, fullname) VALUES (?, ?)
[...] ('spongebob', 'Spongebob Squarepants')
COMMIT
```

Этот простой пример показывает, что INSERT не возвращает никакого результата, однако, если вставляется только одна строчка, то мы обычно можем посмотреть данные, автоматически сгенерированные базой данных. Например, в данном случае это будет информация о первичном ключе:

```
>>> result.inserted_primary_key
(1,)
```

⚠️ Нам возвращается кортеж, это происходит поскольку первичный ключ может быть композитным и состоять из нескольких столбцов.

Выражения INSERT всегда генерируют VALUES автоматически

В примере выше мы могли заметить, что Insert.values() позволяет нам напрямую передать конструкцию VALUES в наше выражение INSERT. Этот

метод действительно удобен, он также позволяет включать множество параметров в запрос. Однако обычно мы пользуемся другой конструкцией INSERT, где алхимия автоматически сгенерирует конструкцию VALUES и связана она с уже знакомым нам .execute():

```
with engine.connect() as conn:
    result = conn.execute(
        insert(user_table),
        [
            {"name": "sandy", "fullname": "Sandy Cheeks"},
            {"name": "patrick", "fullname": "Patrick Star"},
        ],
    )
    conn.commit()
```

```
BEGIN (implicit)
INSERT INTO user_account (name, fullname) VALUES (?, ?)
[...] (('sandy', 'Sandy Cheeks'), ('patrick', 'Patrick Star'))
COMMIT
```

Вы можете вспомнить, что мы уже делали что-то подобное ранее в главе с Множественной параметризацией, только там мы работали с text(), но в случае insert() ситуация никак не поменялась.

▼ INSERT…FROM SELECT

Конструкция INSERT может включать в себя данные из других уже существующих таблиц, которые мы можем получить с помощью SELECT, такое действие нам позволяет сделать Insert.from_select()

```
select_stmt = select(user_table.c.id, user_table.c.name + "@aol.com")
insert_stmt = insert(address_table).from_select(
    ["user_id", "email_address"], select_stmt
)
print(insert_stmt)
```

```
INSERT INTO address (user_id, email_address)
SELECT user_account.id, user_account.name || :name_1 AS anon_1
FROM user_account
```

▼ INSERT…RETURNING

Мы также можем включить в INSERT конструкцию RETURNING, которая позволяет автоматически получать первичный ключ (было обсуждено ранее). Однако RETURNING может быть использован и для других целей с помощью Insert.returning()

```
insert_stmt = insert(address_table).returning(
    address_table.c.id, address_table.c.email_address
)
print(insert_stmt)
```

```
INSERT INTO address (id, user_id, email_address)
VALUES (:id, :user_id, :email_address)
RETURNING address.id, address.email_address
```

Вы можете сочетать эту конструкцию с INSERT…FROM SELECT:

```
select_stmt = select(user_table.c.id, user_table.c.name + "@aol.com")
insert_stmt = insert(address_table).from_select(
    ["user_id", "email_address"], select_stmt
)
print(insert_stmt.returning(address_table.c.id, address_table.c.email_address))
```

```
INSERT INTO address (user_id, email_address)
SELECT user_account.id, user_account.name || :name_1 AS anon_1
FROM user_account RETURNING address.id, address.email_address
```

Возможности RETURNING также поддерживаются конструкциями UPDATE и DELETE, которые будут обсуждены немного позднее. Вообще говоря, стоит сказать, что RETURNING поддерживается только в случае выполнения скриптов с одним набором параметров, и не будут работать со множественной параметризацией, обсужденной ранее. При этом некоторые диалекты, например Oracle, разрешают возвращать исключительно один столбец (например, только первичный ключ), что означает невозможность работы с конструкцией INSERT…FROM SELECT и множественными Update/Delete

Выбор и получение данных

Что для Core, что для ORM, функция select() генерирует конструкцию, которая используется для всех запросов с SQL командой SELECT. Вы можете передавать в нее те или иные параметры, подобно тому, как мы уже это делали ранее, и получать ответ от базы данных в объекте Result.

> 🌐 Данная глава будет полезна как для Core, так и для ORM частей, поскольку здесь будут обсуждены оба варианта использования

Конструкция select()

Данная функция генерирует конструкцию SELECT подобно тому, как это делает insert(), также как и другие конструкции, она тоже может быть сразу же представлена в виде SQL скрипта:

```
from sqlalchemy import select
stmt = select(user_table).where(user_table.c.name == "spongebob")
print(stmt)
```

```
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = :name_1
```

Причем также как и ранее, вы можете выполнить этот скрипт с помощью любого .execute()

```
with engine.connect() as conn:
    for row in conn.execute(stmt):
        print(row)
```

```
BEGIN (implicit)
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?
[...] ('spongebob',)
(1, 'spongebob', 'Spongebob Squarepants')
ROLLBACK
```

При работе с ORM мы можем также захотеть осуществлять выбор данных, в таком случае логично использовать Session.execute(), при этом, используя данный подход, мы можем продолжать получать объекты в виде Row, однако эти строки теперь могут быть представлены в виде классов-моделей, созданных на основе декларативного базиса

```
stmt = select(User).where(User.name == "spongebob")
with Session(engine) as session:
    for row in session.execute(stmt):
        print(row)
```

```
BEGIN (implicit)
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?
[...] ('spongebob',)
(User(id=1, name='spongebob', fullname='Spongebob Squarepants'),)
ROLLBACK
```

Вся разница между ORM и Core заключается в том как будет выглядеть аргумент функции select() - как таблица user_table или как класс-модель User. В общем случае они не обязательно отображают одно и тоже, так как класс-модель ORM, может быть сопоставлен и с другими таблицами. Еще одним отличием является тот факт, что использование модели в качестве аргумента select() говорит алхимии приводить полученные данные к соответствующим моделям, а не выводить их в виде просто кортежей.

Использование выражений COLUMNS и FROM

Функция select() допускает позиционные аргументы, которые могут представлять любое количество табличных или столбцовых выражений, при этом все они будут совмещены в выводе после обращения к базе данных.

```
print(select(user_table))
```

```
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
```

Чтобы выбрать, например, только один столбец, используя Core, мы можем передать их также в аргументы функции

```
print(select(user_table.c.name, user_table.c.fullname))
```

```
SELECT user_account.name, user_account.fullname
FROM user_account
```

Выбор ORM сущностей и столбцов

Если же мы хотим выбрать сущности ORM, например, модель User, то мы можем использовать несколько похожий подход

```
print(select(User))
```

```
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
```

При выполнении таких скриптов через Session.execute() стоит уметь работать с результатом выбора

```
>>> row = session.execute(select(User)).first()

BEGIN...
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
[...] ()

>>> row
(User(id=1, name='spongebob', fullname='Spongebob Squarepants'),)
```

Как вы видите объект Row содержит только один экземляпр модели User:

```
>>> row[0]
User(id=1, name='spongebob', fullname='Spongebob Squarepants')
```

При этом все эти действия можно упростить, используя такой метод как Session.scalars(), данный метод вместо обычного Result вернет ScalarResult, который будет сразу же преобразовывать каждую строчку в соответствующий объект, что избавит нас от лишних действий:

```
>>> user = session.scalars(select(User)).first()

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
[...] ()

>>> user
User(id=1, name='spongebob', fullname='Spongebob Squarepants')
```

Мы можем также выбирать не всю модель, а только отдельные столбцы, подобно тому, как мы это делали в Core:

```
>>> print(select(User.name, User.fullname))

SELECT user_account.name, user_account.fullname
FROM user_account
```

Когда мы выполним данный скрипт, мы получим строчки в следующем виде:

```
>>> row = session.execute(select(User.name, User.fullname)).first()

SELECT user_account.name, user_account.fullname
FROM user_account
[...] ()

>>> row
('spongebob', 'Spongebob Squarepants')
```

Никто не запрещает нам также смешивать данные подходы и использовать как выбор целых моделей, так и отдельных столбцов вместе:

```
>>> session.execute(
    select(User.name, Address).where(User.id == Address.user_id).order_by(Address.id)
).all()

SELECT user_account.name, address.id, address.email_address, address.user_id
FROM user_account, address
WHERE user_account.id = address.user_id ORDER BY address.id
[...] ()

[('spongebob', Address(id=1, email_address='spongebob@sqlalchemy.org')),
('sandy', Address(id=2, email_address='sandy@sqlalchemy.org')),
('sandy', Address(id=3, email_address='sandy@squirrelpower.org'))]
```

Выбор данных с применением алиасов

Мы можем задать определенный алиас для того или иного столбца с помощью такого метода как ColumnElement.label(). Присваивая алиасы мы получим вместо названий столбца те самые алиасы:

```
from sqlalchemy import func, cast

stmt = select(
    ("Username: " + user_table.c.name).label("username"),
).order_by(user_table.c.name)

with engine.connect() as conn:
    for row in conn.execute(stmt):
        print(f"{row.username}")
```

```
BEGIN (implicit)
SELECT ? || user_account.name AS username
FROM user_account ORDER BY user_account.name
[...] ('Username: ',)
Username: patrick
Username: sandy
Username: spongebob
ROLLBACK
```

Как вы можете заметить в результате, мы, обращаясь к `row.username`, получаем строчку `Username: …` - то самое поведение, которое мы и ожидали.

Обратите внимание, что здесь мы также применяем метод `.order_by`, который однако будет обсужден позднее.

Поиск и выбор данных по тексту

Когда мы создаем SELECT конструкции, применяя функцию select(), мы можем передавать объекты Table и Column, или же, в случае ORM, классы-модели. Однако в некоторых ситуациях может потребоваться передать более сложные условия поиска, например, текстовые условия. Метод text(), уже обсужденный ранее, может быть использован и в select() напрямую.

```
from sqlalchemy import text
stmt = select(text("'some phrase'"), user_table.c.name).order_by(user_table.c.name)
with engine.connect() as conn:
    print(conn.execute(stmt).all())
```

```
BEGIN (implicit)
SELECT 'some phrase', user_account.name
```

```
FROM user_account ORDER BY user_account.name
[generated in ...] ()
[('some phrase', 'patrick'), ('some phrase', 'sandy'), ('some phrase', 'spongebob')]
ROLLBACK
```

Пока text() больше полезен для встраивания текстовых конструкций в SQL скрипт, мы все же чаще имеем дело с отдельными текстовыми единицами, каждая из которых представляет собой отдельное выражение для того или иного столбца. В таком случае нам может быть полезнее использовать literal_column(). Эта функция несколько похожа на text() за исключением того, что она представляет не какое либо SQL выражение, а именно отдельный столбец, который мы можем назвать и с которым можем дальше работать

```
from sqlalchemy import literal_column
stmt = select(literal_column("'some phrase'").label("p"), user_table.c.name).order_by(
    user_table.c.name
)
with engine.connect() as conn:
    for row in conn.execute(stmt):
        print(f"{row.p}, {row.name}")
```

```
BEGIN (implicit)
SELECT 'some phrase' AS p, user_account.name
FROM user_account ORDER BY user_account.name
[generated in ...] ()
some phrase, patrick
some phrase, sandy
some phrase, spongebob
ROLLBACK
```

Обратите внимание, что при использовани и text() и literal_column() мы пишем синтаксические выражения SQL, а не значения. Поэтому нам необходимо включать в тексте одинарные ковычки в соответствии с синтаксическими правилами SQL.

Конструкция WHERE

Алхимия предоставляет возможность собирать самые разные выражения SQL с помощью своего собственного языка выражений, интегрированного в Python. Давайте взглянем на простой пример условия:

```
>>> print(user_table.c.name == "squidward")
user_account.name = :name_1
```

```
>>> print(address_table.c.user_id > 10)
address.user_id > :user_id_1
```

Как вы можете заметить, при попытке сравнить столбец с чем либо, создается сразу же объект Statement.

Мы можем на этой основе создать конструкцию WHERE с помощью функции `.where()`

```
>>> print(select(user_table).where(user_table.c.name == "squidward"))

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = :name_1
```

Чтобы объединить сразу несколько условных конструкций, можно использовать функцию `.where()` несколько раз:

```
>>> print(
    select(address_table.c.email_address)
    .where(user_table.c.name == "squidward")
    .where(address_table.c.user_id == user_table.c.id)
)

SELECT address.email_address
FROM address, user_account
WHERE user_account.name = :name_1 AND address.user_id = user_account.id
```

Или, если такой подход не нравится, то можно объединить два условия и в одной функции:

```
>>> print(
    select(address_table.c.email_address).where(
        user_table.c.name == "squidward",
        address_table.c.user_id == user_table.c.id,
    )
)

SELECT address.email_address
FROM address, user_account
WHERE user_account.name = :name_1 AND address.user_id = user_account.id
```

Помимо прочего, вы также можете создавать более сложные условные конструкции в том числе с применением оператора OR, для этого можно

использовать встроенные функции `and_` и `or_`:

```
>>> from sqlalchemy import and_, or_
>>> print(
    select(Address.email_address).where(
        and_(
            or_(User.name == "squidward", User.name == "sandy"),
            Address.user_id == User.id,
        )
    )
)

SELECT address.email_address
FROM address, user_account
WHERE (user_account.name = :name_1 OR user_account.name = :name_2)
AND address.user_id = user_account.id
```

❓ Функции and_ и or_ содержат в имени нижнее подчеркивание лишь для того, чтобы избежать конфликта имен с ключевыми словами Python and/or

Помимо прочего мы также можем создать конструкцию WHERE куда более по-питоновски с помощью метода `.filter_by`

```
>>> print(select(User).filter_by(name="spongebob", fullname="Spongebob Squarepants"))

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = :name_1 AND user_account.fullname = :fullname_1
```

Явные конструкции FROM и JOIN’ы

Как было сказано ранее, выражение FROM обычно создается на основе тех данных, что мы явно указываем в select(). Если мы хотим выбрать один столбец из таблицы, то алхимия самостоятельно неявно создаст конструкцию FROM для этой операции:

```
>>> print(select(user_table.c.name))

SELECT user_account.name
FROM user_account
```

Если мы захотим выбрать два столбца из двух разных таблиц, то в таком случае всё также будет просто:

```
>>> print(select(user_table.c.name, address_table.c.email_address))

SELECT user_account.name, address.email_address
FROM user_account, address
```

Если мы хотим сделать JOIN запрос на две таблицы, то мы обычно будет использовать один двух методов Select. Первый - `Select.join_from` - позволяет указать левую и правую часть конструкции явно:

```
>>> print(
    select(user_table.c.name, address_table.c.email_address).join_from(
        user_table, address_table
    )
)

SELECT user_account.name, address.email_address
FROM user_account JOIN address ON user_account.id = address.user_id
```

А вот в случае с Select.join(), у нас не будет вариантов, данный метод будет принудительно работать с правой стороной JOIN:

```
>>> print(select(user_table.c.name, address_table.c.email_address).join(address_tabl
e))

SELECT user_account.name, address.email_address
FROM user_account JOIN address ON user_account.id = address.user_id
```

Алхимия позволяет нам также указывать элементы FROM явно, если это необходимо при построении запроса. Для этого мы можем использовать метод `Select.select_from()`:

```
>>> print(select(address_table.c.email_address).select_from(user_table).join(address_t
able))

SELECT address.email_address
FROM user_account JOIN address ON user_account.id = address.user_id
```

Другой пример когда нам может потребоваться использовать явное FROM: если конструкции только на столбцах не дают достаточно наводок о той

информации, что мы собираемся достать из базы данных. Например, SELECT с помощью простого SQL выражения - count(*)

```
>>> from sqlalchemy import func
>>> print(select(func.count("*")).select_from(user_table))

SELECT count(:count_2) AS count_1
FROM user_account
```

Работа с конструкцией ON

Предыдущий пример JOIN'ов показал, что конструкция SELECT может JOIN'ить две разные таблицы и создавать конструкцию ON автоматически. Это происходит поскольку мы включаем две таблицы, имеющие между собой ForeignKeyConstraint (address_table ссылается на user_table).

Но может возникнуть необходимость организации JOIN между таблицами, которые не имеют подобной связи. Следовательно, нам потребуется указывать ON явно. Оба метода Select.join() и Select.join_from() позволяют указать дополнительный аргумент для конструкции ON:

```
>>> print(
    select(address_table.c.email_address)
    .select_from(user_table)
    .join(address_table, user_table.c.id == address_table.c.user_id)
)

SELECT address.email_address
FROM user_account JOIN address ON user_account.id = address.user_id
```

При работе с ORM вы можете также создавать такие связи с помощью relationship(), который будет обсужден позднее.

OUTER и FULL JOIN

Оба метода JOIN также принимают ключевой аргумент Select.join.isouter и Select.join.full, которые позволяют создать либо LEFT OUTER JOIN или FULL OUTER JOIN:

```
>>> print(select(user_table).join(address_table, isouter=True))

SELECT user_account.id, user_account.name, user_account.fullname
```

```
FROM user_account LEFT OUTER JOIN address ON user_account.id = address.user_id

>>> print(select(user_table).join(address_table, full=True))

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account FULL OUTER JOIN address ON user_account.id = address.user_id
```

Вы также можете применять метод `Select.outerjoin()`, эквивалентный `.join(…, isouter=True)`.

Вы наверняка знаете о том, что в SQL также имеется RIGHT OUTER JOIN. Алхимия, однако, не позволяет генерировать эту конструкцию напрямую. Вы можете достичь того же результата разворотом таблицы и использованием LEFT OUTER JOIN.

ORDER BY, GROUP BY, HAVING

Выражение SELECT также может включать конструкцию ORDER BY, которая возвращает результат выборки в заданном порядке.

GROUP BY создается похоже на ORDER BY, и имеет цель разделения выборки на субгруппы с целью взаимодействия с каждой такой субгруппой по отдельности (например, сортировка сначала по всей выборке по ID, а затем сортировка по дате добавления внутри каждой субгруппы).

Выражение ORDER BY конструируется с помощью такого метода как Select.order_by():

```
>>> print(select(user_table).order_by(user_table.c.name))

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account ORDER BY user_account.name
```

По умолчанию используется ASC (ascending / по возрастанию), но вы можете также применить DESC (descending / по убыванию) с помощью `.desc()`:

```
>>> print(select(User).order_by(User.fullname.desc()))

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account ORDER BY user_account.fullname DESC
```

Агрегатные функции с GROUP BY / HAVING

В SQL функции, позволяющие работать с несколькими строками одного или нескольких столбцов и производящие единый результат (например, количество строк, или среднее значение), называются агрегатными.

SQLAlchemy позволяет работать с такими функции напрямую, используя неймспейс `func`. Это специальный конструктор объектов, который создает новую инстанцию Function каждый раз, когда мы даем ему имя какой-то SQL функции. Например, чтобы сгенерировать SQL COUNT() мы должны сделать следующее:

```
>>> from sqlalchemy import func
>>> count_fn = func.count(user_table.c.id)
>>> print(count_fn)

count(user_account.id)
```

Алхимия не будет разбирать какую именно функцию вы ей передадите. Она просто возьмет название, которое вы передали и сгенерирует с ним SQL запрос. То есть, если вы передадите в func несуществующую функцию (например, func.some_func), то алхимия сгенерирует запрос с этой функцией, хотя ее может не существовать в СУБД.

При использовании агрегатных функций в SQL конструкция GROUP BY имеет важное значение, поскольку оно позволяет разбивать строки на группы, где работа будет происходить в каждой группе индивидуально. При запросе неагрегированных столбцов в конструкции COLUMNS оператора SELECT, SQL требует чтобы все эти столбцы подпадали под действие GROUP BY, прямо или косвенно основанное на первичном ключе. Предложение HAVING затем используется так же, как и предложение WHERE, за исключением того, что оно отфильтровывает строки на основе агрегированных значений, а не прямого содержимого строк.

Алхимия позволяет работать с этими двумя предложениями с помощью `Select.group_by` и `Select.having`

```
>>> with engine.connect() as conn:
    result = conn.execute(
```

```
        select(User.name, func.count(Address.id).label("count"))
        .join(Address)
        .group_by(User.name)
        .having(func.count(Address.id) > 1)
    )
    print(result.all())

BEGIN (implicit)
SELECT user_account.name, count(address.id) AS count
FROM user_account JOIN address ON user_account.id = address.user_id GROUP BY user_acco
unt.name
HAVING count(address.id) > ?
[...] (1,)

[('sandy', 2)]

ROLLBACK
```

Сортировка и группировка с помощью именований Label

Важная техника, применяемая в некоторых СУБД, это возможность сгруппировать или отсортировать по переменной, которая уже была получена в результате агрегирования (иногда такие выражения называют агрегатами). Например, если мы хотим отсортировать пользователей по числу адресов, которые они имеют. Для этой цели нам понадобится предварительно сгруппировать адресса (GROUP BY), затем посчитать их количество (COUNT), для удобства присвоить какое либо имя данному значению (например, num_addresses) и уж только за тем отсортировать по этому значению. Мы вполне можем сделать всё перечисленное средствами алхимии, для упрощения процесса будем использовать именования `.label()`, для этого необходимо передать текстовое значение в эту функцию:

```
from sqlalchemy import func, desc
stmt = (
    select(Address.user_id, func.count(Address.id).label("num_addresses"))
    .group_by("user_id")
    .order_by("user_id", desc("num_addresses"))
)
print(stmt)

SELECT address.user_id, count(address.id) AS num_addresses
FROM address GROUP BY address.user_id ORDER BY address.user_id, num_addresses DESC
```

Использование именований

Теперь, когда мы уже можем делать выборки из нескольких таблиц с помощью JOIN, мы также должны уметь ссылаться на одну и ту же таблицу несколько раз. Для этого мы можем назвать таблицу или какое-то выражение дополнительным именем (alias) и в дальнейшем использовать его для формирования запросов.

В SQLAlchemy такие имена задаются методом FromClause.alias(). В таком случае alias представляет собой конструкцию таблицы, которая названа по тому или иному имени:

```
>>> user_alias_1 = user_table.alias()
>>> user_alias_2 = user_table.alias()
>>> print(
    select(user_alias_1.c.name, user_alias_2.c.name).join_from(
        user_alias_1, user_alias_2, user_alias_1.c.id > user_alias_2.c.id
    )
)

SELECT user_account_1.name, user_account_2.name AS name_1
FROM user_account AS user_account_1
JOIN user_account AS user_account_2 ON user_account_1.id > user_account_2.id
```

Именования для сущностей ORM

В случае ORM для создания именований мы можем использовать эквивалент - функцию aliased().

```
>>> from sqlalchemy.orm import aliased
>>> address_alias_1 = aliased(Address)
>>> address_alias_2 = aliased(Address)
>>> print(
    select(User)
    .join_from(User, address_alias_1)
    .where(address_alias_1.email_address == "patrick@aol.com")
    .join_from(User, address_alias_2)
    .where(address_alias_2.email_address == "patrick@gmail.com")
)

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
JOIN address AS address_1 ON user_account.id = address_1.user_id
JOIN address AS address_2 ON user_account.id = address_2.user_id
WHERE address_1.email_address = :email_address_1
AND address_2.email_address = :email_address_2
```

Субзапросы

Субзапросы в SQL это дополнительные выборки, которые создаются для того, чтобы конструировать более сложные логические ветвления.

Данный раздел расскажет обо всех видах субзапросов, в том числе так называемые нескалярные запросы, которые часто называют **обобщенными выражениями таблиц (OBT),** применяемые похоже на субзапросы, но имеющие дополнительные особенности.

Алхимия предоставляет субзапросы и OBT в виде соответствующих объектов - Subquery и CTE (common table expression), которые могут быть сгенерированы с помощью применения методов `Select.subquery()` и `Select.cte()`. Оба этих метода могут быть использованы в качестве элемента в запросе FROM внутри большой `select()` конструкции.

Давайте для примера создадим Subquery, который будет считать количество строк в таблице address:

```
>>> subq = (
     select(func.count(address_table.c.id).label("count"), address_table.c.user_id)
     .group_by(address_table.c.user_id)
     .subquery()
)
```

Если мы попробуем вывести этот запрос в виде SQL скрипта, то получим:

```
>>> print(subq)

SELECT count(address.id) AS count, address.user_id
FROM address GROUP BY address.user_id
```

Субзапросы позволяют в дальнейшем обращаться к результатам их выполнения, в чём и состоит их основная идея:

```
>>> print(select(subq.c.user_id, subq.c.count))

SELECT anon_1.user_id, anon_1.count
FROM (SELECT count(address.id) AS count, address.user_id AS user_id
FROM address GROUP BY address.user_id) AS anon_1
```

При работе со строками из субзапроса, мы можем также осуществлять операции JOIN:

```
>>> stmt = select(user_table.c.name, user_table.c.fullname, subq.c.count).join_from(
    user_table, subq
)

>>> print(stmt)

SELECT user_account.name, user_account.fullname, anon_1.count
FROM user_account JOIN (SELECT count(address.id) AS count, address.user_id AS user_id
FROM address GROUP BY address.user_id) AS anon_1 ON user_account.id = anon_1.user_id
```

Для того, чтобы сделать JOIN из user_account в address, мы использовали метод Select.join_from(). Как было показано ранее, конструкция ON в данном случае было снова сгенерировано на основе первичного ключа. Несмотря на то, что субзапрос не включает никакие ограничения, алхимия может взаимодействовать с ними (в т.ч. с ограничением первичного ключа) на основе того, что значения субзапроса (например, subq.c.user_id) приходят из реальных таблиц (address_table.c.user_id).

Обобщенные выражения таблиц (OBT/CTE)

🌐 Обобщенное табличное выражение или CTE (Common Table Expressions) - это временный результирующий набор данных, к которому можно обращаться в последующих запросах. Для написания обощенного табличного выражения используется оператор WITH.

Использование конструкций OBT в алхимии в сущности ничем не отличается от субзапросов. С помощью изменения метода Select.subquery() на Select.cte() мы вмиг достигаем необходимого результата без какого либо изменения самого запроса. При этом, в случае вывода скомпилированного SQL скрипта мы увидим существенную разницу:

```
>>> subq = (
    select(func.count(address_table.c.id).label("count"), address_table.c.user_id)
    .group_by(address_table.c.user_id)
    .cte()
)

>>> stmt = select(user_table.c.name, user_table.c.fullname, subq.c.count).join_from(
    user_table, subq
```

```
)

>>> print(stmt)

WITH anon_1 AS
(SELECT count(address.id) AS count, address.user_id AS user_id
FROM address GROUP BY address.user_id)
 SELECT user_account.name, user_account.fullname, anon_1.count
FROM user_account JOIN anon_1 ON user_account.id = anon_1.user_id
```

Конструкция OBT также имеет возможность использоваться в «рекурсивном» стиле и может в более сложных случаях быть составлена из предложения RETURNING оператора INSERT, UPDATE или DELETE.

В обоих случаях, и субзапрос и OBT были названы в SQL “анонимным” именем. В коде питона мы не обязаны предоставлять эти имена вовсе. Объекты Subquery или OBT берут на себя функцию по синтаксической идентификации при компиляции с SQL. Но мы также можем указывать имена с помощью передачи их в виде первого аргумента методов Select.subquery() и Select.cte().

Субзапросы и OBT в ORM

В ORM конструкция aliased() может использоваться для связывания сущности ORM, такой как наш класс User или Address, с FromClause. Ранее мы уже обсуждали использование aliased() для связывания сопоставленного класса с псевдонимом его сопоставленной таблицы. Здесь мы иллюстрируем aliased(), делающий то же самое как с подзапросом, так и с OBT, сгенерированными с помощью конструкции Select.

Ниже приведен пример применения aliased() к конструкции Subquery, чтобы сущности ORM можно было извлечь из его строк. Результат показывает серию объектов User и Address, где данные для каждого объекта Address в конечном итоге поступили из субзапроса к таблице адресов, а не из этой таблицы напрямую:

```
subq = select(Address).where(~Address.email_address.like("%@aol.com")).subquery()
address_subq = aliased(Address, subq)
stmt = (
    select(User, address_subq)
    .join_from(User, address_subq)
    .order_by(User.id, address_subq.id)
)
with Session(engine) as session:
    for user, address in session.execute(stmt):
        print(f"{user} {address}")
```

```
BEGIN (implicit)
SELECT user_account.id, user_account.name, user_account.fullname,
anon_1.id AS id_1, anon_1.email_address, anon_1.user_id
FROM user_account JOIN
(SELECT address.id AS id, address.email_address AS email_address, address.user_id AS u
ser_id
FROM address
WHERE address.email_address NOT LIKE ?) AS anon_1 ON user_account.id = anon_1.user_id
ORDER BY user_account.id, anon_1.id
[...] ('%@aol.com',)

User(id=1, name='spongebob', fullname='Spongebob Squarepants') Address(id=1, email_add
ress='spongebob@sqlalchemy.org')
User(id=2, name='sandy', fullname='Sandy Cheeks') Address(id=2, email_address='sandy@s
qlalchemy.org')
User(id=2, name='sandy', fullname='Sandy Cheeks') Address(id=3, email_address='sandy@s
quirrelpower.org')

ROLLBACK
```

Следующий пример показывает также работу с OBT:

```
cte_obj = select(Address).where(~Address.email_address.like("%@aol.com")).cte()
address_cte = aliased(Address, cte_obj)
stmt = (
    select(User, address_cte)
    .join_from(User, address_cte)
    .order_by(User.id, address_cte.id)
)
with Session(engine) as session:
    for user, address in session.execute(stmt):
        print(f"{user} {address}")
```

```
BEGIN (implicit)
WITH anon_1 AS
(SELECT address.id AS id, address.email_address AS email_address, address.user_id AS u
ser_id
FROM address
WHERE address.email_address NOT LIKE ?)
SELECT user_account.id, user_account.name, user_account.fullname,
anon_1.id AS id_1, anon_1.email_address, anon_1.user_id
FROM user_account
JOIN anon_1 ON user_account.id = anon_1.user_id
ORDER BY user_account.id, anon_1.id
[...] ('%@aol.com',)

User(id=1, name='spongebob', fullname='Spongebob Squarepants') Address(id=1, email_add
ress='spongebob@sqlalchemy.org')
User(id=2, name='sandy', fullname='Sandy Cheeks') Address(id=2, email_address='sandy@s
```

```
qlalchemy.org')
User(id=2, name='sandy', fullname='Sandy Cheeks') Address(id=3, email_address='sandy@s
quirrelpower.org')

ROLLBACK
```

Скалярные и кореллированные субзапросы

Скалярные субзапросы это запросы, которые возвращают либо ничего, либо только одну колонку (например, id). Такого рода субзапросы чаще всего применяются в COLUMNS или WHERE выражениях и отличаются от обычных субзапросов тем, что не используются в выражении FROM. Коррелированные субзапросы - скалярные субзапросы, ссылающиеся на таблицу в составе выражения SELECT.

Алхимия предоставляет возможность работать со скалярными субзапросами с помощью конструкции ScalarSelect, которая может быть частью ColumnElement, в отличие от обычных субзапросов, которые используются в FromClause.

```
subq = (
    select(func.count(address_table.c.id))
    .where(user_table.c.id == address_table.c.user_id)
    .scalar_subquery()
)
print(subq)
```

```
(SELECT count(address.id) AS count_1
FROM address, user_account
WHERE user_account.id = address.user_id)
```

Хотя скалярный субзапрос сам по себе отображает как user_account, так и adress в своем предложении FROM, когда он компилируется в SQL, при встраивании его во вложенную конструкцию select(), которая имеет дело с таблицей user_account, таблица user_account автоматически кореллирует, что означает, что она не рендерится в выражение FROM субзапроса:

```
stmt = select(user_table.c.name, subq.label("address_count"))
print(stmt)
```

```
SELECT user_account.name, (SELECT count(address.id) AS count_1
FROM address
WHERE user_account.id = address.user_id) AS address_count
FROM user_account
```

Простые кореллированные субзапросы обычно делают ровно то, что мы и хотели. Однако, в случае где кореллирование довольно сложное, алхимия попросит у нас уточнений:

```
stmt = (
    select(
        user_table.c.name,
        address_table.c.email_address,
        subq.label("address_count"),
    )
    .join_from(user_table, address_table)
    .order_by(user_table.c.id, address_table.c.id)
)
print(stmt)
```

```
Traceback (most recent call last):
...
InvalidRequestError: Select statement '<... Select object at ...>' returned
no FROM clauses due to auto-correlation; specify correlate(<tables>) to
control correlation manually.
```

Чтобы указать user_table для кореллирования, мы можем использовать `ScalarSelect.correlate()` или `ScalarSelect.correlate_except()`:

```
subq = (
    select(func.count(address_table.c.id))
    .where(user_table.c.id == address_table.c.user_id)
    .scalar_subquery()
    .correlate(user_table)
)
```

Выражение в таком случае возвратит:

```
with engine.connect() as conn:
    result = conn.execute(
        select(
            user_table.c.name,
            address_table.c.email_address,
            subq.label("address_count"),
```

```
        )
        .join_from(user_table, address_table)
        .order_by(user_table.c.id, address_table.c.id)
    )
    print(result.all())
```

```
BEGIN (implicit)
SELECT user_account.name, address.email_address, (SELECT count(address.id) AS count_1
FROM address
WHERE user_account.id = address.user_id) AS address_count
FROM user_account JOIN address ON user_account.id = address.user_id ORDER BY user_acco
unt.id, address.id
[...] ()

[('spongebob', 'spongebob@sqlalchemy.org', 1), ('sandy', 'sandy@sqlalchemy.org', 2),
 ('sandy', 'sandy@squirrelpower.org', 2)]

ROLLBACK
```

Побочное кореллирование

Побочное кореллирование (LATERAL) - одна из специальных подкатегорий SQL-корелляций, которая позволяет выбираемой единице ссылаться на другую в рамках одного выражения FROM. Это супер специфичный случай для обычной практики SQL, и на данный момент такое поддерживается лишь только последними версиями PostgreSQL.

Обычно, если оператор SELECT ссылается на таблицу 1 *JOIN (SELECT ...) AS* в своем предложении FROM, подзапрос с правой стороны не может ссылаться на выражение «table1» с левой стороны; корреляция может относиться только к таблице, которая является частью другого SELECT, которая полностью включает этот SELECT. Ключевое слово LATERAL позволяет нам изменить это поведение и разрешить корреляцию с правой стороны JOIN.

Алхимия поддерживает такую возможность с помощью `Select.lateral()`, который создает соответственно объект Lateral.

```
subq = (
    select(
        func.count(address_table.c.id).label("address_count"),
        address_table.c.email_address,
        address_table.c.user_id,
    )
    .where(user_table.c.id == address_table.c.user_id)
    .lateral()
```

```
)
stmt = (
    select(user_table.c.name, subq.c.address_count, subq.c.email_address)
    .join_from(user_table, subq)
    .order_by(user_table.c.id, subq.c.email_address)
)
print(stmt)
```

```
SELECT user_account.name, anon_1.address_count, anon_1.email_address
FROM user_account
JOIN LATERAL (SELECT count(address.id) AS address_count,
address.email_address AS email_address, address.user_id AS user_id
FROM address
WHERE user_account.id = address.user_id) AS anon_1
ON user_account.id = anon_1.user_id
ORDER BY user_account.id, anon_1.email_address
```

В приведенном примере правая сторона JOIN, являясь субзапросом, который кореллирует к таблице user_account является частью левой стороны.

При использовании Select.lateral(), поведение Select.correlate() и Select.correlate_except() абсолютно совпадает с Lateral конструкцией.

UNION, UNION ALL и другие операции множеств

В SQL выражения SELECT могут быть объединены вместе с помощью UNION или UNION ALL операций, которые создают множество из строк, выбранных одним или множеством выражений вместе. Существуют также и другие операции множеств, вроде INTERSECT или EXCEPT.

Алхимия позволяет делать операции такого рода с помощью функций union(), intersect() и except_(), а также их all варианты: union_all(), intersect_all() и except_all(). Все эти функции принимают определенное количество запросов SELECT.

Конструкция создаваемая этими функциями является объектом CompoundSelect, с ней можно работать похоже на Select, за исключением более скудного набора методов.

```
from sqlalchemy import union_all
stmt1 = select(user_table).where(user_table.c.name == "sandy")
stmt2 = select(user_table).where(user_table.c.name == "spongebob")
u = union_all(stmt1, stmt2)
with engine.connect() as conn:
```

```
    result = conn.execute(u)
    print(result.all())
```

```
BEGIN (implicit)
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?
UNION ALL SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?

[generated in ...] ('sandy', 'spongebob')
[(2, 'sandy', 'Sandy Cheeks'), (1, 'spongebob', 'Spongebob Squarepants')]

ROLLBACK
```

Выбор сущностей ORM из UNION

Предыдущие примеры продемонстрировали как мы можем создавать UNION из двух таблиц. Если же мы хотим использовать UNION или другие операции множеств для выбора строк, которые необходимо определить из объектов ORM, то нам потребуется сделать кое-что еще. Есть всего два способа как мы можем такое сделать. В обоих из них мы сначала создаем select() или CompoundSelect объект, представляющий SELECT / UNION или другие выражения, которые мы хотим выполнить.

Первый способ:

```
stmt1 = select(User).where(User.name == "sandy")
stmt2 = select(User).where(User.name == "spongebob")
u = union_all(stmt1, stmt2)
```

Для простых SELECT запросов с UNION, которые еще не включают внутри себя субзапросы, мы можем сделать всё то же самое через Select.from_statement(). В данном подходе, UNION представляет весь запрос, без дополнительных критериев:

```
orm_stmt = select(User).from_statement(u)
with Session(engine) as session:
    for obj in session.execute(orm_stmt).scalars():
        print(obj)
```

```
BEGIN (implicit)
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ? UNION ALL SELECT user_account.id, user_account.name, user_
account.fullname
FROM user_account
WHERE user_account.name = ?
[generated in ...] ('sandy', 'spongebob')
User(id=2, name='sandy', fullname='Sandy Cheeks')
User(id=1, name='spongebob', fullname='Spongebob Squarepants')
ROLLBACK
```

Использовать UNION или другую операцию множеств, в качестве компонента, связанного с сущностью, более гибким способом можно с помощью конструкции CompoundSelect и ее метода CompoundSelect.subquery(), которая затем ссылается на объекты ORM через функцию aliased(). Это работает ровно тем же способом, как и в случае с субзапросами ORM:

```
user_alias = aliased(User, u.subquery())
orm_stmt = select(user_alias).order_by(user_alias.id)
with Session(engine) as session:
    for obj in session.execute(orm_stmt).scalars():
        print(obj)
```

```
BEGIN (implicit)
SELECT anon_1.id, anon_1.name, anon_1.fullname
FROM (SELECT user_account.id AS id, user_account.name AS name, user_account.fullname A
S fullname
FROM user_account
WHERE user_account.name = ? UNION ALL SELECT user_account.id AS id, user_account.name
 AS name, user_account.fullname AS fullname
FROM user_account
WHERE user_account.name = ?) AS anon_1 ORDER BY anon_1.id
[generated in ...] ('sandy', 'spongebob')

User(id=1, name='spongebob', fullname='Spongebob Squarepants')
User(id=2, name='sandy', fullname='Sandy Cheeks')

ROLLBACK
```

Субзапрос EXISTS

В SQL ключевое слово EXISTS является оператором, которое использует скалярный субзапрос чтобы вернуть boolean значение true/false, показывающим может ли выражение SELECT вернуть хотя бы одну строчку. Алхимия включает объект Exists в ScalarSelect, который генерируется с помощью метода SelectBase.exists(). В следующем примере мы используем EXISTS чтобы понять есть ли хоть один адресс у пользователя из user_account:

```
subq = (
    select(func.count(address_table.c.id))
    .where(user_table.c.id == address_table.c.user_id)
    .group_by(address_table.c.user_id)
    .having(func.count(address_table.c.id) > 1)
).exists()
with engine.connect() as conn:
    result = conn.execute(select(user_table.c.name).where(subq))
    print(result.all())
```

```
BEGIN (implicit)
SELECT user_account.name
FROM user_account
WHERE EXISTS (SELECT count(address.id) AS count_1
FROM address
WHERE user_account.id = address.user_id GROUP BY address.user_id
HAVING count(address.id) > ?)
[...] (1,)
[('sandy',)]
ROLLBACK
```

Конструкция EXISTS чаще всего используется в качестве отрицания, например NOT EXISTS, так как такой подход предоставляет эффективную SQL форму поиска строк. Ниже мы выбираем имена пользователей, которые не имеют адресов электронной почты; обратите внимание на двоичный оператор отрицания (~), используемый во втором предложении WHERE:

```
subq = (
    select(address_table.c.id).where(user_table.c.id == address_table.c.user_id)
).exists()
with engine.connect() as conn:
    result = conn.execute(select(user_table.c.name).where(~subq))
    print(result.all())
```

```
BEGIN (implicit)
SELECT user_account.name
FROM user_account
```

```
WHERE NOT (EXISTS (SELECT address.id
FROM address
WHERE user_account.id = address.user_id))
[...] ()
[('patrick',)]
ROLLBACK
```

Работа с SQL функциями

Как уже было рассказано ранее, объект `func` служит нам в качестве фабрики для создания объектов Function, которые могут быть использованы в конструкциях наподобие select().

- функция count() - агрегатная функция, которая считает количество строк на вход:

```
>>> print(select(func.count()).select_from(user_table))

SELECT count(*) AS count_1
FROM user_account
```

- функция lower() переводит строчку на входе в нижний регистр

```
>>> print(select(func.lower("A String With Much UPPERCASE")))

SELECT lower(:lower_2) AS lower_1
```

- функция now() предоставляет текущие значения даты и времени

```
stmt = select(func.now())
with engine.connect() as conn:
    result = conn.execute(stmt)
    print(result.all())

BEGIN (implicit)
SELECT CURRENT_TIMESTAMP AS now_1
[...] ()
[(datetime.datetime(...),)]
ROLLBACK
```

Поскольку большинство баз данных предоставляют сотни, если не тысячи, разных функций, func пытается быть наиболее либеральным (насколько это

возможно и где это приемлемо). Любое имя переданное в качестве имени функции будет принято “за правду”:

```
>>> print(select(func.some_crazy_function(user_table.c.name, 17)))

SELECT some_crazy_function(user_account.name, :some_crazy_function_2) AS some_crazy_fu
nction_1
FROM user_account
```

В тоже время такие наиболее часто используемые функции как `count`, `now`, `max`, `concat` дополнительно проработаны под капотом алхимии, в том числе для более точного определения типа данных на входе, и при этом их поведение в разных диалектах может быть разным:

```
>>> from sqlalchemy.dialects import postgresql
>>> print(select(func.now()).compile(dialect=postgresql.dialect()))

SELECT now() AS now_1

>>> from sqlalchemy.dialects import oracle
>>> print(select(func.now()).compile(dialect=oracle.dialect()))

SELECT CURRENT_TIMESTAMP AS now_1 FROM DUAL
```

Тип данных, возвращаемых из функции

Поскольку функции являются выражениями столбцов, они также имеют типы данных SQL, которые описывают тип данных сгенерированного SQL-выражения. Стоит различать типы данных в SQL и в Python.

Доступ к возвращаемому типу любой функции SQL можно получить, как правило, в целях отладки, сославшись на атрибут Function.type

```
>>> func.now().type
DateTime()
```

Способность получить эту информацию скорее важна при использовании выражения функции в контексте более крупного выражения. Например, математические операторы будут работать лучше, когда тип данных выражения является чем-то вроде Integer или Numeric.

Тип возврата SQL функции также может быть полезным при выполнении инструкции и возврате строк в тех случаях, когда SQLAlchemy должна применить обработку результата. Ярким примером этого являются функции, связанные с датой и временем, где DateTime в SQLAlchemy и связанные с ним типы данных берут на себя роль преобразования строковых значений в объекты Python datetime().

Типы возвращаемых значений у встроенных функций

Для наиболее простых агргеатных функций, таких как count, max, min, now и concat SQLAlchemy может определить возвращаемый тип на основе передаваемых данных без запроса в базу данных:

```
>>> m1 = func.max(Column("some_int", Integer))
>>> m1.type
Integer()

>>> m2 = func.max(Column("some_str", String))
>>> m2.type
String()
```

Тоже самое мы можем увидеть и у остальных функций, например `concat` всегда будет возвращать String(), а `now` - DateTime().

CAST в SQLAlchemy

В SQL нам часто требуется обозначить определенный тип данных для того или иного выражения, для этого мы можем использовать, например, ключевое слово CAST. Алхимия позволяет реализовать тот же функционал с помощью функции `cast()`. Данная функция принимает на вход выражение и тип данных, а затем генерирует необходимый запрос для исполнения, например, в следующем примере мы получим целочисленное id в виде значения VARCHAR:

```
>>> from sqlalchemy import cast
>>> stmt = select(cast(user_table.c.id, String))
>>> with engine.connect() as conn:
...     result = conn.execute(stmt)
...     result.all()
```

```
BEGIN (implicit)
SELECT CAST(user_account.id AS VARCHAR) AS id
FROM user_account

[...] ()
[('1',), ('2',), ('3',)]

ROLLBACK
```

Но мощь данной функции сводится не только к организации SQL скрипта, мы также можем привести какие-то данные к тому или иному типу данных и взаимодействовать с результатом как с Python объектом:

```
>>> from sqlalchemy import JSON
>>> print(cast("{'a': 'b'}", JSON)["a"])

CAST(:param_1 AS JSON)[:param_2]
```

Использование выражений UPDATE и DELETE

После того как мы поговорили про Insert, мы обсудили использование Select, теперь наступило время конструкций Update и Delete, которые отвечают за изменение уже имеющихся данных в базе данных.

update() в SQLAlchemy

Функция `update()` подобно уже известным нам select() и insert() создает новый экземпляр класса Update, который представляет выражение UPDATE в SQL.

Самое обычное использование UPDATE в Core выглядит следующим образом:

```
from sqlalchemy import update
stmt = (
    update(user_table)
    .where(user_table.c.name == "patrick")
```

```
    .values(fullname="Patrick the Star")
)
print(stmt)
```

```
UPDATE user_account SET fullname=:fullname WHERE user_account.name = :name_1
```

Использование update() очень хорошо напоминает нам конструкцию UPDATE… WHERE из обычного SQL лишь с одним отличием - явного отделения .values(). Данная функция отвечает за SET в типичном Update.

Мы можем также выполнить UPDATE скрипт в “bulk” режиме (множественной записи):

```
>>> from sqlalchemy import bindparam
>>> stmt = (
...     update(user_table)
...     .where(user_table.c.name == bindparam("oldname"))
...     .values(name=bindparam("newname"))
... )
>>> with engine.begin() as conn:
...     conn.execute(
...         stmt,
...         [
...             {"oldname": "jack", "newname": "ed"},
...             {"oldname": "wendy", "newname": "mary"},
...             {"oldname": "jim", "newname": "jake"},
...         ],
...     )
```

```
BEGIN (implicit)
UPDATE user_account SET name=? WHERE user_account.name = ?

[...] [('ed', 'jack'), ('mary', 'wendy'), ('jake', 'jim')]

<sqlalchemy.engine.cursor.CursorResult object at 0x...>
COMMIT
```

Обратите внимание, что для параметризации запроса мы используем функцию bindparam().

Кореллированный UPDATE

UPDATE может использовать строки из других таблиц с помощью кореллированного субзапроса.

```
>>> scalar_subq = (
...     select(address_table.c.email_address)
...     .where(address_table.c.user_id == user_table.c.id)
...     .order_by(address_table.c.id)
...     .limit(1)
...     .scalar_subquery()
... )
>>> update_stmt = update(user_table).values(fullname=scalar_subq)
>>> print(update_stmt)
```

```
UPDATE user_account SET fullname=(SELECT address.email_address
FROM address
WHERE address.user_id = user_account.id ORDER BY address.id
LIMIT :param_1)
```

UPDATE…FROM

Некоторые СУБД вроде PostgreSQL и MySQL поддерживают синтаксис “UPDATE…FROM” где дополнительные таблицы могут быть указаны непосредственно в FROM.

```
>>> update_stmt = (
...     update(user_table)
...     .where(user_table.c.id == address_table.c.user_id)
...     .where(address_table.c.email_address == "patrick@aol.com")
...     .values(fullname="Pat")
... )
>>> print(update_stmt)
```

```
UPDATE user_account SET fullname=:fullname FROM address
WHERE user_account.id = address.user_id AND address.email_address = :email_address_1
```

Также имеет место специфичный синтаксис для MySQL который позволяет осуществлять UPDATE сразу в несколько таблиц:

```
>>> update_stmt = (
...     update(user_table)
...     .where(user_table.c.id == address_table.c.user_id)
...     .where(address_table.c.email_address == "patrick@aol.com")
...     .values(
```

```
...         {
...             user_table.c.fullname: "Pat",
...             address_table.c.email_address: "pat@aol.com",
...         }
...     )
... )
>>> from sqlalchemy.dialects import mysql
>>> print(update_stmt.compile(dialect=mysql.dialect()))
```

```
UPDATE user_account, address
SET address.email_address=%s, user_account.fullname=%s
WHERE user_account.id = address.user_id AND address.email_address = %s
```

Другой “эксклюзив” MySQL заключается в том, что порядок параметров в предложении SET UPDATE фактически влияет на вычисление каждого выражения, что может вызвать непредсказуемое поведение. Поэтому для такого сценария следует использовать метод Update.ordered_values(), который принимает последовательность кортежей, чтобы их порядок можно было контролировать:

```
>>> update_stmt = update(some_table).ordered_values(
...     (some_table.c.y, 20), (some_table.c.x, some_table.c.y + 10)
... )
>>> print(update_stmt)
UPDATE some_table SET y=:y, x=(some_table.y + :y_1)
```

delete() в SQLAlchemy

Много говорить про delete() нет смысла, так как его использование донельзя похоже на уже разобранный update():

```
>>> from sqlalchemy import delete
>>> stmt = delete(user_table).where(user_table.c.name == "patrick")
>>> print(stmt)
```

```
DELETE FROM user_account WHERE user_account.name = :name_1
```

Единственное, что стоит обсудить, так это удаление записей сразу в нескольких таблицах:

```
>>> delete_stmt = (
...     delete(user_table)
...     .where(user_table.c.id == address_table.c.user_id)
...     .where(address_table.c.email_address == "patrick@aol.com")
... )
>>> from sqlalchemy.dialects import mysql
>>> print(delete_stmt.compile(dialect=mysql.dialect()))
```

```
DELETE FROM user_account USING user_account, address
WHERE user_account.id = address.user_id AND address.email_address = %s
```

⚠️ Обратите внимание, что далеко не все СУБД поддерживают такую возможность. На данный момент такой синтаксис поддерживает только MySQL.

Количество затронутых строк в UPDATE/DELETE запросах

Мы можем получить количество строк, которые были изменены\удалены в ходе запросов UPDATE или DELETE:

```
>>> with engine.begin() as conn:
...     result = conn.execute(
...         update(user_table)
...         .values(fullname="Patrick McStar")
...         .where(user_table.c.name == "patrick")
...     )
...     print(result.rowcount)
```

```
BEGIN (implicit)
UPDATE user_account SET fullname=? WHERE user_account.name = ?
[...] ('Patrick McStar', 'patrick')

1

COMMIT
```

Использование RETURNING с UPDATE/DELETE

Как и Insert, Update\Delete поддерживают выражение RETURNING, которое добавляется функциями `Update.returning()` и `Delete.returning()`. Данная конструкция позволяет передать в качестве аргументов ту или иную колонку и вернуть ее значение после выполнения скрипта для задействованных строк.

```
>>> update_stmt = (
...     update(user_table)
...     .where(user_table.c.name == "patrick")
...     .values(fullname="Patrick the Star")
...     .returning(user_table.c.id, user_table.c.name)
... )
>>> print(update_stmt)
```

```
UPDATE user_account SET fullname=:fullname
WHERE user_account.name = :name_1
RETURNING user_account.id, user_account.name
```

```
>>> delete_stmt = (
...     delete(user_table)
...     .where(user_table.c.name == "patrick")
...     .returning(user_table.c.id, user_table.c.name)
... )
>>> print(delete_stmt)
```

```
DELETE FROM user_account
WHERE user_account.name = :name_1
RETURNING user_account.id, user_account.name
```

6. Манипуляции с данными в ORM

В прошлом разделе мы сосредоточили внимание на языке выражений SQL из части Core, теперь же, в порядке продолжения исследования частей алхимии, мы приступим к обзору жизненного цикла Session и как она взаимодействует с теми или иными конструкциями.

Вставка строк с помощью “объединения работ”

Мы уже говорили в первых разделах о том, что ORM в Алхимии использует такой паттерн как “объединение работ” (unit of work). Теперь же нам необходимо понять как именно этот паттерн используется в ORM части.

Сессия (Session) ответственна за конструирование INSERT запросов и их отправку в базу данных. Для нас же важно уметь **добавлять объекты в сессию**, а также переносить объекты в транзакцию (**или флашить их**). Об остальном позаботится алхимия.

Инстанции моделей представляют строки

В предыдущем разделе мы научились делать INSERT используя Python словари для передачи той информации, которую мы хотим добавить в базу, с такой же целью в ORM мы используем модели, или если быть точнее, их инстанции (мы уже обсуждали модели ранее). Модели User и Adress служат нам как место, куда мы можем записать соответствующую информацию о строках в базе данных. Ниже мы создадим два объекта User, каждый из которых представляет потенциальную строчку в бд, которые мы хотим за-INSERT-ить:

```
squidward = User(name="squidward", fullname="Squidward Tentacles")
krabs = User(name="ehkrabs", fullname="Eugene H. Krabs")
```

Модели по умолчанию переопределяют метод __init__(), который, как известно, отвечает за инициализацию инстанций класса, поэтому мы можем просто передать в качестве именованных аргументов те или иные столбцы в наших классах.

Обратите внимание, что мы не передаем никакое значение для столбца id при создании модели. Дело в том, что это значение автоинкрементируемое, и является внешним ключем. Иначе говоря, это значение должна определять база данных самостоятельно. Давайте попробуем запринтить только что созданный объект и посмотреть на то, какое же будет значение id у него:

```
>>> squidward
User(id=None, name='squidward', fullname='Squidward Tentacles')
```

Как вы видите, значение является None. Столбец еще не имеет никакого значения, поскольку объект модели еще не был записан в БД. Данное состояние модели, когда она еще не была добавлена в сессию, называется transient (с англ. “временный”) - то есть она еще не является полноценной записью, а только лишь потенциальной.

Добавление объектов в сессию

Чтобы показать процесс добавления в сессию по шагам, мы создадим сессию без использования контекстного менеджера:

```
session = Session(engine)
```

Теперь объекты могут быть добавлены с помощью такого метода как Session.add. Как только объект будет передан в этот метод он получит статус pending (с англ. “ожидающий”):

```
session.add(squidward)
session.add(krabs)
```

Теперь, мы можем получить коллекцию из всех объектов в данном состоянии:

```
>>> session.new
IdentitySet([User(id=None, name='squidward', fullname='Squidward Tentacles'), User(id=
None, name='ehkrabs', fullname='Eugene H. Krabs')])
```

Мы видим коллекцию, которая называется IdentitySet, она по своей сути напоминает Python словарь, который использует для проверки идентичности объектов встроенную функцию id(), и чуть реже hash(). Как вы уже понимаете, только один идентичный объект может находится в сессии.

Флашинг

Вы уже знаете о таком паттерне, как объединение работ. Вообще говоря, это значит ровным счетом то, что любые изменения, которые вы делаете с вашими объектами, будут лишь записаны в набор отложенных изменений, а не отправлены напрямую в БД. Таким образом, даже если вы измените то или иное значение у объекта-модели, оно не будет изменено до тех пор, пока вы не вызовите `Session.flush()`. Это позволяет принимать лучшие решения о том, как SQL конструкции должны быть преобразованы перед отправкой в транзакцию на основе набора соответствующих отложенных изменений.

Мы можем продемонстрировать процесс флаша вызвав вручную метод `.flush()`:

```
>>> session.flush()
BEGIN (implicit)
INSERT INTO user_account (name, fullname) VALUES (?, ?), (?, ?) RETURNING id
[...] ('squidward', 'Squidward Tentacles', 'ehkrabs', 'Eugene H. Krabs')
```

В данном примере мы можем наблюдать как сессия сначала открывает новую транзакцию и лишь затем отправляет INSERT конструкцию для двух объектов. Транзакция при этом остается открытой до тех пор пока мы не вызовем один из следующих методов: `Session.commit()`, `Session.rollback()` или `Session.close()`.

Мы можем отправлять изменения в транзакцию вручную, вызывая метод `.flush()`, либо использовать такое поведение сессии как автофлаш (autoflush). В таком случае флаш будет вызываться каждый раз вместе с `Session.commit()`.

Автогенерируемые атрибуты

В начале данного раздела мы уже видели, что если объект был создан, но не отправлен в транзакцию, то автогенерируемые поля, вроде автоинкремента, будут None.

Как только мы INSERT-тим объекты, они переходят в состояние persistent (с англ. “постоянный”), где они уже представляют не потенциальные строки, а вполне конкретные. Вместе с этим, атрибуты моделей обновляются и теперь значение id уже не None:

```
>>> squidward.id
4
>>> krabs.id
5
```

❓ Почему ORM просто не объединит две конструкции INSERT в один executemany()? Как мы увидим в следующей главе, сессии при флаше всегда нужно знать первичные ключи вновь созданных объектов. К сожалению, далеко не все драйверы DBAPI способны вернуть их при executemany(), а требуют вызывать INSERT для каждой строчки отдельно. Всё же некоторые драйверы баз данных, например, psycopg2, могут вставлять несколько строк одновременно, сохраняя при этом возможность получить значения первичного ключа для всех строк.

Возвращение объектов по первичному ключу из IdentityMap

Идентификатор первичного ключа объектов имеет важное значение для сессии, так как объекты теперь связаны с этим идентификатором в памяти с помощью IdentityMap. IdentityMap - это хранилище в памяти, которое связывает все объекты, загруженные в настоящее время в памяти, с их идентификатором первичного ключа. Мы можем наблюдать это, получив один из вышеперечисленных объектов с помощью метода `Session.get()`, который вернет запись с IdentityMap, если она присутствует или же выполнит соответствующий SELECT:

```
>>> some_squidward = session.get(User, 4)
>>> some_squidward
```

```
User(id=4, name='squidward', fullname='Squidward Tentacles')
```

Стоит отметить, что поскольку IdentityMap основывается на уникальных инстанциях, она не просто создает новый объект, а возвращает ссылку на уже существующий:

```
>>> some_squidward is squidward
True
```

Сохранение транзакций (COMMIT)

Теперь, когда мы уже добавили объекты в сессию, мы можем сохранить текущие изменения в транзакции с помощью COMMIT:

```
>>> session.commit()
COMMIT
```

Коммит не закрывает транзакцию, но переносит все изменения из нее в основную базу данных.

⚠️ Необходимо сказать, что атрибуты объектов, которые были за-COMMIT-ены теперь считаются устарелыми (expired), при этом если мы обратимся к ним, то сессия откроет новую транзакцию и перезагрузит их заного. Это может привести к некоторым проблемам, например, если вы захотите продолжить работу с объектом (который уже находится в состоянии detached), при этом не будет открыто ни одной сессии с помощью которой вы смогли бы перезагрузить объект, то это может привести к проблемам “отсоединенной инстанции”, то есть когда существующая инстанция не может быть сопоставлена с соответствующей ей строчкой в таблице БД. Это поведение контролируется параметром Session.expire_on_commit.

Обновление объектов ORM используя паттерн объединения работ

В предыдущем разделе туториала мы уже говорили про Update и Delete конструкции. В ORM они используются целиком и полностью при работе с сессиями, что означает, что вам нет необходимости осуществлять подобные запросы самостоятельно.

Предположим, что мы загрузили объект User с username=’sandy’ в транзакцию:

```
>>> sandy = session.execute(select(User).filter_by(name="sandy")).scalar_one()
BEGIN (implicit)
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?
[...] ('sandy',)
```

В данном случае мы используем .filter_by() для реализации WHERE вместе с методом .scalar_one(), который вернет строго один объект из результата.

Как и было сказано ранее, объект User представляет строку в таблице БД в условиях текущей транзакции.

```
>>> sandy
User(id=2, name='sandy', fullname='Sandy Cheeks')
```

Если мы изменим тот или иной атрибут объекта, сессия отследит это изменение и объект появится в специальной коллекции под именем dirty (”грязный”):

```
>>> sandy in session.dirty
True
```

Когда мы сделаем флаш, сессия сама исполнит необходимые UPDATE запросы на обновление соответствующих значений в БД. Флаш происходит автоматически когда мы исполняем SELECT запрос при включенном параметре autoflush.

```
>>> sandy_fullname = session.execute(select(User.fullname).where(User.id == 2)).scalar
_one()
UPDATE user_account SET fullname=? WHERE user_account.id = ?
[...] ('Sandy Squirrel', 2)
SELECT user_account.fullname
FROM user_account
WHERE user_account.id = ?
[...] (2,)
```

```
>>> print(sandy_fullname)
Sandy Squirrel
```

Мы можем видеть как вроде бы мы всего лишь запросили SELECT, но под капотом сессия сразу же обновляет объект User:

```
>>> sandy in session.dirty
False
```

В списке dirty объекта также нет. Однако не спешите. Несмотря на то, что мы вроде как обновили данные, они находятся только лишь в транзакции, а не постоянном хранилище БД. Чтобы исполнить транзакцию, ее необходимо закоммитить.

Удаление объектов ORM

Чтобы завершить обзор основ работы с операциями в ORM, мы должны рассмотреть также как объект может быть помечен к удалению с помощью `Session.delete()`. Давайте сначала достанем объект из базы данных:

```
>>> patrick = session.get(User, 3)
SELECT user_account.id AS user_account_id, user_account.name AS user_account_name,
user_account.fullname AS user_account_fullname
FROM user_account
WHERE user_account.id = ?
[...] (3,)
```

Если мы хотим отметить объект к удалению нам необходимо вызвать соответствующий метод перед флашем:

```
>>> session.delete(patrick)
```

Текущее поведение ORM заключается в том, что объект останется в сессии до флаша, который как уже было сказано ранее будет вызван при любом запросе в БД:

```
>>> session.execute(select(User).where(User.name == "patrick")).first()
SELECT address.id AS address_id, address.email_address AS address_email_address,
```

```
address.user_id AS address_user_id
FROM address
WHERE ? = address.user_id
[...] (3,)
DELETE FROM user_account WHERE user_account.id = ?
[...] (3,)
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?
[...] ('patrick',)
```

Как вы можете заметить, несмотря на то, что мы всего лишь попросили сделать простой SELECT, алхимия также проводит операцию DELETE. Но помимо этого алхимия также отправила запрос SELECT на таблицу адресов с поиском тех строк, которые потенциально могут быть связаны с данным пользователем. Данное поведение является настраиваемым и называется каскадами, о них мы поговорим позже чуть подробнее.

Обратите внимание, что теперь нашего объекта нет в сессии:

```
>>> patrick in session
False
```

Множественный INSERT/ UPSERT/UPDATE и DELETE

Паттерн “объединения работ” позволяет сполна интегрировать DML (Data Manipulation Language) с помощью Python объектов. Как только объект был добавлен в сессию, процесс объединения работ незаметно отправляет INSERT/UPDATE/DELETE в БД по мере создания и изменения атрибутов на наших объектах.

Однако, сессия в ORM также имеет способность отправлять команды в БД и без использования ORM объектов, а с помощью передачи множества значений, которые должны быть записаны, обновлены или удалены. Это может быть полезно для множественных операций (bulk). Почему мы не можем использовать для этой цели те же объекты? Всё дело в том, что bulk-операции обычно используются при необходимости вставки(или обновления/удаления) огромного количества строк (тысяча, сто тысяч, миллион, миллиард и т.д.). ORM будет тратить слишком много времени на преобразование каждого объекта в строку и наоборот. Это может создать серьезнейшее замедление при

подобных процессах, поэтому от основных идей ORM в данном случае будет нужно отказаться.

К счастью, ничего нового в данном случае не будет. Вы можете делать ровным счетом то же самое, что и в прошлом разделе туториала только лишь заменив Connection на Session. Сессия, как вы уже знаете, является каскадой над соединением.

Возврат или ROLLBACK

Сессия также предоставляет возможность отменить транзакцию с помощью метода Session.rollback(). Этот метод просто отправит ROLLBACK в БД для текущей транзакции, но на деле будут и другие последствия, в том числе среди объектов в сессии. В прошлом примере мы использовали объект User. Мы меняли ему атрибут fullname и флашили это изменение, но теперь, допустим, мы передумали и хотим откатить его. Откатить флаш не получится, да и это бессмысленно, зато отменить транзакцию - вполне.

```
>>> session.rollback()
ROLLBACK
```

Давайте посмотрим что уже произошло с нашим объектом:

```
>>> sandy.__dict__
{'_sa_instance_state': <sqlalchemy.orm.state.InstanceState object at 0x...>}
```

В данном случае мы имеем устаревшее состояние (expired), давайте попробуем обратиться к атрибуту fullname:

```
>>> sandy.fullname
BEGIN (implicit)
SELECT user_account.id AS user_account_id, user_account.name AS user_account_name,
user_account.fullname AS user_account_fullname
FROM user_account
WHERE user_account.id = ?
[...] (2,)
'Sandy Cheeks'
```

ORM сразу же обратилась в БД за актуальной информацией и теперь посмотрим что стало с объектом:

```
>>> sandy.__dict__
{'_sa_instance_state': <sqlalchemy.orm.state.InstanceState object at 0x...>,
 'id': 2, 'name': 'sandy', 'fullname': 'Sandy Cheeks'}
```

Теперь наш объект загрузился полностью и его состояние - persistent. Если помните мы также удаляли объект User, зафлашили это изменение, так вот после ROLLBACK мы снова можем его увидеть в сессии:

```
>>> patrick in session
True
```

И более того:

```
>>> session.execute(select(User).where(User.name == "patrick")).scalar_one() is patric
k
SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.name = ?
[...] ('patrick',)
True
```

Как вы помните алхимия использует механизм IdentityMap, который сопоставляет первичному ключу объект Python. Несмотря на то, что мы пытались получить объект из БД, он всё еще ссылается на наш старый объект, созданный ранее.

Закрытие сессии

Теперь необходимо рассмотреть важный элемент при работе с сессиями - их закрытие. Так как мы работали с ней без контекстного менеджера, это ни в коем случае нельзя забыть сделать. Если же вы используете контекстный менеджер, то для вас будет всё проще, так как за вас об этом позаботится он.

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

Закрытие сессии:

- **Высвобождает все используемые соединения** обратно в пул соединений машины, отменяет (ROLLBACK) все актуальные транзакции. Это означает, что если мы использовали сессию только для чтение чего-либо, то нам нет необходимости делать ROLLBACK самостоятельно, сессия позаботится об этом сама при закрытии.
- **Вычищает все объекты из сессии.** Это означает, что все объекты, которые были загружены сессией, перейдут в состояние detached. В частности, если мы захотим обратиться к любому атрибуту модели, то получим ошибку `DetachedInstanceError`. Но “отсоединенные” объекты всё же могут быть переприсоединены к сессии (той же либо другой) с помощью уже знакомого вам метода Session.add().

7. Работа со связями

В данном разделе мы обсудим наиболее важную для ORM концепцию, как связи (relationship). Мы обсудим как ORM взаимодействует с моделями, ссылающимися на другие (то есть находящиеся в связи с ними). А также как мы можем создавать и настраивать их.

Чтобы описать основную идею связи и relationship(), для начала давайте посмотрим на уже созданные нами модели, опуская все остальные столбцы для простоты:

```
from sqlalchemy.orm import Mapped
from sqlalchemy.orm import relationship


class User(Base):
    __tablename__ = "user_account"

    # ... mapped_column() mappings

    addresses: Mapped[List["Address"]] = relationship(back_populates="user")


class Address(Base):
    __tablename__ = "address"

    # ... mapped_column() mappings

    user: Mapped["User"] = relationship(back_populates="addresses")
```

Наш класс User теперь имеет атрибут addresses, в свою очередь Address теперь имеет атрибут user. Конструкция relationship() в связке с Mapped аннотацией может быть использована для работы со связью между User и Address. Поскольку таблица адресов имеет ограничение внешнего ключа (ForeignKeyConstraint), которое ссылается на таблицу user_account, relationship() может автоматичсески определить что это связь типа один-ко-многим к User.adresses из Adress. Или иначе говоря, несколько адресов могут быть связаны с одним пользователем.

Любые связи один-ко-многим могут быть слегкостью инвертированы до многие-к-одному, это настраивается с помощью параметра back_populates в relationship().

Сохраняющиеся и загружающиеся связи

Начнем разбираться с тем как работает relationship(). Если мы создадим новый объект User, то заметим, что атрибуту addresses теперь имеет тип list:

```
>>> u1 = User(name="pkrabs", fullname="Pearl Krabs")
>>> u1.addresses
[]
```

Этот объект на самом деле не является списком в чистом виде, это его версия от Алхимии, которая хотя и полностью повторяет его поведение, но всё же также отслеживает изменения, которые с ним происходят. Этот список появится автоматически при любой попытке обратиться к этому атрибуту. Поскольку u1 до сих пор transient и список addressess не был изменен, мы еще не имеем ассоциации с записью в БД.

Давайте теперь добавим в наш список объект Address:

```
>>> a1 = Address(email_address="pearl.krabs@gmail.com")
>>> u1.addresses.append(a1)
```

Теперь, запринтим этот список:

```
>>> u1.addresses
[Address(id=None, email_address='pearl.krabs@gmail.com')]
```

Так как мы еще не добавили объект User в сессию, то и Address тоже является лишь потенциальной строкой в БД. Но при этом мы уже можем увидеть, что объект User находится в Address:

```
>>> a1.user
User(id=None, name='pkrabs', fullname='Pearl Krabs')
```

Это происходит поскольку мы определили параметр back_populates для связи между моделями. Обратите внимание, что еще один способ связать два объекта, это не добавлять его в список у объекта User, а, наоборот, указать объект User при создании Address:

```
>>> a2 = Address(email_address="pearl@aol.com", user=u1)
>>> u1.addresses
```

```
[Address(id=None, email_address='pearl.krabs@gmail.com'), Address(id=None, email_addre
ss='pearl@aol.com')]
```

Ну и конечно мы можем сделать тоже самое следующим образом:

```
>>> a2.user = u1
```

Каскадное добавление в сессию

Теперь у нас есть объект User и два объекта Address, которые ассоциированы двухсторонней структурой в памяти, но как было сказано ранее в предыдущем разделе туториала, эти объекты всё еще находятся в состоянии transient. Мы можем добавить их в сессию уже знакомым нам методом:

```
>>> session.add(u1)
>>> u1 in session
True
>>> a1 in session
True
>>> a2 in session
True
```

Обратите внимание, что мы добавили лишь объект User, но каскадно в сессию добавились и ассоциированные с ним объекты Address. Это поведение определяется каскадой **save-update** (определена по умолчанию для relationship()).

Теперь все три объекта находятся в состоянии ожидания (pending), что означает что они уже готовы к INSERT, но он еще не был вызван. Обратите внимание, что поскольку для них также еще не были определены первичные ключи, то и внешние ключи тоже None:

```
>>> print(u1.id)
None
>>> print(a1.user_id)
None
```

После коммита транзакции все объекты будут вставлены в соответствующие таблицы, а мы получим их первичные и внешние ключи. Таким образом, алхимия самостоятельно без вашего участия создаст связь на уровне БД.

```
>>> session.commit()
INSERT INTO user_account (name, fullname) VALUES (?, ?)
[...] ('pkrabs', 'Pearl Krabs')
INSERT INTO address (email_address, user_id) VALUES (?, ?), (?, ?) RETURNING id
[...] ('pearl.krabs@gmail.com', 6, 'pearl@aol.com', 6)
COMMIT
```

Загрузка связей

Мы использовали Session.commit() чтобы отправить в БД команду COMMIT для сохранения транзакции, но теперь, как вы помните, наши объекты имеют состояние expired. Как только мы обратимся к атрибуту нашей модели, то увидим как алхимия сразу же обратится к БД за актуальной информацией:

```
>>> u1.id
BEGIN (implicit)
SELECT user_account.id AS user_account_id, user_account.name AS user_account_name,
user_account.fullname AS user_account_fullname
FROM user_account
WHERE user_account.id = ?
[...] (6,)
6
```

Но обратите внимание, что несмотря на загрузку объекта User, таблица адресов никак затронута не была. Это определяется поведением загрузки, которое называется “ленивое” или **lazy load**. То есть алхимия не будет загружать связанные объекты до первого требования.

```
>>> u1.addresses
SELECT address.id AS address_id, address.email_address AS address_email_address,
address.user_id AS address_user_id
FROM address
WHERE ? = address.user_id
[...] (6,)
[Address(id=4, email_address='pearl.krabs@gmail.com'), Address(id=5, email_address='pe
arl@aol.com')]
```

И User и Address теперь постоянны и имеют конкретную связь с БД. По этой причине в следующей попытке обращения к атрибутам алхимия уже не будет обращаться к БД:

```
>>> u1.addresses
[Address(id=4, email_address='pearl.krabs@gmail.com'), Address(id=5, email_address='pe
arl@aol.com')]
```

Поскольку объекты адресов уже были загружены и добавлены в уже известную нам коллекцию IdentityMap, обращаясь к ним напрямую они уже будут загружены:

```
>>> a1
Address(id=4, email_address='pearl.krabs@gmail.com')
>>> a2
Address(id=5, email_address='pearl@aol.com')
```

То есть, несмотря на то, что объекты a1 и a2 уже были добавлены в БД, стали устарелыми, после загрузки u1 они вновь стали актуальными и мы можем продолжить работу с ними.

Такое поведение определяется стратегией загрузки и они будут подробнее обсуждаться позднее.

Использованией связей в запросах

Ранее мы обсуждали поведение конструкции relationship() при работе с инстанциями моделей. А теперь мы поговорим о поведении relationship() при работе не с инстанциями, а именно классами моделей.

Использование связей в JOIN

Уже известные вам методы `Select.join()` и `Select.join_from()` используются для построения JOIN запросов SQL. Чтобы явно описать КАК мы должны объединять несколько таблиц алхимия дает возможность явно указать конструкцию ON, основываясь на внешнем ключе одной из таблиц.

При использовании ORM, мы можем не указывать конструкцию ON для уточнения запроса:

```
>>> print(select(Address.email_address).select_from(User).join(User.addresses))
```

```
SELECT address.email_address
FROM user_account JOIN address ON user_account.id = address.user_id
```

Однако конструкция ON формируется не на основе relationship(), а за счет внешних ключей, которые были созданы ранее, как и в случае Core.

Оператор WHERE при работе со связями

> Данная часть была импортирована из ORM Querying Guide, поскольку ее не было в основном тексте туториала

Помимо примитивных запросов, которые напоминают обычные конструкции WHERE из Core, relationship() открывает для нас новые возможности.

Конструкция EXISTS может быть использована в алхиими по-новому, с помощью таких методов как `has()` и `any()`. Для связей один-ко-многим, например, User.addresses, конструкция EXISTS может быть использована следующим образом:

```
>>> stmt = select(User.fullname).where(
...     User.addresses.any(Address.email_address == "squirrel@squirrelpower.org")
... )
>>> session.execute(stmt).all()

SELECT user_account.fullname
FROM user_account
WHERE EXISTS (SELECT 1
FROM address
WHERE user_account.id = address.user_id AND address.email_address = ?)
[...] ('squirrel@squirrelpower.org',)
[('Sandy Cheeks',)]
```

В том числе мы можем использовать оператор байтового инвертирования (~) для построения конструкций NOT EXISTS:

```
>>> stmt = select(User.fullname).where(~User.addresses.any())
>>> session.execute(stmt).all()
SELECT user_account.fullname
FROM user_account
WHERE NOT (EXISTS (SELECT 1
FROM address
WHERE user_account.id = address.user_id))
[...] ()
[('Eugene H. Krabs',)]
```

Метод has() может быть использован похожим на any() образом, но он дает возможность создать конструкцию EXISTS для фильтрования строк:

```
>>> stmt = select(Address.email_address).where(Address.user.has(User.name == "sandy"))
>>> session.execute(stmt).all()

SELECT address.email_address
FROM address
WHERE EXISTS (SELECT 1
FROM user_account
WHERE user_account.id = address.user_id AND user_account.name = ?)
[...] ('sandy',)
[('sandy@sqlalchemy.org',), ('squirrel@squirrelpower.org',)]
```

Из интересного также стоит показать метод contains():

```
>>> address_obj = session.get(Address, 1)
SELECT ...
>>> print(select(User).where(User.addresses.contains(address_obj)))

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account
WHERE user_account.id = :param_1
```

Стратегии загрузки

Ранее мы уже обсуждали одну из стратегий загрузки - ленивая (lazy load). Она заключается в том, что тот или иной атрибут модели, представляющий связанные объекты, не будет загружен до тех пор пока не потребуется именно он.

Ленивая загрузка - наиболее известная техника для ORM, но при этом является в то же время наиболее спорной. Когда несколько десятков ORM-объектов в памяти ссылаются на несколько незагруженных атрибутов, рутинные манипуляции с этими объектами могут создать слишком много дополнительных запросов, которые могут складываться (проблема N+1), и, что еще хуже, они могут испускаться неявно. Эти неявные запросы могут быть не замечены, привести к ошибкам, а при использовании asyncio они вообще не будут работать.

❓ Проблема N+1 - наиболее распространенный побочный эффект при использовании ленивой загрузки. Она возникает тогда, когда вы пытаетесь проитерировать атрибуты модели, настроенные на ленивую загрузку, но в результате ORM начинает дополнительно загружать каждый член по-отдельности. Вместо того, чтобы отправить один запрос, ORM отправит N+1 запросов, которые могут создать большие задержки в исполнении программы. Эта проблема частично решается с помощью жадной загрузки (eager load).

Тем не менее, данная стратегия загрузки считается наиболее универсальной, и поэтому в алхимии она находится по-умолчанию при настройке связей relationship().

Вы можете без особых проблем посмотреть насколько эффективно вы используете lazy load. Для этого достаточно включить echo=True и посмотреть, есть ли у вас какие-либо множественные запросы, которые могут быть объединены в один. Если да, тогда стоит для данной связи настроить другие стратегии.

Стратегии загрузки могут быть определены при настройке relationship() либо же прямо в запросе:

```
for user_obj in session.execute(
    select(User).options(selectinload(User.addresses))
).scalars():
    user_obj.addresses
```

Кстати, в данном примере адреса нашего пользователя загружаются вместе с загрузкой и самого пользователя, то есть в одном SELECT, вместо того, чтобы сначала загрузить пользователя, и только затем адреса.

Мы можем определить стратегию загрузки и в самой связи:

```
from sqlalchemy.orm import Mapped
from sqlalchemy.orm import relationship


class User(Base):
    __tablename__ = "user_account"

    addresses: Mapped[List["Address"]] = relationship(
        back_populates="user", lazy="selectin"
    )
```

Каждая стратегия загрузки содержит определенную информацию о том как именно будет происходить загрузка дополнительных таблиц. Мы обсудим несколько широко используемых стратегий.

Стратегия загрузки Selectin

Наиболее полезная стратегия в современной алхимии это именно selectinload(). Она позволяет решить проблему N+1, поскольку загружает все связанные объекты в основном запросе. Однако делает она это не с помощью JOIN'ов, а с помощью нескольких SELECT.

```
>>> from sqlalchemy.orm import selectinload
>>> stmt = select(User).options(selectinload(User.addresses)).order_by(User.id)
>>> for row in session.execute(stmt):
...     print(
...         f"{row.User.name}  ({', '.join(a.email_address for a in row.User.addresse
s)})"
...     )

SELECT user_account.id, user_account.name, user_account.fullname
FROM user_account ORDER BY user_account.id
[...] ()

SELECT address.user_id AS address_user_id, address.id AS address_id,
address.email_address AS address_email_address
FROM address
WHERE address.user_id IN (?, ?, ?, ?, ?, ?)
[...] (1, 2, 3, 4, 5, 6)

spongebob  (spongebob@sqlalchemy.org)
sandy  (sandy@sqlalchemy.org, sandy@squirrelpower.org)
patrick  ()
squidward  ()
ehkrabs  ()
pkrabs  (pearl.krabs@gmail.com, pearl@aol.com)
```

Стратегия загрузки Joined

Данная стратегия является наиболее старой в алхимии и заключается она, как понятно из названия, в загрузке связанных строк с помощью JOIN конструкций.

Это лучшая стратегия для загрузки many-to-one связи, поскольку она требует лишь одну дополнительную колонку для полной загрузки связи практически в любом случае. Для улучшения эффективности, она также могжет принимать

параметр joinedload.innerjoin, тогда вместо OUTER JOIN будет использован INNER JOIN, к слову говоря он является лучшим выбором для загрузки адресов нашего пользователя:

```
>>> from sqlalchemy.orm import joinedload
>>> stmt = (
...     select(Address)
...     .options(joinedload(Address.user, innerjoin=True))
...     .order_by(Address.id)
... )
>>> for row in session.execute(stmt):
...     print(f"{row.Address.email_address} {row.Address.user.name}")

SELECT address.id, address.email_address, address.user_id, user_account_1.id AS id_1,
user_account_1.name, user_account_1.fullname
FROM address
JOIN user_account AS user_account_1 ON user_account_1.id = address.user_id
ORDER BY address.id
[...] ()
spongebob@sqlalchemy.org spongebob
sandy@sqlalchemy.org sandy
sandy@squirrelpower.org sandy
pearl.krabs@gmail.com pkrabs
pearl@aol.com pkrabs
```

joinedload() неплохо работает и для связей типа один-ко-многим, однако это приводит к более сложной конструкции запроса, в том числе с рекурсивным умножением (для каждого дочернего объекта будут JOIN’иться и его дочерние), поэтому его использование, должно оцениваться в каждом конкретном случае, порой лучше выбрать selectinload().

Явный JOIN и жадная загрузка (Eager load)

Если нам нужно загрузить адреса во время JOIN’а к пользователю, используя Select.join(), мы могли бы использовать этот запрос, чтобы также “жадно” загружать содержимое атрибута Address.user на каждый возвращенный объект Address. Это поведение достигается с помощью опции contains_eager(). Она очень похожа на joinedload() с одним лишь отличием: она предполагает, что мы сами настроили JOIN, и вместо этого указывает только на то, что дополнительные столбцы в предложении COLUMNS должны быть загружены в связанные атрибуты для каждого возвращаемого объекта.

```
>>> from sqlalchemy.orm import contains_eager
>>> stmt = (
```

```
...     select(Address)
...     .join(Address.user)
...     .where(User.name == "pkrabs")
...     .options(contains_eager(Address.user))
...     .order_by(Address.id)
... )
>>> for row in session.execute(stmt):
...     print(f"{row.Address.email_address} {row.Address.user.name}")

SELECT user_account.id, user_account.name, user_account.fullname,
address.id AS id_1, address.email_address, address.user_id
FROM address JOIN user_account ON user_account.id = address.user_id
WHERE user_account.name = ? ORDER BY address.id
[...] ('pkrabs',)
pearl.krabs@gmail.com pkrabs
pearl@aol.com pkrabs
```

В примере выше мы видим фильтрацию строк пользователей по имени, а также загрузку атрибута Address.user для каждого полученного объекта. Если бы мы использовали только joinedload(), то встретили бы более сложную конструкцию для JOIN:

```
>>> stmt = (
...     select(Address)
...     .join(Address.user)
...     .where(User.name == "pkrabs")
...     .options(joinedload(Address.user))
...     .order_by(Address.id)
... )
>>> print(stmt)

SELECT address.id, address.email_address, address.user_id,
user_account_1.id AS id_1, user_account_1.name, user_account_1.fullname
FROM address JOIN user_account ON user_account.id = address.user_id
LEFT OUTER JOIN user_account AS user_account_1 ON user_account_1.id = address.user_id
WHERE user_account.name = :name_1 ORDER BY address.id
```

Стратегия загрузки Raiseload

Стоит упомянуть еще одну стратегию - raiseload. Она используется для того, чтобы окончательно победить проблему N+1. Она имеет два варианта, которые настраиваются с помощью параметра `raiseload.sql_only`. Работает она таким образом, что выкидывает исключение при попытке загрузить тот или иной атрибут. При включенном параметре sql_only, raiseload будет блокировать только ту загрузку, которая требует SQL запроса, при выключенном - вообще любую.

```
>>> from sqlalchemy.orm import Mapped
>>> from sqlalchemy.orm import relationship


>>> class User(Base):
...     __tablename__ = "user_account"
...     id: Mapped[int] = mapped_column(primary_key=True)
...     addresses: Mapped[List["Address"]] = relationship(
...         back_populates="user", lazy="raise_on_sql"
...     )


>>> class Address(Base):
...     __tablename__ = "address"
...     id: Mapped[int] = mapped_column(primary_key=True)
...     user_id: Mapped[int] = mapped_column(ForeignKey("user_account.id"))
...     user: Mapped["User"] = relationship(back_populates="addresses", lazy="raise_on
_sql")
```

Как вы можете заметить, в данном случае sql_only меняются на raise_on_sql. Если мы вдруг где-то захотим загрузить атрибут addresses, то получим такой блок:

```
>>> u1 = session.execute(select(User)).scalars().first()
SELECT user_account.id FROM user_account
[...] ()
>>> u1.addresses
Traceback (most recent call last):
...
sqlalchemy.exc.InvalidRequestError: 'User.addresses' is not available due to lazy='rai
se_on_sql'
```

Вам, наверное, интересно, если же данная стратегия выкидывает исключение при любой загрузке атрибута, то как же нам загрузить его, если это вдруг необходимо? Всё очень просто, достаточно в запросе дополнительно вручную указать другую стратегию:

```
>>> u1 = (
...     session.execute(select(User).options(selectinload(User.addresses)))
...     .scalars()
...     .first()
... )

SELECT user_account.id
FROM user_account
[...] ()
SELECT address.user_id AS address_user_id, address.id AS address_id
FROM address
```

```
WHERE address.user_id IN (?, ?, ?, ?, ?, ?)
[...] (1, 2, 3, 4, 5, 6)
```

Послесловие

Поздравляю! Вы подошли к концу данного туториала. Теперь перед вами открываются двери в необъятный мир алхимии, а сами вы можете называть себя алхимиками. Материал данного туториала охватывает практически все основы, и даже не только “основы”, но и мастерские секреты (особенно в разделе Работа с данными).

Теперь, чтобы двигаться дальше, я оставлю пару полезных и важных ссылок:

Полноценная документация SQLAlchemy	https://docs.sqlalchemy.org/en/20/
Подробнее про совмещение алхимии с asyncio	https://docs.sqlalchemy.org/en/20/orm/extensions/asyncio.html
Подробнее про диалект PostgreSQL	https://docs.sqlalchemy.org/en/20/dialects/postgresql.html
Youtube-канал автора перевода MassonNn	https://www.youtube.com/@massonnn

Ссылки будут добавляться и корректироваться.

Автор: MassonNn (https://github.com/MassonNN)

Рецензент: Александр Антонов (https://github.com/AbstractiveNord/)