О. С. ЧЕРЕПАНОВ, А. В. МАЕР

ВВЕДЕНИЕ В PONYORM

УЧЕБНОЕ ПОСОБИЕ

ISBN 978-5-4217-0569-7

Курганский государственный университет

Библиотечно-издательский центр

65-48-12

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Курганский государственный университет»

О. С. Черепанов, А. В. Маер

ВВЕДЕНИЕ В PONYORM

Учебное пособие

Курган 2020

УДК 004.657
ББК 32.972.13
Ч–46

Рецензенты:

кандидат технических наук, доцент кафедры информатики и вычислительной техники ФГБОУ ВО «Омский государственный технический университет» А. С. Грицай;

директор департамента прикладных систем ГК ХОСТ (г. Екатеринбург), руководитель отдела разработки А. А. Завада.

Печатается по решению методического совета Курганского государственного университета.

Научный редактор – кандидат технических наук, доцент В. К. Волк.

Черепанов О. С., Маер А. В.
Введение в PonyORM : учебное пособие / О. С. Черепанов, А. В. Маер. – Курган : Изд-во Курганского гос. ун-та, 2020. – 85 с.

Учебное пособие посвящено введению в PonyORM – одному из популярных ORM для языка Python. Пособие состоит из двух глав. Первая глава посвящена описанию современных паттернов работы с СУБД. Во второй главе описывается основной функционал PonyORM.

Пособие предназначено для студентов IT-специальностей и может быть рекомендовано широкому кругу начинающих программистов, владеющих языком программирования Python, основами технологии баз данных и элементами языка SQL и желающих познакомиться с современными программными средствами работы с СУБД посредством технологии ORM.

ISBN 978-5-4217-0569-7

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5

ГЛАВА 1 ШАБЛОНЫ СЛОЯ ИСТОЧНИКА ДАННЫХ 9
1.1 Шлюз таблицы данных 9
1.2 Шлюз записи данных 11
1.3 Активная запись 12
1.4 Преобразователь данных 15
1.5 Паттерны отображения наследования классов 17

ГЛАВА 2 PONYORM 22
2.1 Особенности PonyORM 22
2.2 Установка PonyORM 22
2.3 Модель предметной области 23
2.4 Объявление классовых сущностей 26
2.4.1 Объявление сущностей 28
2.4.2 Атрибуты сущностей 28
2.4.3 Обязательность и опциональность 28
2.4.4 Set 29
2.4.5 Составные ключи и индексы 29
2.4.6 Типы данных атрибутов 30
2.4.7 Наследование 30
2.4.8 Настройка преобразования 33
2.4.9 Определение сущностей абонентского отдела библиотеки 34
2.5 Отношения между сущностями 37
2.5.1 Отношение «один-к-одному» 38
2.5.2 Отношение «один-ко-многим» 38
2.5.3 Отношение «многие-ко-многим» 39
2.5.4 Самоссылаемость 39
2.5.5 Несколько отношений между двумя сущностями 40
2.5.6 Определение отношений между сущностями в модели абонентского отдела библиотеки 41
2.6 Привязка к базе данных 43
2.7 Преобразование сущностей в таблицы базы данных 44
2.8 Режим отладки 44
2.9 Транзакции и db_session 44
2.9.1 Работа с db_session 45
2.9.2 db_session и объем транзакций 46
2.9.3 Несколько транзакций в одной сессии 47
2.9.4 Вложенные db_session 47

2.9.5 Кэш db_session 49
2.9.6 Работа с несколькими базами данных 49
2.9.7 Функции для работы с транзакциями 50
2.10 Работа с экземплярами сущностей 50
2.10.1 Создание экземпляров сущностей 50
2.10.2 Загрузка объектов из базы данных 51
2.10.3 Обновление объектов 54
2.10.4 Удаление объектов 55
2.10.5 Проблемы сохранения объектов в базе данных 57
2.11 Запросы 60
2.11.1 Использование генераторов выражений 60
2.11.2 Использование лямбда-функций 61
2.11.3 Примеры запросов 61
2.11.4 Использование даты и времени в запросах 63
2.11.5 Дубликаты 63
2.11.6 Использование сырых SQL-запросов 65
2.11.7 Использование select_by_sql() и get_by_sql() 68
2.12 Работа с отношениями между сущностями 69
2.12.1 Отношение «один-к-одному» 69
2.12.2 Отношение «один-ко-многим» 69
2.12.3 Отношение «многие-ко-многим» 70
2.12.4 Операции с коллекциями 71
2.12.5 Параметры атрибута коллекции 72
2.12.6 Запросы атрибутов коллекции и другие методы 73
2.13 Агрегатные функции 74
2.13.1 Использование агрегатных функций через объект запроса 75
2.13.2 Несколько агрегатных функций в одном запросе 75
2.13.3 Функция count 76
2.13.4 Условный count 76
2.13.5 Запросы с HAVING 78
2.13.6 ORDER_BY 79
2.14 Проектный практикум по PonyORM 80
2.14.1 Основные положения 80
2.14.2 Практические задания 80
ЗАКЛЮЧЕНИЕ 83
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 84

ВВЕДЕНИЕ

С момента зарождения отрасли разработки программного обеспечения (ПО) и до настоящего времени главным «врагом» программиста считается возрастающая сложность поставленных перед ним задач. У данной проблемы существуют две главные движущие силы: совершенствование вычислительной техники и неутолимое желание человечества решать все более сложные задачи. Эта проблема приводит к трудностям разработки, сопровождения, модификации программного обеспечения. Разработчики все время находятся в поисках «серебряной пули» [1] в виде новой парадигмы, языка, технологии программирования или методологии разработки, которые позволят успешно бороться с возрастающей сложностью.

Базовым принципом, который позволяет справиться со сложной задачей, в том числе и в разработке ПО, является принцип «разделяй и властвуй». В отрасли разработки программного обеспечения этот принцип реализовался на фундаментальном понятии «расслоение системы» [2]. Считается, что любое программное обеспечение архитектурно делится на три слоя: представление (Presentation), домен (Doman) и источник данных (Data source). Представление ответственно за отображение данных, предоставление услуг, обработку событий пользовательского интерфейса и т. д. Домен отвечает за бизнес-логику приложения, реализует основной функционал предметной области. Слой источника обеспечивает взаимодействие со сторонними системами, например, с системой управления базами данных (СУБД). Мы не будем касаться слоев представления и домена, а устремим свой взгляд на одну из сторон слоя источника данных – взаимодействие с СУБД.

В данном учебном пособии речь пойдет о современных способах взаимодействия с СУБД, поддерживающих только реляционный подход к хранению данных, так как на данный момент он пока еще преобладает на рынке ПО.

Взаимодействовать с СУБД можно различными способами. На достаточно низком уровне, например, если Вы используете язык C или C++, то можно воспользоваться функционалом библиотеки libpq для взаимодействия с СУБД PostgreSQL или libmysqlclient – для MySQL. Для других распространенных языков программирования существуют библиотеки практически ко всем наиболее часто встречаемым СУБД. Однако у данного подхода существуют как свои достоинства, так и недостатки.

Достоинства:

1) полное понимание процессов, происходящих в базе данных (БД). Полный контроль над схемой БД, хранимыми процедурами и триггерами;

2) использование расширений СУБД и собственных типов данных и индексов;

3) возможность оптимизации запросов.

Основной недостаток низкоуровневого подхода – привязка к конкретной СУБД. Если программист решил использовать другую СУБД, то это потребует переписывания значительной части кода слоя «Источник данных». Чтобы не попасть в такую ситуацию, можно воспользоваться структурным паттерном проектирования «Адаптер» [5], который позволяет адаптировать клиентский код к разным интерфейсам библиотек взаимодействия с СУБД, тем самым сделав шаг в сторону второго подхода.

Рассмотрим второй подход. На этом уровне ярким представителем является стандарт Java DataBase Connectivity (JDBC). Это платформенно-независимый стандарт взаимодействия Java-приложений с реляционными базами данных. Он решает следующие задачи:

1) соединение с СУБД;
2) создание SQL-запросов;
3) выполнение SQL-запросов.

Взаимодействие с СУБД осуществляется с использованием следующих элементов:

1) менеджер драйверов (Driver Manager) управляет списком драйверов БД;

2) драйвер (Driver) отвечает за связь с конкретной СУБД (Адаптер к СУБД);

3) соединение (Connection) – интерфейс, ответственный за соединение с БД;

4) выражение (Statement) используется для создания и выполнения SQL-запросов;

5) результат (ResultSet) отвечает за представление результатов SQL-запросов.

Рассмотренные задачи и структурные элементы JDBC показывают, что его принципиальные отличия от первого подхода на основе библиотек состоят в следующем:

1) позволяет безболезненно менять СУБД даже на этапе эксплуатации приложения за счет абстрагирования от конкретной реализации взаимодействия с ней. Это реализуется системой драйверов (паттерн «Адаптер»);

2) не требует знания специфики работы с конкретной СУБД. Это достигается благодаря введению абстрактного уровня взаимодействия с БД.

Хотя стандарт JDBC и написан с использованием объектно-ориентированной парадигмы проектирования, но он все еще не позволяет напрямую проектировать модель предметной области в виде объектной модели со встроенным механизмом объектно-реляционного отображения.

Известно, что весь путь программиста – это борьба за простоту. Естественное желание разработчика, который проектирует приложения с

использование ООП парадигмы, описывать предметную область в виде набора классов, объектов и связей между ними. Но если программист решил хранить данные в реляционной базе данных, он сталкивается с противоречием объектной и реляционной моделей. Такие противоречия проявляются, например, в виде описания различных отношений между сущностями: отношение «многие-ко-многим», отношение обобщения. Связи между сущностями в реляционных БД реализуются с использованием внешних и первичных ключей, а в объектной модели – с использованием указателей и ссылок. Требуются механизмы отображения объекта программного слоя в строку (строки) таблицы базы данных и наоборот. Такие задачи решаются на основе ORM.

Объектно-реляционное отображение (ORM) – технология программирования, которая решает задачу отображения сущностей объектной модели в сущности реляционной модели и наоборот. Современные ORM предоставляют прозрачные механизмы работы с базами данных так, как будто данные представлены на объектной модели. Этот подход полностью скрывает механизмы конвертации данных разных моделей и предоставляет удобный интерфейс работы с СУБД в виде набора классов и методов, которые встроены непосредственно в классы предметной области или в классы слоя источника данных.

У данного подхода есть свои достоинства и недостатки. Главными достоинствами ORM являются преодоление семантического разрыва парадигм и, что самое главное, повышение скорости разработки ПО за счет:

1) отсутствия необходимости писать SQL-запросы в типовых ситуациях. Типовые операции создания новой сущности, редактирования, удаления и поиска осуществляются естественно в рамках ООП парадигмы в виде вызова методов или создания объектов определенных классов. Но это не означает, что программисту запрещено писать SQL-код. Большинство современных ORM предоставляют возможность написания либо части SQL-запроса (например, блока WHERE), либо целого SQL-запроса;

2) некоторые ORM предоставляют в распоряжение программистам синтаксический сахар. Например, как мы далее увидим, PonyORM предоставляет преобразование генераторов выражений или лямбда-функций в части SQL-запросов. Это позволяет практически бесшовно работать с хранимыми данными.

Но у ORM есть свои недостатки. Это, в первую очередь, потеря производительности в двух направлениях.

1 Потеря производительности за счет трансляции в неоптимизированные запросы. Данный недостаток был особенно заметен в начале становления ORM. Сложные запросы, формируемые в ООП-стиле, транслировались в неоптимальные SQL-запросы. Но по мере развития технологии ORM трансляция становилась все лучше и лучше. Если при разработке программного обеспечения программист понимает, что его запрос

транслируется не оптимально, а в этом участке кода требуется высокая производительность, то этот запрос следует написать в SQL-стиле или в виде хранимой процедуры на уровне БД.

2 Высокий *overhead*. Как показывает практика, например, Hibernate [3] (ORM для языка Java) скрывает в себе работу с JDBC, а реализация драйверов последнего использует библиотеки для работы с конкретной СУБД. Отсюда получаем, что вся наша работа с Hibernate транслируется в работу с JDBC, а далее преобразуется во взаимодействие с библиотекой, например, libpq. На каждом уровне происходит преобразование одних вызовов в другие, одних форматов данных в другие, что приводит к падению производительности.

Средства объектно-реляционного отображения в настоящее время широко распространены при разработке программного обеспечения. Трудно найти современный язык, для которого не реализовано ORM. В таблице 1 представлены примеры ORM для некоторых распространенных языков программирования.

Таблица 1 – ORM для современных языков программирования

Язык	**ORM**
C++	odb, QxOrm
Java	EclipseLink, Hibernate
C#	NHibernate, Dapper, Entity Framework
Python	Peewee, SQLAlchemy, PonyORM
JavaScript/TypeScript	Sequelize, TypeORM

Данное учебное пособие представляет собой введение в PonyORM и по большей части является переводом официальной документации. За более подробной информацией предлагаем обратиться к официальному сайту проекта [6].

ГЛАВА 1
ШАБЛОНЫ СЛОЯ ИСТОЧНИКА ДАННЫХ

Паттернами или шаблонами проектирования называют решения типовых (часто встречаемых) задач в области проектирования программного обеспечения. Паттерны не предлагают точный, формализованный алгоритм решения таких задач, они только предлагают идею, принципы их решения, часто оформленные в виде диаграмм разного уровня.

Концепцию паттернов изначально ввел Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство» [4]. Данная идея была заимствована «бандой четырех» в лице Эриха Гамма, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса и оформлена в виде книги «банды четырех» [5], вышедшей в свет в 1995 году. В эту книгу вошли 23 шаблона, которые решали типовые проблемы проектирования архитектуры ПО.

В этом разделе будут рассмотрены шаблоны проектирования слоя «Источник данных». Этот слой обеспечивает интерфейс взаимодействия с компонентами инфраструктуры проектируемого приложения. Данное учебное пособие посвящено рассмотрению взаимодействия с реляционными базами данных, поэтому в качестве инфраструктуры будет выступать СУБД, поддерживающая реляционную модель хранения данных.

Традиционно выделяют четыре паттерна слоя «Источника данных» [2]:

1) шлюз таблицы данных (Table Date Gateway);
2) шлюз записи данных (Row Date Gateway);
3) активная запись (Active Record);
4) преобразователь данных (Data Mapper).

Все эти типовые решения предназначены в первую очередь для сокрытия от пользовательского кода способов взаимодействия с СУБД. Клиенту только предоставляется общий интерфейс взаимодействия с СУБД. Если Вы используете реляционную базу данных, то рассмотренные типовые решения за счет инкапсуляции логики взаимодействия скроют от клиентского кода SQL-запросы и не дадут «размазать» его по всей кодовой базе приложения.

1.1 Шлюз таблицы данных

Принцип действия паттерна чрезвычайно прост. Для каждой таблицы базы данных, а точнее сущности, создается класс-шлюз (рисунок 1.1), который включает несколько методов поиска объектов по разным критериям, методы создания, обновления и удаления. Каждый метод шлюза в итоге создает SQL-запрос, помещая в него значения переданных ему аргументов, а далее создает соединение с СУБД и отправляет ей запрос на выполнение. Как правило, шлюз таблицы данных не обладает своим состоянием, так как передает данные в таблицы и получает из них. Важная особенность

паттерна заключается в том, что он должен возвращать. При поиске по первичному ключу шлюз может возвращать запись (*Record*), которая может быть представлена в виде словаря, ключом которого является название столбца, а значением выступает значение ячейки таблицы. При поиске человека, например, по фамилии, шлюз должен возвращать множество записей типа *Record* (*RecordSet*). При успешном обновлении и удалении записей шлюз может ничего не возвращать. Нередко шлюз при создании новой записи может возвращать значение первичного ключа, если он назначается автоматически средствами СУБД.

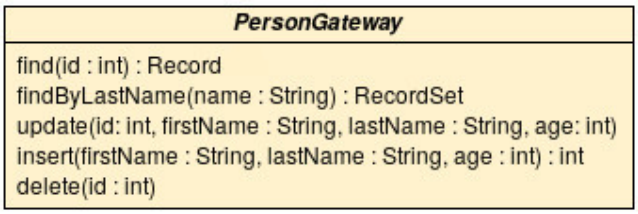

Рисунок 1.1 – Шлюз таблицы persons

Приведем краткое описание примера шлюза на рисунке 1.1. *PersonGateway* – шлюз к таблице *persons*, которая включает столбцы: *id(первичный ключ)*, *first_name(имя)*, *second_name(фамилия)*, *age(возраст)*. Возвращаемое значение типа *Record* содержит соответствующие колонкам таблицы ключи: *id*, *first_name*, *second_name*, *age*.

1 *find(id: int): Record* – метод поиска человека по первичному ключу. Если человек найден, то шлюз возвратит соответствующий ему объект типа *Record*.

2 *findByLastName(name: String): RecordSet* – метод поиска людей по фамилии (*name*). Возвращает множество записей типа (*RecordSet*).

3 *update(id: int, first_name: String, second_name: String, age: int)* – метод обновления человека в БД по его первичному ключу.

4 *insert(first_name: String, second_name: String, age: int)* – метод создания человека в БД. Возвращает присвоенное ему значение первичного ключа.

5 *delete(id: int)* – метод удаления человека по его *id*.

Приведенный на рисунке 1.1 шлюз используют в том случае, когда слой предметной области реализован с использованием паттерна «Сценарий транзакций» [2]. В этом случае вся бизнес-логика – это набор процедур/функций. Данный паттерн рекомендуют использовать, когда предметная область простая, количество вызовов таких процедур невелико. В противном случае код превратится в «спагетти-код». Поддерживать и развивать такой код становится чересчур затруднительно. Одно из решений данной проблемы – переход к объектной модели предметной области.

Объектная модель предметной области представляет собой набор классов, связей между ними и порождаемыми ими объектами (объектно-

ориентированная парадигма проектирования). Если проектировщик выбрал данный подход, использовать шлюз таблицы данных в представленном выше случае затруднительно. Вместо этого в методы шлюза удобней передавать объекты соответствующих шлюзу классов. В нашем случае в методы *insert()*, *update()* и *delete()* могут быть переданы объекты класса *Человек* (*Person*). А методы поиска людей могли бы возвращать объект класса *Person* и коллекцию объектов класса *Person*.

1.2 Шлюз записи данных

В предыдущем паттерне создаются классы-шлюзы обычно для каждой сущности и представляют собой интерфейсы к таблицам базы данных. Шлюз записи данных – это, как правило, интерфейс к строке базы данных. С другой стороны, объект такого класса – это отображение строки базы данных в экземпляре класса предметной области. Поэтому такой класс инкапсулирует как свойства, которые соответствуют колонкам таблицы базы данных, так и методы работы со строкой базы данных, такие как создание, обновление и удаление. Шлюзы к записи данных должны знать, как конвертировать типы объектной модели в типы, поддерживаемые выбранной СУБД.

Но не стоит думать, что если класс содержит свойства, характерные для предметной области, то он находится в слое домена. Нет, как и полагается, шлюзы записи находятся в слое источника данных и содержат данные поля исключительно с целью отображения их в реляционную модель. Логика предметной области в виде, например, набора методов в таких классах отсутствует. Шлюз записи данных включает только логику работы с СУБД.

Когда приходит время реализовывать шлюзы записи данных, возникает дилемма определения места методов поиска. Конечно, можно разместить эти методы непосредственно в шлюз, сделав их статическими. Однако в таком случае программист столкнется с проблемой полиморфизма статических методов. Поэтому лучшим решением будет поместить методы поиска в отдельный класс, который занимается исключительно поиском.

Рассмотрим вариант использования данного паттерна. На рисунке 1.2 представлены два типовых класса. *PersonGateway* – шлюз к записи в таблице *person*. Объект этого класса – это объектное представление строки в таблице *person*. Данный класс содержит свойства *id(первичный ключ)*, *first_name (имя)*, *second_name(фамилия)*, *age(возраст)*, которые соответствуют колонкам в таблице *person*. *PersonFinder* – класс, ответственный за поиск людей по первичному ключу (метод *find ()*) или по фамилии (*метод findByLastName ()*). На рисунке 1.3 представлена диаграмма последовательности обновления информации о человеке. Первоначально необходимо создать объект *PersonFinder*, далее

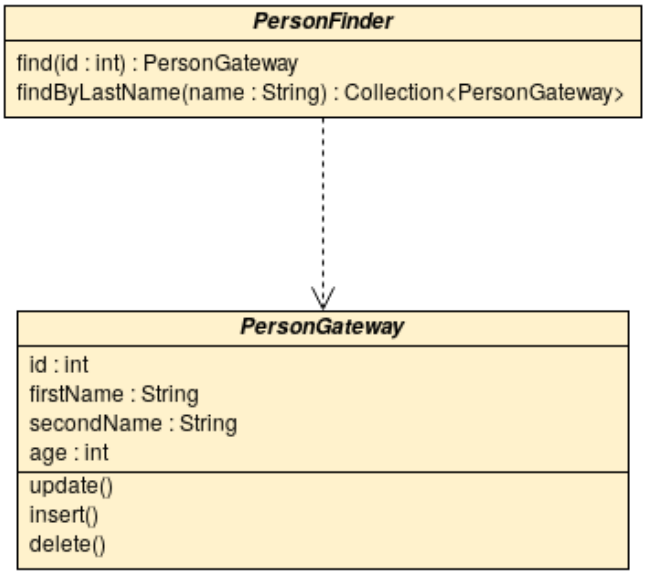

Рисунок 1.2 – Шлюз записи данных таблицы persons

вызываем у него метод *find ()* и, например, передаем ему значение 64. В коде выполнения данного метода будет создан запрос вида:

```
SELECT "persons"."id", "persons"."fist_name",
       "persons"."last_name", "persons"."age"
FROM "persons"
WHERE "persons"."id" = 64 LIMIT 1;
```

После разбора результата исполненного запроса будет создан объект *pg* типа *Person-Gateway*, который будет передан клиенту. Пусть клиентский код захотел изменить фамилию данного человека, вызвав соответствующий *set*-метод объекта *pg*, передав ему значение «Иванов». После изменения фамилии необходимо закрепить эти изменения в базе данных. Для этого необходимо вызвать метод *update ()* объекта *pg*. В ходе выполнения этого метода будет сформирован и исполнен примерно такой SQL-код:

```
UPDATE "persons" SET "persons"."last_name" = 'Иванов'
WHERE "persons"."id" = 64;
```

1.3 Активная запись

В паттерне «Шлюз записи данных», если мы принимаем объектную модель предметной области, то для каждой сущности, как правило, приходится описывать класс предметной области и класс-шлюз. Мы будем вынуждены дублировать атрибуты сущности, так как они схожи как в классе домена, так и в шлюзе записи. Однако это – плата за расслоение системы. Поэтому данный паттерн редко используется совместно с моделью

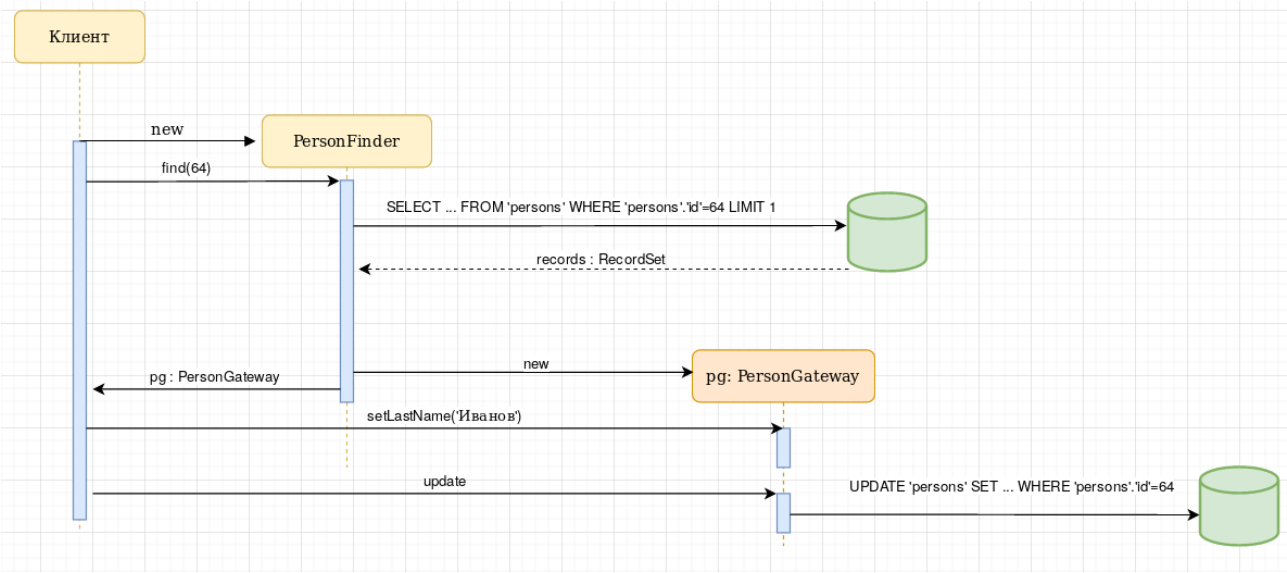

Рисунок 1.3 – Диаграмма последовательности обновления фамилии человека

предметной области. Наиболее частое его применение – связка с паттерном «Сценарий транзакций».

Но что делать, если у нас сложная модель предметной области и мы вынуждены использовать ее объектную модель? В этом случае может помочь активная запись. Чтобы решить вышеописанную проблему дублирования атрибутов сущности, нужно два этих класса объединить в один. Он будет включать и поведение объектов бизнес-логики, и методы взаимодействия с СУБД.

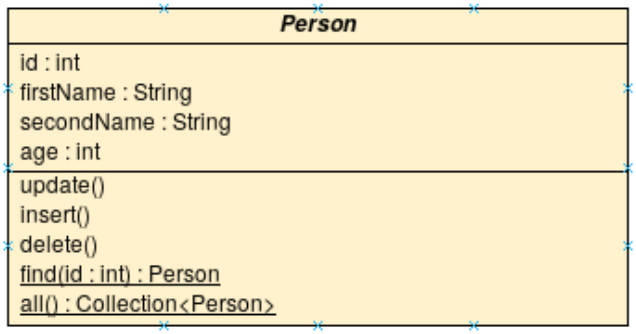

Рисунок 1.4 – Диаграмма класса Person с использованием активной записи

На рисунке 1.4 представлена диаграмма классов паттерна «Активная запись». На первый взгляд, активную запись и шлюз записи данных можно спутать. Но здесь существует правило: «Если класс включает в себя методы предметной области и методы работы с СУБД, то мы имеем дело с активной записью». В активную запись часто включают методы поиска непосредственно в класс, делая их статическими или методами класса (если в языке существуют одновременно два таких понятия). Одним из главным свойств активной записи является изоморфность отображения, т. е. один

класс соответствует одной таблице базы данных, каждое свойство находит себе соответствие в виде колонки данной таблицы, а между ними существует простое отображение типов.

Активная запись – чрезвычайно популярный паттерн, порог понимания его работы невысокий. Поэтому он лежит в основе многих фреймворков и библиотек. Однако у него существует один фундаментальный недостаток – смешение двух слоев: бизнес-логики и источника данных. В этом случае мы не можем абстрагироваться от способа хранения данных. И если Вы в будущем захотите изменить способ хранения данных, Вам необходимо будет провести рефакторинг домена, а это идет вразрез с фундаментальными принципами расслоения системы и принципами SOLID.

Рассмотрим вариант использования активной записи – изменение фамилии человека с идентификатором 64. Алгоритм следующий:

1) вызываем статический метод *find ()* класса *Person*, передавая ему идентификатор человека и получая объект типа *Person*;

2) изменяем фамилию человека путем вызова у полученного объекта метода *setLastName ()*, передав ему новую фамилию;

3) для закрепления изменений в БД вызываем метод *update ()*.

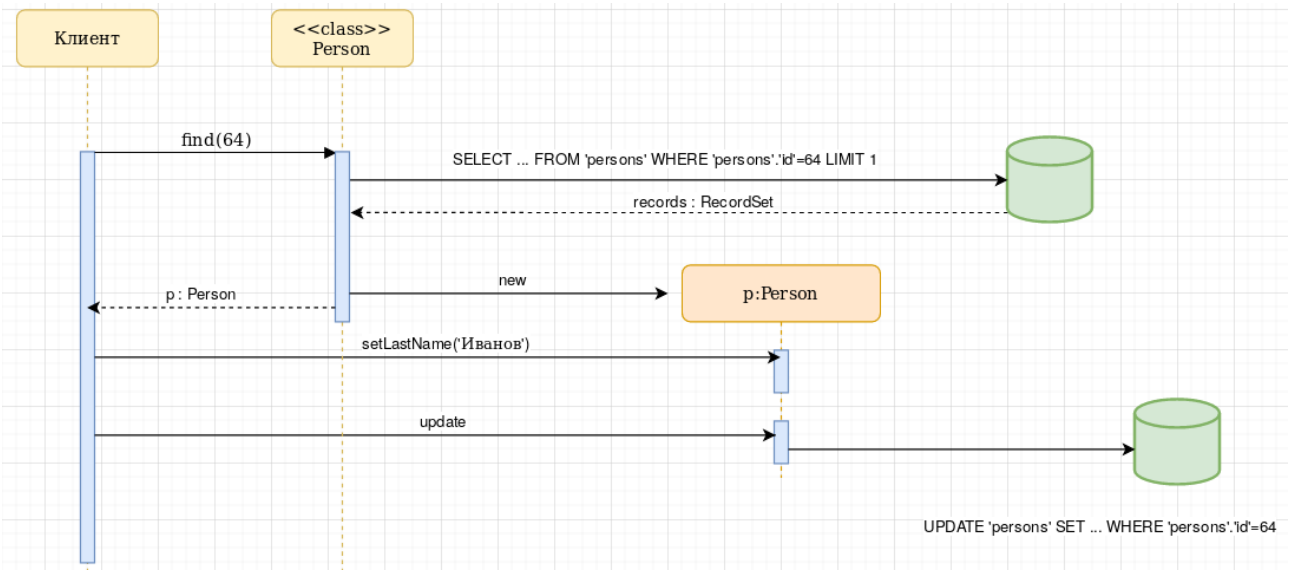

Рисунок 1.5 – Диаграмма последовательности обновления фамилии человека

Приведенный алгоритм и его представление в виде диаграммы последовательности (рисунок 1.5) показывают, что работа с активной записью несколько проще, чем со шлюзом записи данных. Но не стоит считать, что в методах классов активной записи или шлюза записи непосредственно производится формирование SQL-кода. Эта функциональность наследуется от родительских классов, а также часто используются механизмы шаблонов.

1.4 Преобразователь данных

Как было сказано выше, активная запись напрямую или косвенно влияет на структуру базы данных. В этом случае нельзя отдельно проводить рефакторинг классов домена и схемы базы данных. При этом теряется гибкость системы. Иногда случается, что схема базы данных проектируется независимо от объектной модели, достигая высоких форм нормализации. В результате чего схема базы данных четко не соответствует объектной модели. Например, одной таблице базы данных может соответствовать несколько объектов предметной области или наоборот. Возникает необходимость привнести независимость схемы базы данных от объектной модели предметной области. Это можно достичь путем использования паттерна «Преобразователь данных».

Преобразователь данных представляет собой программный подслой, основная задача которого заключается в преобразовании сущностей базы данных в объекты программного слоя и наоборот. В итоге объекты, находящиеся в оперативной памяти, не подозревают о том, как они представлены в БД. Как правило, для каждой сущности предметной области описывается преобразователь. Часто каждый преобразователь хранит у себя коллекцию объектов с целью их кэширования. При этом он должен знать, какие объекты были загружены, какие изменены, а какие должны быть удалены из базы данных.

Обычно объекты тесно связаны между собой, поэтому может случиться так, что при загрузке объекта из базы данных он начнет тянуть за собой связанные с ним другие объекты. Это может привести к тому, что в оперативную память будет загружена вся база данных. Для решения этой проблемы часто применяют паттерны «Ленивая загрузка» [2]. Суть ее заключается в том, что загружаются связанные объекты только по требованию, например, при вызове *get*-метода. Но у загрузки по требованию имеются и недостатки. Каждая подгрузка связанных объектов приводит к формированию и исполнению запроса к БД. Частое применение ленивой загрузки приведет к множеству небольших SQL-запросов, исполнение которых негативно скажется на производительности всей системы. Если при анализе варианта использования становится понятно, что при загрузке объекта всегда требуется загрузка связанного с ним другого объекта, то ленивую загрузку здесь не стоит применять.

Из диаграммы классов преобразователя (рисунок 1.6) видно, что на вход преобразователя подаются объекты определенного класса. Это означает, что преобразователь должен знать структуру данного класса для выполнения своих задач. Доступ к свойствам класса может осуществляться разными способами. Первый – с использованием *set/get*-методов. Второй – организация непосредственного доступа к структуре объектов домена через

механизмы дружественности или рефлексии. Однако в том и другом случаях возникают проблемы.

1 Какие поля нужно отражать на какие столбцы таблицы базы данных?

2 Что произойдет, если в приложении используется иерархия классов?

3 Какие изменения произойдут в слое преобразователя данных?

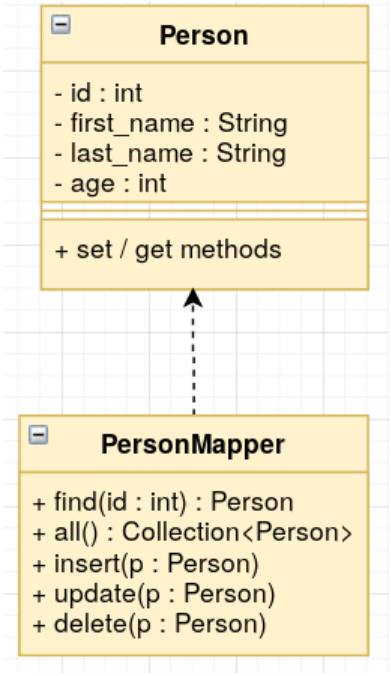

Рисунок 1.6 – Диаграмма класса паттерна «Преобразователь данных»

Существуют два способа сообщить преобразователю, какие поля необходимо отображать. Первый – явно поместить эту логику в код преобразователя. Второй – использовать мета-программирование, например, применить технологию аннотирования, как и предлагается в JPA. Решение проблемы отображения иерархии классов определяется способом ее представления в схеме базы данных. Но чаще это приводит к созданию иерархии преобразователей, каждый из которых ответственен за отображение одного из классов иерархии в таблицу базы данных.

На рисунке 1.7 представлена диаграмма последовательности обновления фамилии человека с идентификатором 64. Первоначально необходимо найти человека. Для этого клиентский код создает объект преобразователя (*PersonMapper*) и вызывает метод *find()*, передавая значение идентификатора. В ответ клиентскому коду возвращается объект типа *Person*. Для изменения его фамилии воспользуемся методом *setLastName ()*, отправляя ему новое значение фамилии. Для закрепления результатов необходимо вызвать метод *update ()* преобразователя, передав ему измененный объект человека.

Типовое решение «Преобразователь данных» является наиболее гибким решением работы с хранилищем данных, позволяет отделить слой

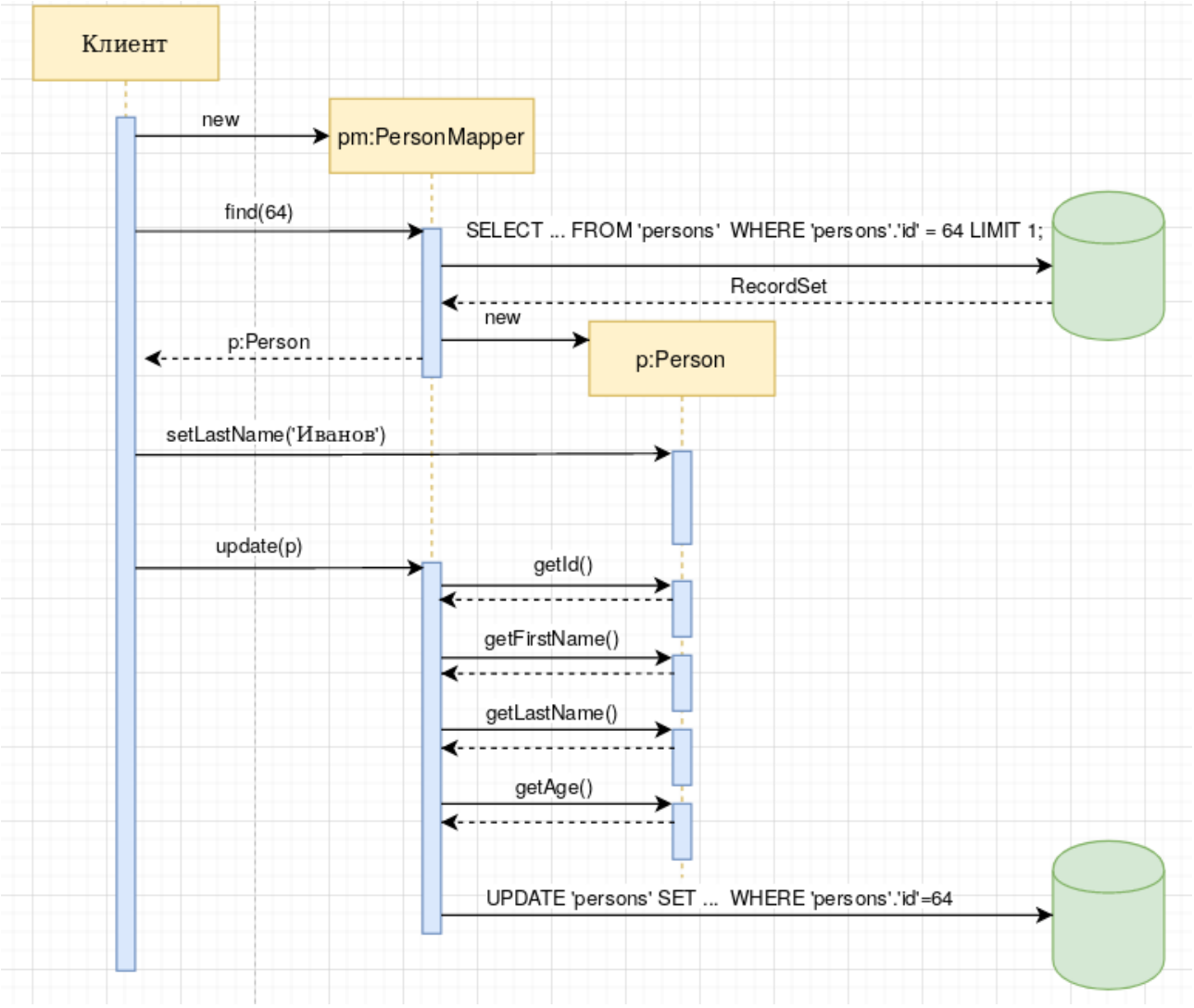

Рисунок 1.7 – Диаграмма последовательности обновления фамилии человека с использованием паттерна источника данных «Преобразователь данных»

домена от источника данных, тем самым соблюдая фундаментальное правило расслоения системы.

1.5 Паттерны отображения наследования классов

Существует несколько реализаций наследования в базе данных [2].

1 Наследование с одной таблицей.

2 Наследование с таблицами классов.

3 Наследование с таблицами для каждого конкретного класса.

Пусть имеется иерархия классов, диаграмма которых представлена на рисунке 1.8.

Наследование с одной таблицей (Single table inheritance – STI). Рассмотрим, во что отобразится приведенная иерархия классов

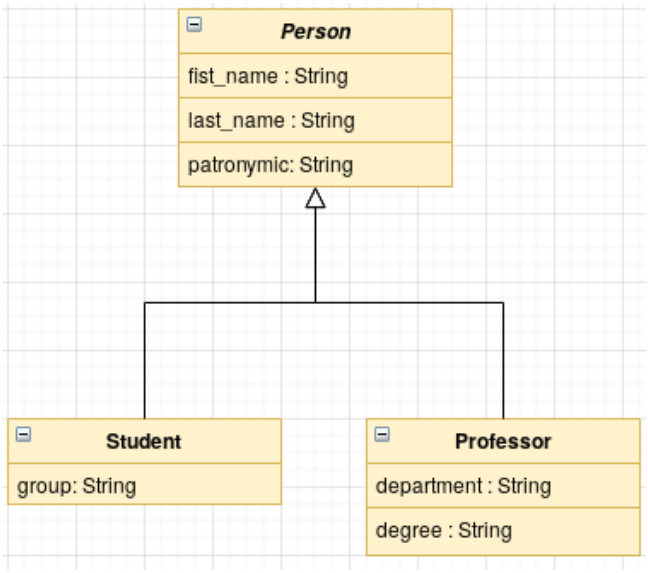

Рисунок 1.8 – Диаграмма иерархии классов

согласно данному паттерну. На рисунке 1.9 представлена схема базы данных. Видим, что для всей иерархии создается одна таблица базы данных, в которую помещаются все атрибуты всех классов иерархии.

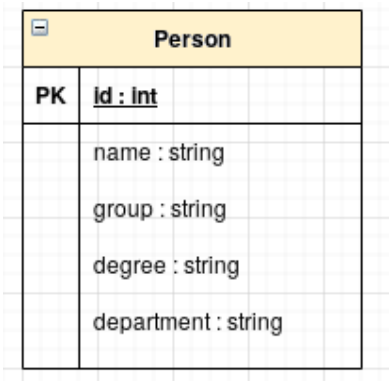

Рисунок 1.9 – Схема базы данных для паттерна STI

Данная стратегия отображения является лучшей с точки зрения производительности и простоты. Запросы выборки работают быстро, не нужно применять сложные операции соединения таблиц или объединения результатов. Но есть одна главная проблема – это проблема целостности данных. Столбцы для атрибутов, объявленных в дочерних классах, могут содержать значения *null*. Если каждый дочерний класс объявляет несколько атрибутов, которым нельзя присваивать значение *null*, то отказ от ограничения NOT NULL может стать непреодолимой проблемой. Представьте, что приложение требует наличия ФИО у человека, но схема базы данных не способна это требование обеспечить, так как каждый столбец в таблице может иметь значение NULL.

Другим слабым местом этого подхода является проблема нормализации схемы базы данных. Создается функциональная зависимость между

неключевыми столбцами, при этом нарушается третья нормальная форма. Денормализация схемы для повышения производительности может привести к снижению долговременной стабильности, удобства сопровождения и отсутствию гарантий целостности данных.

Однако данный подход часто распространен в современных фреймворках благодаря своей простоте использования. Стоит отметить, что в таблице хранятся строки, которые должны быть отображены в объекты разных классов, участвующих в иерархии. Требуется знать, какие строки соответствуют объектам каких классов. Для этого в таблицу добавляется еще одна колонка – дескриминатор – метка, по которой можно однозначно определить класс данной строки. Часто этот столбец имеет строковый тип, а значение в ней соответствует названию класса.

Наследование с таблицами для каждого класса (Class Table Inheritance – CTI).

Данная стратегия для каждого класса/подкласса, включая абстрактные классы, объявляет отдельную таблицу. Каждая такая таблица содержит только столбцы для ненаследуемых свойств, объявленных в самом подклассе наряду с первичным ключом, также являющимся внешним ключом таблицы суперкласса. Схема базы данных для иерархии классов представлена на рисунке 1.10.

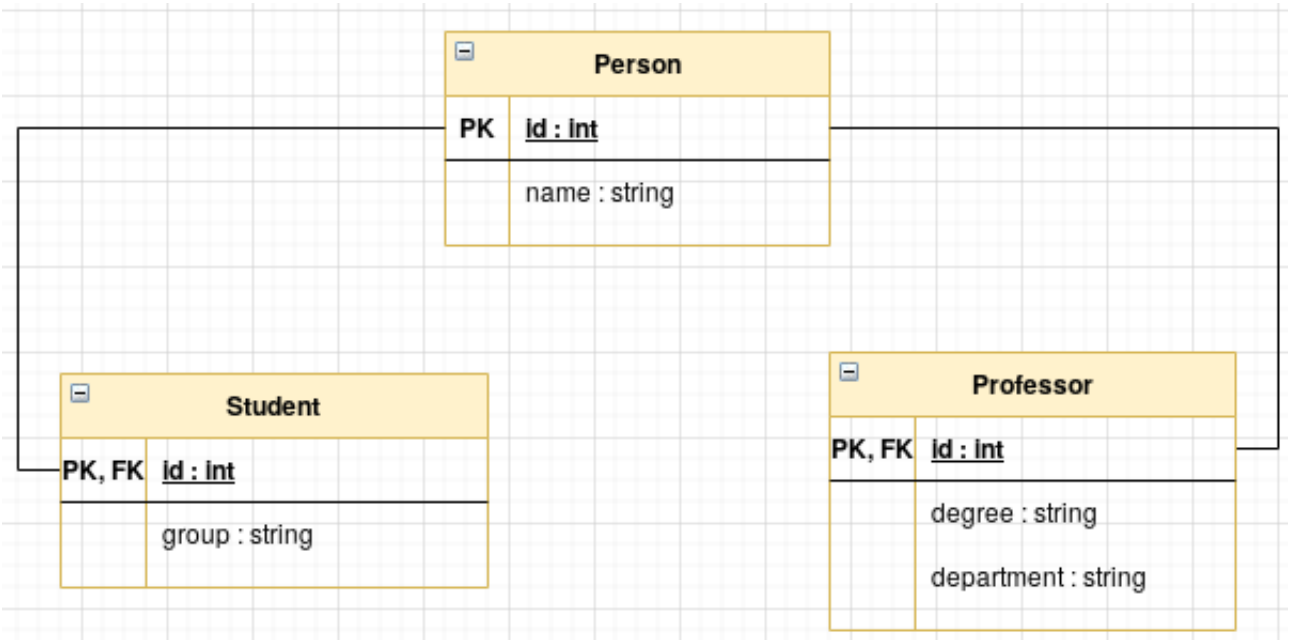

Рисунок 1.10 – Схема базы данных для паттерна CTI

При сохранении экземпляра подкласса *Student* в базу данных вставляются две записи. Значения свойства суперкласса *Person* сохраняются в таблице *persons*. Значения свойств, объявленных лишь в подклассе, сохраняются в таблице *students*. Эти две записи объединяются по средствам первичного ключа. Экземпляр подкласса может быть получен из базы

данных путем соединения таблиц *students* и *persons*, используя первичный ключ.

Главное достоинство этого паттерна – нормализация схемы SQL. Рефакторинг с соблюдением ограничений целостности осуществляется довольно легко. Внешний ключ, который ссылается на запись в таблицу конкретного дочернего класса, может представлять полиморфную ассоциацию с этим конкретным подклассом. Схема базы данных полностью соответствует объектной модели предметной области.

Однако у данного подхода представления наследования в реляционной базе данных существует и недостаток – низкая производительность. Извлечение значений всех свойств объекта листового уровня иерархии классов требует соединения таблиц. Чем глубже иерархия классов, тем больше таких соединений. Каждое такое соединение таблиц приводит к «проседанию» производительности приложения. А таблица корня иерархии может стать «узким горлышком» из-за частого к ней обращения.

Наследование с таблицами для каждого конкретного класса. (Concrete Table Inheritance (CoTI)).

В рамках предыдущего подхода таблица создавалась для всех классов, даже если он был абстрактным. В данном же случае, как говорит название паттерна, таблицы создаются только для конкретных классов, для абстрактных классов таблицы не создаются. При этом каждая таблица содержит столбцы, соответствующие полям конкретного класса и всех его предков, а потому поля родительских классов дублируются во всех таблицах его производных классов. На рисунке 1.11 представлена схема базы данных для рассмотренной выше иерархии классов согласно данному паттерну.

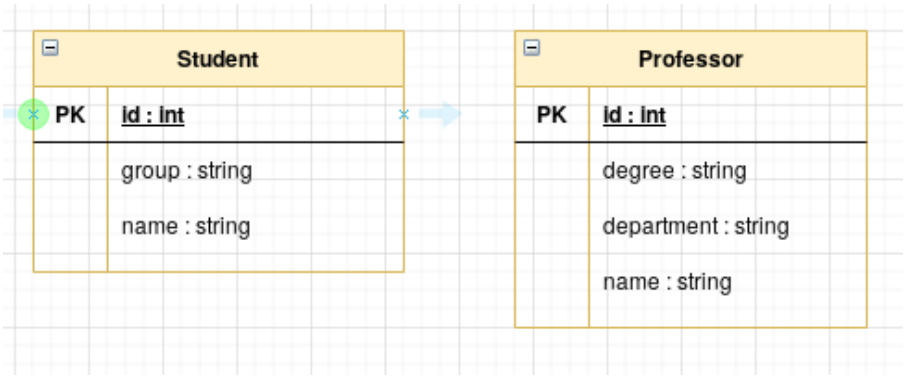

Рисунок 1.11 – Схема базы данных для паттерна CoTI

Несложно заметить, данный подход является промежуточным между STI и CTI. Перечислим основные преимущества наследования с таблицами для каждого конкретного класса.

1 Каждая таблица является замкнутой и не содержит ненужных полей, поэтому ее удобно использовать в других приложениях, не работающих с объектами.

2 Не нужно выполнять соединения таблиц для того, чтобы получить полностью всю информацию об объекте конкретного класса.

3 Практически равномерное распределение нагрузки по всей базе данных.

У данного подхода есть недостатки. Главный – отсутствие достаточной поддержки полиморфных ассоциаций. Данная проблема может проявиться, если приложению потребуется получить всех людей. В данном подходе отсутствует такая таблица с информацией о свойствах класса *Person*. В этом случае необходимо выполнить запросы к каждой таблице, а затем объединять их результаты. Другой проблемой является разделение одинаковой семантики по разным таблицам. Это усложняет рефакторинг схемы базы данных. Например, если изменился тип атрибута родительского класса верхнего уровня, это приведет к рекурсивным изменениям у всех таблиц данной ветки.

Рекомендации.

Рекомендации по выбору паттерна реализации наследования в реляционной базе данных выразим в виде следующих правил.

1 Если у Вас иерархия классов неширокая и неглубокая и классы отличаются по свойствам незначительно, то стоит присмотреться к паттерну STI.

2 Если требуется высокая производительность выборки данных, необходимо использовать STI.

3 Если имеется разветвленная иерархия классов, но не требуется наличие высокой производительности, лучше использовать CTI.

4 Если в приложении существует вариант использования, в котором фигурируют полиморфные ассоциации, то использовать CoTI не рекомендуется.

5 Если Вам требуется баланс между производительностью и высокой формой нормализации схемы БД, то стоит использовать CoTI.

ГЛАВА 2
PONYORM

2.1 Особенности PonyORM

PonyORM – программное средство расширенного объектно-реляционного отображения данных для языка Python. Существуют и другие популярные ORM для данного языка, например, Django и SQLAlchemy, но у PonyORM есть ряд явных преимуществ.

1 Исключительно удобный синтаксис для написания запросов.

2 Оптимизация запросов.

3 Онлайн редактор схемы базы данных.

По сравнению с Django, PonyORM обладает:

1) автоматическим управлением транзакциями;
2) автоматическим кэшированием запросов и объектов;
3) поддержкой составных ключей;
4) способностью легко писать запросы и использованием LEFT JOIN, HAVING и других особенностей SQL.

Одна из особенностей PonyORM – он позволяет работать с БД в Python-стиле, используя генераторы выражений, лямбда-функции, которые транслируются в SQL. Такие запросы легко могут быть написаны разработчиками, использующими Python, но не являющимися экспертами в SQL. Приведем ряд примеров:

```
select (p for p in Person if len(c.languages) > 3)
Person.select(lambda p: len(c.languages) > 3)
```

В этом примере мы получаем всех людей, говорящих более чем на трех языках. Запрос формирует *select*-функция, а аргумент — генератор списка или лямбда-функция, привычные приемы в языке Python. На самом деле PonyORM не создает генератор, а транслирует его в SQL-запрос и отправляет его в СУБД. Такой подход позволяет программистам писать сложные запросы, не будучи экспертами в SQL-языке.

Не каждый ORM имеет такой синтаксис, как PonyORM. Важно отметить, что он эффективно работает с данными. Трансляция в SQL-запросы осуществляется эффективно и быстро. Главная цель PonyORM, как и других ORM, – облегчить процесс разработки приложений, которые хранят данные в реляционных базах данных.

2.2 Установка PonyORM

Для установки PonyORM используйте следующую команду:

```
pip install pony
```

PonyORM может быть установлена для Python 2.7 и Python 3. Если Вы собираетесь работать с SQLite базой данных, то устанавливать кроме *pony* ничего не нужно. Если Вы собираетесь работать с другими СУБД, то Вам необходимо будет установить следующие пакеты (драйвера):

- Для PostegreSQL — psycopg2 или psycopgcffi;
- Для MySQL — MySQL-python or PyMySQL;
- Для Oracle — cx_Oracle.

Чтобы удостовериться, что PonyORM был удачно установлен, можете написать следующее:

```
from pony.orm import *
```

Произойдет импорт всего множества классов ORM и функций, необходимых для работы с PonyORM.

Если Вы не хотите импортировать все в глобальное пространство имен, Вы можете импортировать только *orm* пакет:

```
from pony.orm import orm
```

2.3 Модель предметной области

Рассмотрим учебный пример – автоматизация абонентского отдела библиотеки. Данный пример не претендует на полноту, на реализацию всех нюансов системы хранения и выдачи печатных изданий. Цель заключается только в изучении фреймворка PonyORM.

Опишем предметную область. Пусть имеется библиотека, которая хранит и может выдавать книги читателям. Каждая книга имеет название, написана автором или коллективом авторов, относится к одному или нескольким жанрам, содержит год издания и количество экземпляров. Автор, сотрудник и читатель библиотеки имеют имя, фамилию и отчество (не обязательно). Отличительная особенность сотрудника – он имеет право выдавать и принимать книги от читателя, записывая эту информацию в журнал выдачи/возврата, а также он работает в какой-то должности. Читатель может брать и возвращать книги.

При проектировании модели предметной области принято выделять сущности и объект-значения. Под **сущностью** понимают логически целостный объект, определяемый совокупностью индивидуальных черт [7]. Сущность – это все то, что сохраняет свое индивидуальное существование и отличие на протяжении срока «жизни» независимо от атрибутов, важных для пользователя приложения. Сущностью может быть человек, город, автомобиль и т. д.

Объектом-значением называется объект, который предоставляет описательный аспект предметной области и не имеет индивидуального существования, собственной идентичности [7]. Объектами-значениями часто бывают цвет, строки, числа. Считается, что объекты-значения являются неизменяемыми объектами. Часто сущности ссылаются на объекты-значение. В этом случае разные сущности могут ссылаться на один и тот же объект-значение, ведь он никогда не будет изменен. Важно, что объект-значение лишен индивидуальности.

Давайте из описания предметной области выделим основные классы домена:

- Читатель;
- Сотрудник;
- Автор;
- Книга;
- Жанр;
- Должность сотрудника;
- Журнал выдачи/возврата;
- Действие (возврат книги или ее выдача).

Из данных классов постараемся выделить сущности и объекты-значения. Важным признаком сущностей является наличие индивидуальности, поэтому к сущностям отнесем:

- Читателя;
- Сотрудника;
- Автора;
- Книгу;
- Журнал выдачи/возврата.

Тогда к объектам-значениям отнесем:

- Жанр;
- Должность сотрудника;
- Действие (возврат книги или ее выдача).

К сожалению, в PonyORM нет различия между сущностями и объектами-значениями. Все есть сущность. Однако мы считаем, что деление на сущности и объекты-значения является важным вне зависимости от того, поддерживается это программными средствами или нет.

Вторым этапом проектирования модели предметной области является определение отношений между классами (в нашем случае сущностями). В объектно-ориентированной парадигме проектирования существует 5 видов отношений.

1 **Обобщение** – отношение специализации или наследования. Явным признаком этой связи является утвердительный ответ на вопрос «Является?». Например, медведь является млекопитающим? Да, значит между ними есть отношение обобщения. Говорят, что объекты дочернего класса (в нашем примере это медведи) могут наследовать структуру и поведение

родительского класса (млекопитающее). В нашей предметной области есть сущности *Читатель*, *Сотрудник* и *Автор*, которые обладают едиными атрибутами ФИО. Чтобы не нарушать принцип DRY (Don't repeat yourself – не повторяйся), мы можем вынести эти общие атрибуты в родительский класс *Человек*. Это будет родительский класс, а *Читатель*, *Сотрудники* и *Автор* будут дочерними классами.

2 **Ассоциация** – именованное отношение между двумя или более классами, которое специфицирует характер этой связи между объектами этих классов. Данный вид связи обладает свойствами: название, арность и кратность. Арность определяет количество классов, задействованных в отношении, а кратность – свойство, которое обозначает, какое количество объектов одного класса имеют место в случае, когда остальные объекты отношения фиксированы. Для нашей предметной области такой вид отношений достаточно распространен:

- ассоциация между книгой и автором. Название ассоциации – «написал(и)», арность – 2, кратность – «n – m»;
- ассоциация между книгой и жанром. Название ассоциации – «имеет», арность – 2, кратность – «n – m»;
- ассоциация между сотрудником и должностью. Название – «работает в», арность – 2, кратность – «n – 1» и т. д.

3 **Агрегация и композиция** – частный случай ассоциации, определяющий отношение «целое – часть». Агрегация и композиция встречаются, когда одни объекты являются коллекциями или контейнерами других объектов. Отношение между целым и его частями в агрегации более слабое, чем в композиции. Это означает, что если объект-целое будет уничтожен, то части останутся существовать. Композиция же является более строгой агрегацией. В этом случае части не могут существовать без целого. Примером агрегации может служить системный блок компьютера, который состоит из процессора, материнской платы, видеокарты и т. д. В этом случае компоненты могут существовать в отдельности от системного блока и могут быть перенесены в другой системный блок. Пример композиции – университет, который состоит из институтов. В этом случае институты не могут существовать в отрыве от университета, и вряд ли разные университеты могут обмениваться институтами. В нашей предметной области сущности «Жанр» и «Книга» связаны отношением «Агрегация».

4 **Зависимость** – слабая форма отношения между объектами, при которой изменение в спецификации одного влечет за собой изменения в другом. Часто данный вид отношения реализуется в нескольких случаях:

- один класс имеет метод, в который передается объект другого класса в качестве одного из аргументов;
- один объект использует другие глобальные объекты.

Главная особенность зависимости – один класс не хранит ссылки внутри себя на объекты другого класса. В нашей модели предметной области такого вида

отношения нет.

5 Реализация – отношение между двумя элементами модели (чаще всего между интерфейсом и классом), в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком). В нашем слое домена нет интерфейсов, поэтому и отношения реализации также отсутствуют.

Модель предметной области представлена на диаграмме (рисунок 2.1).

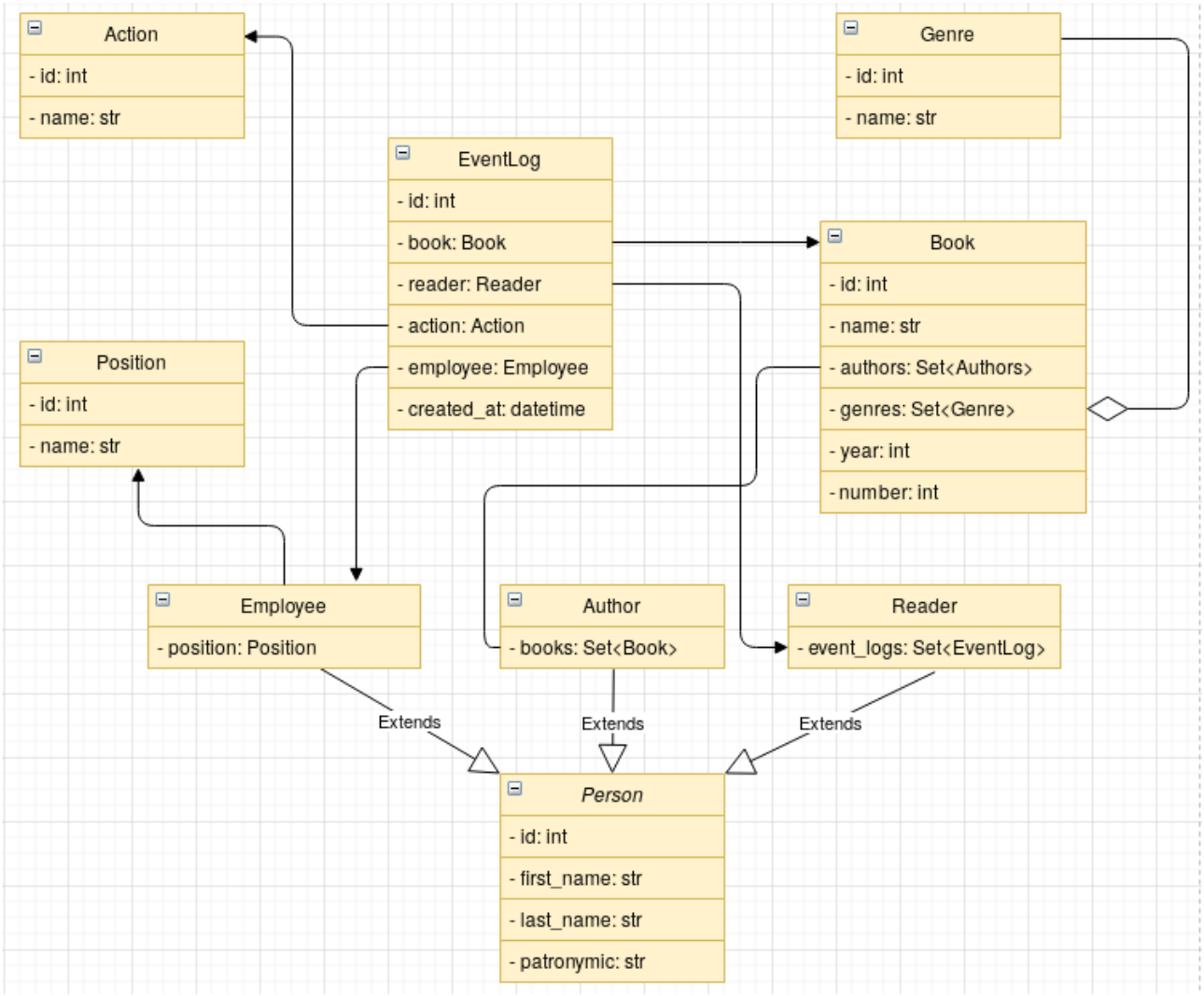

Рисунок 2.1 – Диаграмма классов модели предметной области

В таблице 2.1 приведено описание представленных классов.

2.4 Объявление классовых сущностей

В PonyORM классовые сущности – это классы языка Python, объекты которых хранят свое состояние в базе данных. Каждый экземпляр классовой сущности соответствует строке определенной таблицы базы данных.

Перед созданием экземпляра сущности необходимо отобразить классовые сущности в таблицы базы данных. PonyORM может отобразить

Таблица 2.1 – Описание классов предметной области

Person (Человек) – абстрактный класс	
id	Первичный ключ
first_name	Имя
last_name	Фамилия
patronymic	Отчество (необязательное свойство)
Employee (Сотрудник)	
position	Должность
Author (Автор книг)	
books	Книги
Reader (Читатель)	
event_logs	Записи в его читательском билете. Содержат информацию о том, какие книги, когда взял.
Book (Книга)	
id	Первичный ключ
authors	Авторы
genres	Жанры
year	Год издания
number	Количество экземпляров
Genre (Жанр)	
id	Первичный ключ
name	Название
Position (Должность)	
id	Первичный ключ
name	Название
EventLog (Журнал возврата/выдачи книг)	
id	Первичный ключ
book	Книга
reader	Читатель
employee	Сотрудник
action	Действие
created_at	Дата и время создания записи

сущности в существующие таблицы или создать новые. После генерации отображения можно будет создавать запросы к базе данных и создавать новые экземпляры классовых сущностей.

2.4.1 Объявление сущностей

Каждая сущность принадлежит одной базе данных. Поэтому до объявления сущностей должен быть создан объект базы данных:

```
from pony.orm import *

db = Database()

class MyFirstEntity(db.Entity):
        first_attribute = Required(str)
```

Объект типа *Database* имеет атрибут *Entity*, который используется как базовый класс для всех классовых сущностей. Каждая новая объявляемая классовая сущность должна быть дочерним классом по отношению к *Entity*-классу.

2.4.2 Атрибуты сущностей

Атрибуты сущности указываются как атрибуты внутри класса, используя следующий синтаксис:

```
        attr_name = kind(type, options)
```

Например,

```
class Customer(db.Entity):
        name = Required(str)
        email = Required(str, unique=True)
```

В скобках после типа атрибута Вы можете указать его опции. Существуют следующие типы атрибутов.

1 Required.
2 Optinal.
3 PrimaryKey.
4 Set.

2.4.3 Обязательность и опциональность

Обычно большинство атрибутов имеют тип *Required* или *Optional*. Если атрибут определен как *Required*, то он всегда должен иметь значение. Если атрибут имеет тип *Optional*, то он может быть пустым. Если Вы хотите, чтобы значение атрибута было уникальным, то Вы можете указать опцию атрибута *unique* со значением *True*.

Тип *PrimaryKey* определяет атрибут сущности, который используется как первичный ключ таблицы базы данных. Каждая сущность должна всегда иметь первичный ключ. Если первичный ключ явно не определен программистом, то PonyORM его автоматически создаст. Следующие два примера эквивалентны.

```
class Product(db.Entity):
        name = Required(str, unique=True)
        ...
```

```
class Product(db.Entity):
        id = PrimaryKey(int, auto=True)
        name = Required(str, unique=True)
        ...
```

Первичный ключ, который создал PonyORM, автоматически имеет по умолчанию имя *id* и тип *int*. Опция *auto=true* означает, что значение атрибута будет назначаться автоматически, используя счетчики или специальный объект-последовательность базы данных (зависит от СУБД).

Вы можете самостоятельно указать тип и имя первичного ключа. Например, Вы можете создать сущность *Account* (Учетная запись) и объявить адрес электронной почты первичным ключом.

```
class Account(db.Entity):
        email = PrimaryKey(str)
        name = Required(str)
```

2.4.4 Set

Атрибут типа *Set* представляет собой коллекцию. Также он часто определяет вторую сторону отношений между классовыми сущностями. Данный тип используется для создания связи «один-ко-многим». В настоящее время PonyORM не позволяет использовать тип *Set* в качестве первичного ключа.

2.4.5 Составные ключи и индексы

Используя *composite_index()*, можно создавать составные индексы для ускорения извлечения данных. Составной индекс может объединять два или более атрибута:

```
class Person(db.Entity):
        first_name = Required(str)
        last_name = Requeired(str)
        patronymic = Optional(str)
        composite_index(first_name, last_name, patronymic)
```

Если Вы хотите создать неуникальный индекс только для одной колонки, то можете указать опцию *index* при объявлении атрибута.

2.4.6 Типы данных атрибутов

PonyORM поддерживает следующие типы:

- *str* – строковый тип;
- *unicode* – строковый тип формата Юникод;
- *int* – целочисленный тип данных;
- *float* – вещественный тип данных;
- *Decimal* – тип данных с фиксированной точкой;
- *datetime* – датавременной тип данных;
- *date* – тип хранит только дату;
- *time* – тип хранит только время;
- *timedelta* – тип интервала времени;
- *bool* – логический тип данных;
- *bytes* – последовательность байтов;
- *LongStr* – длинные строки;
- *LongUnicode* – длинные строки в формате Юникод;
- *UUID* – тип, поддерживающий одноименный стандарт для создания идентификаторов;
- *Json* – текстовый формат сериализации данных;
- *IntArray* – целочисленный массив;
- *StrArray* – строковый массив;
- *FloatArray* – вещественный массив.

2.4.7 Наследование

Объявление иерархии сущностей в PonyORM

Наследование сущностей схоже наследованию обычных классов Python. Давайте рассмотрим пример, где сущности *Student* и *Professor* являются дочерними классами по отношению к классу *Person*:

```
class Person(db.Entity):
  name = Required(str)
```

```
class Student(Person):
  group = Required(str)
```

```
class Professor(Person):
  degree = Required(str)
  department = Required(str)
```

Все атрибуты и связи базовой сущности *Person* наследуются его детьми.

В PonyORM запрос экземпляров базового класса возвращает корректные экземпляры сущностей:

```
for p in Person.select():
  if isinstance(p, Professor):
    print(p.name, p.degree)
  elif isinstance(p, Student):
    print(p.name, p.group)
  else:
    print(p.name)
```

С версии 0.7.7 *instance* может быть использован внутри запроса:

```
staff = select(p for p in Person if not isinstance(p, Student))
```

Данный пример вернет всех людей, которые не являются студентами.

Реализация наследования в PonyORM

PonyORM для реализации отображения наследования в реляционную модель использует паттерн «Наследование с одной таблицей». Как уже говорилось, данное решение использует дополнительную колонку, дискриминатор для определения типа, в который строка будет отображена. По умолчанию он представляет собой колонку строкового типа, в котором указывается имя сущности.

По умолчанию PonyORM создает атрибут *classtype* для каждой сущности, которая является частью иерархии наследования. Вы можете использовать свои названия и тип дискриминатора. Если Вы изменили тип колонки-дискриминатора, то должны указать значение _*discrimintator*_ для каждой сущности.

Рассмотрим описанный выше пример, для которого создадим свой дискриминатор, имеющий целый тип:

```
class Person(db.Entity):
  cls_id = Discriminator(int)
  _discriminator_ = 1
  ...
```

```
class Student(Person):
  _discriminator_ = 2
  ...
```

```
class Professor(Person):
  _discriminator_ = 3
  ...
```

Множественное наследование

PonyORM поддерживает множественное наследование. Если Вы используете множественное наследование, тогда родительские классы определяемой сущности должны быть потомками базовой сущности *db.Entity*.

Рассмотрим пример, в котором программист может выполнять роль учителя. Для этого представим сущность *Teacher* и продолжим иерархию до классов *Professor* и *TeacherProgrammer*. Сущность *TeacherProgrammer* является дочерним классом от двух классов: *Programmer* и *Teacher*.

```
class Person(db.Entity):
name = Required(str)

class Programmer(Person):
...

class Teacher(Person):
...

class Professor(Teacher):
...

class TeacherProgrammer(Programmer, Teacher):
```

Сущность *TeacherProgrammer* является потомком двух классов: *Teacher* и *Programmer*, а значит унаследует их атрибуты. Множественное наследование возможно, если обе сущности *Teacher* и *Programmer* являются потомками базовой сущности *db.Entity*.

Наследование является очень мощным средством, которое должно быть использовано рационально.

2.4.8 Настройка преобразования

Когда PonyORM создает таблицы согласно определениям сущностей, он использует имена сущностей в качестве названия таблицы и названия атрибутов в качестве названия колонок. Но можно переопределить данное поведение.

Имена таблиц не всегда имеют те же названия, что и имена сущностей. В MySQL, PostgreSQL и CockroachDB по умолчанию получают имена, равные именам сущностей в нижнем регистре, в Oracle – в верхнем. Вы можете всегда узнать имена таблиц, обратившись к атрибуту *_table_* сущности. Если Вы хотите установить свое имя таблицы, используйте данный атрибут:

```
class Person(db.Entity):
  _table_ = "person_table"
  name = Required(str)
```

Также Вы сможете установить схему:

```
class Person(db.Entity):
  _table_ = ("my_schema", "person_table")
  name = Required(str)
```

Если Вы хотите назначить свое имя колонки, то используйте опцию *column*:

```
class Person(db.Entity):
  _table_ = "person_table"
  name = Required(str, column="person_name")
```

Для составных атрибутов используйте опцию *columns* – список имен колонок:

```
class Course(db.Entity):
  name = Required(str)
  semester = Required(int)
  lectures = Set("Lecture")
  PrimaryKey(name, semester)
```

```
class Lecture(db.Entity):
  date = Required(datetime)
  course = Required(Course, columns=["name_of_course", "semester"])
```

В этом примере мы переопределили название колонок для составного атрибута *Lecture.course*. По умолчанию PonyORM определяет следующие названия колонок: «couse_ name» и «course_semestr». PonyORM совмещает название сущности и название атрибута для того, чтобы облегчить понимание.

Если Вам необходимо установить название колонок для слабой сущности в отношении «многие-ко-многим», Вам необходимо указать опцию *column* или *columns* для атрибута типа *Set*. Рассмотрим следующий пример:

```
class Student(db.Entity):
  name = Required(str)
  courses = Set("Course")
```

```
class Course(db.Entity):
  name = Required(str)
  semester = Required(int)
  students = Set(Student)
  PrimaryKey(name, semester)
```

По умолчанию для хранения связи типа «многие-ко-многим» между сущностями *Student* и *Course* PonyORM создаст промежуточную таблицу для слабой сущности *Course_Student*. Эти таблицы будут иметь три колонки: *course_name*, *course_semestr* и *student*. Давайте рассмотрим, как изменить название промежуточной таблицы на *Study_Plans*, которая будет содержать колонки: *course*, *semestr* и *student_id*. Это можно сделать следующим способом:

```
class Student(db.Entity):
  name = Required(str)
  courses = Set("Course", table="Study_Plans",
                columns=["course", "semester"]))
```

```
class Course(db.Entity):
  name = Required(str)
  semester = Required(int)
  students = Set(Student, column="student_id")
  PrimaryKey(name, semester)
```

2.4.9 Определение сущностей абонентского отдела библиотеки

В данном пункте мы определим сущности без атрибутов, характеризующих связи между сущностями. Вопросу описания отношений будет посвящен отдельный раздел.

Согласно диаграмме классов модели предметной области (рисунок 2.1), определим все классы. Они должны либо напрямую, либо косвенно быть дочерними классами по отношению к классу *db.Entity*.

Человек

```
class Person(db.Entity):
  first_name = Required(str)
  last_name = Required(str)
  patronymic = Optional(str)

  @property
  def full_name(self):
    return ' '.join([self.last_name,  self.first_name, self.patronymic])
```

В теле класса *Человек* (*Person*) определили три атрибута: *first_name* (*имя*), *last_name* (*фамилия*) и *patronymic* (*отчество*). Необходимо отметить, что имя и фамилия являются обязательными атрибутами, а отчество – нет. По умолчанию PonyORM создаст первичный ключ с именем *id* и типом *int*. Добавленный декорированный метод *full_name ()* возвращает полное имя человека и будет использован позже в запросах.

Далее три класса будут явными наследниками класса *Человек* (*Person*), поэтому унаследуют от него атрибуты ФИО и полное имя.

Автор

```
class Author(person.Person):
        pass
```

Пока класс *Автор* (*Author*) никак не специализирует и не расширяет класс *Человек* (*Person*). Позже мы добавим в него атрибут, ответственный за связь данного класса с классом *Книга* (*Book*). Это отношение будет показывать, какие книги были написаны каким автором. **Сотрудник**

```
class Employee(Person):
        pass
```

Аналогичная ситуация с классом *Сотрудник* (*Employee*). Однако далее мы добавим отношения, которые определят, какие книги данный сотрудник выдавал и кому, а также в какой должности он работает.

Читатель

```
class Reader(Person):
        pass
```

Позже данный класс будет иметь атрибут, который будет указывать, какие книги и когда брал данный читатель.

Жанр

```
class Genre(db.Entity):
  name = Required(str, unique=True)
```

Класс *Жанр* (*Genre*) содержит пока единственный атрибут *name* (название) с ограничением уникальности и обязательности. Далее придется

(так в PonyORM регламентировано) указать связь с книгами, которая будет определять, какие книги написаны в данном жанре.

Должность

```
class Position(db.Entity):
        id = PrimaryKey(int, auto=True)
        name = Required(str, unique=True)
```

Для разнообразия в явном виде добавим первичный ключ с именем *id* и типом *int* и добавим ему свойство автоинкрементности. Атрибут *name* имеет опцию обязательного присутствия значения, которое должно быть уникальным. Далее добавим атрибут-связь с сотрудниками. Он будет отвечать за то, какие сотрудники работают в данной должности.

Книга

```
class Book(db.Entity):
  name = Required(str)
  year = Required(int, min=1500)
  number = Required(int, min=1)
```

Сущность *Книга* (*Book*) включает пока три атрибута: обязательное название книги (*name*), год издания (*year*) и количество экземпляров данной книги (*number*). В описаниях атрибутов *year* и *number* указано свойство *min*, которое показывает нижнюю границу значения атрибута. Книги, выпущенные до 1500 года, создать будет нельзя. Диаграмма классов (рисунок 2.1) показывает, что у книги есть связи с авторами, жанрами и журналом выдачи/возврата книги. Соответствующие атрибуты будут добавлены позже.

Действие

```
class Action(db.Entity):
  name = Required(str, unique=True)
```

Класс *Действие* (*Action*) представляет собой простой справочник, состоящий из двух объектов: выдача книги и ее возврат. За это отвечает атрибут с обязательным уникальным значением *name* (*название*). Конечно, данный класс чаще не создают. Вместо него определяют в журнале выдачи/возврата атрибут, например, с логическим значением. Но мы за счет введения новой сущности предвкушаем, что могут добавиться новые действия, например, забронировано или др. В этом случае менять схему базы данных не понадобится, стоит только добавить новое название действия в соответствующую таблицу.

Журнал выдачи/возврата книг

```
class EventLog(db.Entity):
  created_at = Required(datetime, default=datetime.utcnow())
```

Данная сущность выглядит бедно, так как в ней отсутствуют атрибуты, ответственные за то, какой сотрудник выдал или принял какую книгу от какого читателя. Над этим мы задумаемся в следующем разделе, а пока отметим атрибут *created_at*. Данный атрибут показывает время и дату создания записи в журнале. Стоит отметить, что у данного атрибута указана опция *default*. Если не будут указаны время и дата пользователем, то значением по умолчанию будет результат выражения *datetime.utcnow()*, которое возвращает текущую дату и время.

2.5 Отношения между сущностями

Сущности, как правило, связаны друг с другом. Любое отношение между двумя сущностями определяется с помощью атрибутов, которые определяют обе стороны связи. Это важный момент. Если Вы используете PonyORM и описываете связь между двумя сущностями, но она будет использоваться только в одном направлении, то все равно Вы должны ее описать как двунаправленную, т. е. определить атрибуты в двух связанных сущностях. Не многие современные ORM, например, Django, ActiveRectord в Ruby on Rails, Hibirnate требуют обязательного описания отношения с обеих сторон, но на практике часто используется только одно направление. Обязательные двунаправленные отношения в PonyORM приводят к тому, что при обновлении объекта с одной стороны отношения, с другой стороны изменения произойдут автоматически.

Существует три типа отношений: «один-к-одному», «один-ко-многим» и «многие-ко-многим». Некоторые считают, что отношение «один-к-одному» используется редко, так как, если две сущности связаны отношением «один-к-одному», это означает, что они могут быть объединены в одну сущность. А если Ваша диаграмма данных перегружена связями «один-к-одному», это означает, что пришла пора пересмотреть диаграмму. Однако эти утверждения могут быть с легкостью парированы. Имеет смысл не объединять две сущности в одну, если они семантически представляют из себя разные сущности. Например, пусть у человека имеется один паспорт гражданина РФ, и один паспорт может быть выдан только одному человеку. На лицо эти две сущности, *Паспорт* и *Человек*, связаны отношением с кратностью «один-к-одному». Например, мы объединим эти две сущности в одну – *Человек*. Но семантически номер, серия, кем и когда выдан документ не являются свойствами человека. А что делать с опциональностью отношения?! Поэтому, если в модели предметной области у Вас имеются отношения кратностью «один-к-одному», это не повод бездумно их объединять в одну сущность.

2.5.1 Отношение «один-к-одному»

Чтобы создать связь «один-к-одному», Вам необходимо определить два атрибута на каждой стороне отношения, которые могут иметь следующую пару типов: *Optional-Required* или *Optional-Optional*.

```
class Person(db.Entity):
  ...
  passport = Optional("Passport")

class Passport(db.Entity):
  ...
  person = Required(Person)
```

Стоит отметить, зачем при определении атрибута *passport* в классе *Person* типа *Passport* мы заключили в кавычки. Дело в том, что класс *Passport* определен ниже данной строки и пока не существует.

В данном примере определены две сущности: *Человек* (*Person*) и *Паспорт* (*Passport*). В сущности *Человек* объявлен атрибут *passport*, который имеет атрибут *Optional*, в сущности *Паспорт* – атрибут *person*, который обязательно должен быть проинициализирован. Другими словами, человека мы можем создать без указания его паспорта, а паспорт без указания человека – нет.

Определение одновременно двух атрибутов типа *Required* недопустимо, так как не имеет смысла.

2.5.2 Отношение «один-ко-многим»

Рассмотрим пример:

```
class Position(db.Entity):
  employees = Set("Employee")
```

```
class Employee(db.Entity):
  positon = Required(Position)
```

В этом примере между классами *Position* (*Должность*) и *Employee* (*Сотрудник*) мы определили отношение кратности «один-ко-многим». В одной должности может работать несколько сотрудников, но сотрудник может работать только в одной должности. Экземпляр класса *Employee* не может существовать без указания должности. Если мы захотим создать сотрудника без указания его должности, то мы можем определить атрибут *position* как *Optional*:

```
class Position(db.Entity):
  employees = Set("Employee")
```

```
class Employee(db.Entity):
  positon = Optional(Position)
```

2.5.3 Отношение «многие-ко-многим»

Для того чтобы создать отношение кратности «многие-ко-многим», необходимо создать атрибуты типа *Set* в каждой сущности.

```
class Book(db.Entity):
  authors = Set("Author")
```

```
class Author(db.Entity):
  books = Set("Book")
```

В данном примере в обеих сущностях указаны атрибуты типа *Set*, определяющие отношение «многие-ко-многим», то есть одна книга может быть написана группой авторов, и один автор может написать несколько книг. На уровне СУБД такое определение приведет к тому, что будет создана промежуточная таблица, в которой будут два внешних ключа к каждой таблице сущности.

2.5.4 Самоссылаемость

Могут существовать связи, на концах которых находятся объекты одного типа. Такие отношения могут быть симметричными или асимметричными. Асимметричное отношение определяется двумя атрибутами, которые принадлежат одной и той же сущности. Специфика симметричных отношений заключается в том, что у сущности указан только один атрибут отношения, и этот атрибут определяет обе стороны отношения. Такие отношения могут быть «один-к-одному» или «многие-ко-многим». Рассмотрим пример:

```
class Person(db.Entity):
  name = Required(str)
  spouse = Optional("Person", reverse="spouse")
  friends = Set("Person", reverse="friends")
  manager = Optional("Person", reverse="employees")
  employees = Set("Person", reverse="manager")
```

В данном листинге мы объявили сущность *Человек* (*Person*). Он включает обязательный атрибут *name* (ФИО) и атрибуты, которые ссылаются на объект(ы) типа *Person*.

• Атрибут *spouse* (*Супруг*) указывает на супруга(у) типа *Person*. Например, пусть имеется пара женатых людей: Иван и Мария. У Марии атрибут *spouse* ссылается на Ивана, а у Ивана *spouse* ссылается на Марию. Стоит заметить, что данный атрибут определяет симметричное отношение «один-к-одному».

• Атрибут *friends* определяет круг друзей человека. Например, есть дружная компания, которая состоит из Ивана, Михаила и Семена. Это означает, что у Ивана во множестве *friends* находятся Михаил и Семен, у Михаила – Иван и Семен, а у Семена – Иван и Михаил. Но, например, у Ивана могут быть друзья (элементы множества *friends*), которые отсутствуют во множестве друзей Семена и Михаила. Данная симметричная связь имеет кратность «многие-ко-многим».

• Два атрибута, *manager* и *employees*, определяют асимметричную связь «начальник – подчиненные». Пусть у нас имеется объект типа *Person*. Его атрибут *manager* (если это не пустая ссылка) ссылается на его начальника, имеющего тип также *Person*, а атрибут *employees* – множество объектов *Person*, которые находятся у него в подчинении. Стоит заметить, что *manager* имеет тип *Optional*. Это означает, что у человека может и не быть начальника. А множество *employees* может быть пустым, нет подчиненных.

У атрибутов самоссылаемости должна быть определена опция *reverse* – название атрибута на второй стороне отношения.

2.5.5 Несколько отношений между двумя сущностями

Когда две сущности могут иметь более чем одну связь между собой, PonyORM требует указать обратные атрибуты. Это необходимо для того, чтобы PonyORM знал, какая пара атрибутов связана друг с другом. Рассмотрим пример, в котором пользователь может создавать твиты и добавлять их в избранное.

```
class User(db.Entity):
  tweets = Set("Tweet", reverse="author")
  favorites = Set("Tweet", reverse="favorited")
```

```
class Tweet(db.Entity):
  author = Required(User, reverse="tweets")
  favorited = Set(User, reverse="favorites")
```

В этом примере мы создали две сущности: *User* (*Пользователь*) и *Tweet* (*Твит*). Пользователь может создавать твиты. Это отношение

определяется парой атрибутов *User.tweets* и *Tweet.author*. Кратность этой ассоциации – «многие-к-одному». У пользователя могут быть любимые твиты (не обязательно свои). Данное отношение формируется парой атрибутов *User.favorites* и *Tweet.favorited*. Кратность этой ассоциации – «многие-ко-многим». Мы указали опцию *reverse* у всех наших атрибутов обеих сущностей. Если Вы попытаетесь сгенерировать преобразование для данных сущностей без использования опции *reverse*, Вы получите ошибку:

```
pony.orm.core.ERDiagramError: "Ambiguous reverse attribute for Tweet.author".
```

Это означает, что в этом случае атрибут *author* может быть технически связан с атрибутом *tweets* или *favorited*, и у PonyORM отсутствует информация, чтобы разобраться с каким. А за счет использования опции *reverse* мы однозначно связали атрибуты с их парами.

2.5.6 Определение отношений между сущностями в модели абонентского отдела библиотеки

Пришло время полностью описать сущности нашей предметной области с атрибутами, определяющими связи между ними. Сущность *Person* не изменилась, так как не участвует в связи с другими сущностями.

```
class Employee(Person):
  position = Required('Position')
  event_logs = Set('EventLog')
```

Сотрудник должен работать в определенной должности. Эту ассоциацию определяет атрибут *position*, имеющий тип *Position*. Кратность данной связи – «один-ко-многим». Сотрудник может выдавать книги читателю или принимать от него книги. Данный факт заносится в журнал возврата/выдачи книги (*EventLog*). Поэтому в сущности *Сотрудник* появился атрибут, ответственный за данную связь. Так как одну и ту же книгу может выдать разный сотрудник в разное время и разным читателям, то кратность данной связи – «многие-ко-многим».

```
class Author(person.Person):
  books = Set('Book')
```

У автора появился атрибут *books*, который показывает, какие книги кем написаны. Известно, что книги могут быть написаны в соавторстве, а один и тот же человек может быть автором нескольких книг. Это определяет тип атрибутов на обеих концах связи как *Set* («многие-ко-многим»).

```
class Reader(Person):
  event_logs = Set('EventLog')
```

Атрибут *event_ logs* создает связь, которая семантически представима как читательский билет. Но на практике это всего лишь связь с журналом выдачи/возврата книг. Тип связи – «многие-ко-многим».

```
class Book(db.Entity):
  name = Required(str)
  authors = Set('Author')
  genres = Set('Genre')
  year = Required(int, min=1500)
  number = Required(int, min=0)
  event_logs = Set('EventLog')
```

Данная сущность богата на атрибуты-связи. У книги может быть несколько авторов (атрибут *authors*), она может быть написана в нескольких жанрах (*genres*), и ее могут выдавать разные сотрудники и возвращать разные читатели в разное время (атрибут *event_ logs*).

```
class Genre(db.Entity):
  name = Required(str, unique=True)
  books = Set('Book')
```

Сущность *Жанр* (*Genre*) содержит атрибут *books*, представляющий из себя множество книг, которые написаны в данном жанре (связь – «многие-ко-многим»).

```
class Position(db.Entity):
  id = PrimaryKey(int, auto=True)
  name = Required(str, unique=True)
  employees = Set('Employee')
```

В сущности *Должность* (*Position*) появился редко используемый в нашей предметной области атрибут *employees*, который определяет круг сотрудников, работающих в данной должности.

```
class Action(db.Entity):
        name = Required(str, unique=True)
        event_logs = Set('EventLog')
```

Мы никогда не будем использовать в нашей системе вариант использования «Получение всех записей в журнале выдачи/возврата книг по действию». Однако по правилам PonyORM мы всегда должны определять двунаправленные связи, поэтому в данной сущности появился атрибут *event_ logs*.

```
class EventLog(db.Entity):
  book = Required('Book')
  reader = Required('Reader')
  employee = Required('Employee')
  action = Required('Action')
  created_at = Required(datetime, default=datetime.utcnow())
```

Журнал выдачи и возврата книг содержит достаточно много атрибутов, определяющих связи: *book* – обязательный атрибут, который говорит, какая книга была выдана или возвращена, *reader* – кто вернул или взял книгу, *employee* – какой сотрудник выдал или принял книгу, и *action* – какое действие (возврат или выдача).

2.6 Привязка к базе данных

Объект типа *Database* управляет соединениями с базой данных, используя пул соединений. Он потокобезопасен и может быть разделен между потоками в приложении. Объект *Database* позволяет работать с базой данных напрямую с использованием SQL (метод *execute*), но большую часть времени Вы будете работать с сущностями и позволять PonyORM генерировать SQL-код самостоятельно. Можно работать с несколькими базами данных одновременно, имея отдельный объект *Database* для каждой базы данных, но каждый объект всегда принадлежит одной базе данных.

Объект базы данных *Database* имеет метод *bind*. Он необходим для присоединения сущностей к базе данных. Для SQLite базы данных можно написать следующее:

```
  db.bind(provider="sqlite", filename=":memory:")
```

В настоящее время PonyORM поддерживает 5 баз данных: SQLite, PostgreSQL, MySQL, Oracle, CockroachDB.

Для SQLite имеется параметр *filename*, который может принимать значение *:memory:* или путь до файла базы данных. Значение *:memory:* определяет, что база данных будет создана в оперативной памяти и после окончания работы приложения будет удалена. Для работы с файловой базой данных необходимо написать следующий код:

```
db.bind(provider="sqlite", filename="database.sqlite", create_db=True)
```

В этом случае, если файл базы данных не существовал, то он будет создан.

Если Вы используете другие СУБД, необходимо установить к ним адаптеры. Далее Вы можете создавать связки следующим образом:

Для PostgreSQL:

```
db.bind(provider='postgres', user='', password='', host='', database='')
```

Для MySQL:

```
db.bind(provider='mysql', host='', user='', passwd='', db='')
```

2.7 Преобразование сущностей в таблицы базы данных

Теперь необходимо создать таблицы базы данных, где будут храниться Ваши данные. Для этих целей используется метод *generate_mapping()* класса *Database*.

```
db.generate_mapping(create_tables=True)
```

Параметр *create_tables* сигнализирует о том, что если таблица не была создана ранее, то она должна быть создана запросом CREATE TABLE.

Необходимо отметить, что все сущности должны быть определены до вызова этого метода.

После определения сущностей предметной области абонентского отдела библиотеки и вызова метода *generate_mapping ()* схема базы данных будет иметь вид (рисунок 2.2).

2.8 Режим отладки

Если Вы хотите видеть в консоли SQL-код, который генерирует PonyORM, используйте функцию *set_sql_debug(True)*. Если эта команда была вызвана до *generate_mapping()*, то Вы увидите, какие таблицы создаются.

2.9 Транзакции и db_session

Транзакции в базах данных – логическая единица работы, которая состоит из одной или нескольких запросов. Транзакции атомарны. Это означает, что изменения в базу данных будут внесены только тогда, когда все запросы в рамках транзакции выполнятся успешно, в противном случае база данных примет состояние до момента транзакции.

PonyORM поддерживает управление транзакциями, используя механизм сессий.

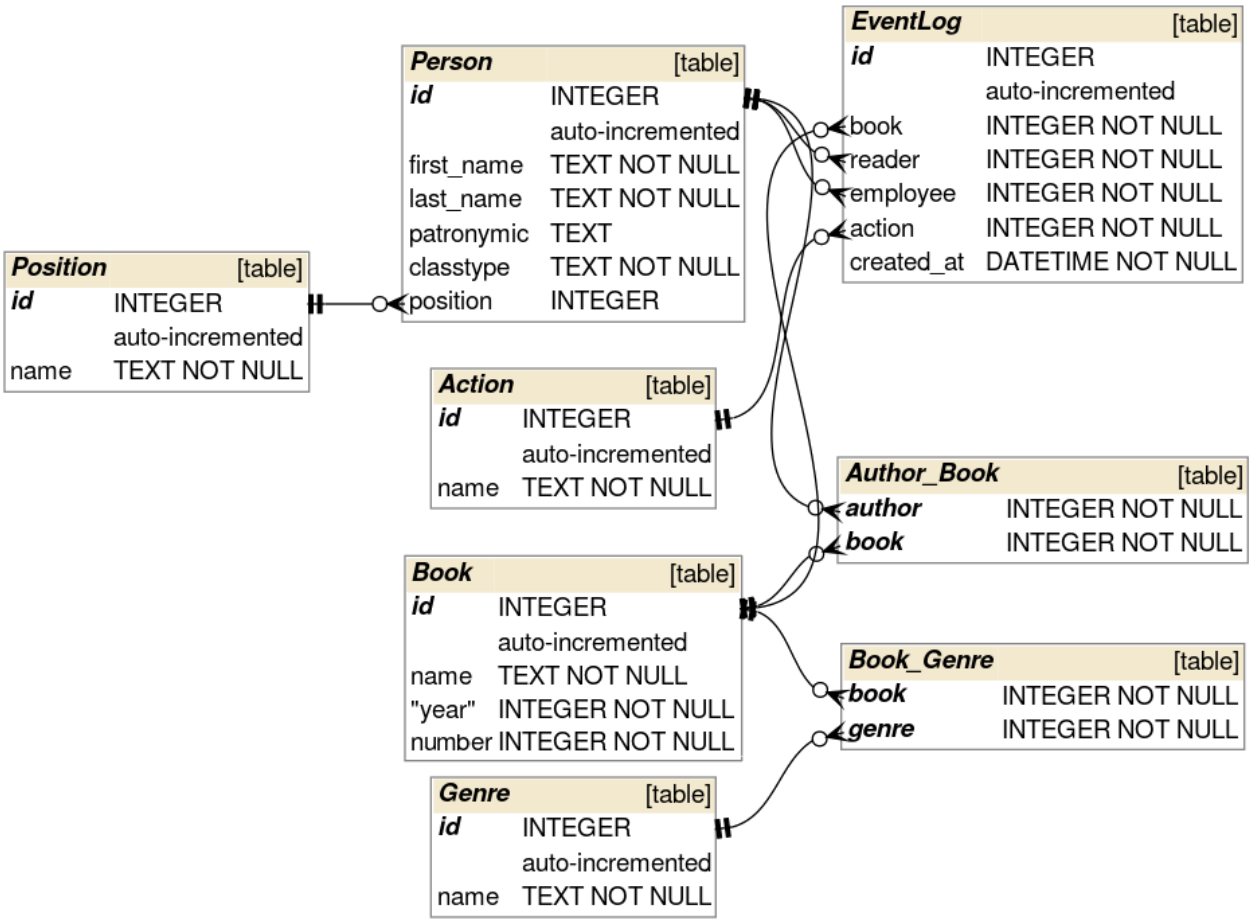

Рисунок 2.2 – Схема базы данных модели предметной области

2.9.1 Работа с db_session

Код, который взаимодействует с базой данных, должен быть помещен в сессию. Сессия устанавливает границы диалога приложения и базы данных. Каждая нить (поток) приложения, которая работает с базой данных, устанавливает сеанс с базой данных и использует отдельный экземпляр *Карты присутствия* [2]. Данный паттерн предназначен для хранения и управления объектами, которые были когда-то запрошены из БД. Он выполняет в том числе и функцию кэширования.

Для работы с базой данных на основе сессии необходимо использовать декорирование *db_session* или использовать менеджер контекста. Когда сессия закрывается, то выполняются следующие действия:

1) если не было никаких исключений, то изменения, вызванные транзакцией закрепляются в БД, иначе инициируется процесс отката к состоянию до исполнения транзакции;

2) соединение возвращается в пул соединений;

3) очищается кэш карты присутствия.

Если вы забыли указать *db_session* там, где это необходимо, PonyORM создаст исключение:

```
TransactionError: db_session is required when working with the database.
```

Пример с использованием декоратора *db_session*:

```
@db_session
def check_person(first_name, last_name, patronymic):
  return Person.exists(first_name=first_name,
    last_name=last_name, patronymic=patronymic)
```

В этом примере создается функция (метод), декорированная *db_session*, которая проверяет, существует ли человек с заданными ФИО.

Пример с использованием менеджера контекста:

```
def my_fun():
  ...
  with db_session:
    u = Person.get(first_name=first_name, last_name=last_name,
                   patronymic=patronymic)
    ...
```

В данном примере определяется функция, в которой с использованием менеджера контекста делается попытка получить экземпляр класса *Person* по его ФИО.

Когда PonyORM читает объекты из базы данных, он помещает их в карту присутствия. Позже, когда обновляются атрибуты объекта, создается или удаляется объект, изменения будут накапливаться сначала в карте присутствия. Изменения будут сохранены в базе данных при закреплении транзакции или до вызова метода *select*, *get*, *exists* или *execute*.

2.9.2 db_session и объем транзакций

Обычно у Вас будет одна транзакция в рамках сессии. В PonyORM не предусмотрено явной команды для начала транзакции. Транзакция начинается с первого отправленного SQL-запроса. Перед отправкой первого запроса PonyORM получает соединение с базой данных из пула соединений. Любые следующие запросы будут выполняться в контексте данной транзакции. Транзакция заканчивается, когда будут вызваны явно или нет методы *commit()* или *rollback ()* или выйдем из *db_session*.

```
@db_session
def func():
  # Начало транзакции
  # Получение человека по индентификатору 123
  p = person[123]
  # Изменение имени
  p.first_name = 'Иван'
```

```
# commit() будет вызван автоматически
# Автоматически очистится кэш сессии
# Соединение с БД будет автоматически возвращено в пул соединений
```

2.9.3 Несколько транзакций в одной сессии

Если Вам необходима более чем одна транзакция, то Вы можете вызывать методы *commit ()* и *rollback ()* в любое время в рамках сессии. В результате следующий запрос будет помещен в новую транзакцию. Карта присутствия будет сохранять данные после каждого явного вызова *commit ()*, но если Вы вызовите *rollback ()*, то кэш будет очищен.

```
@db_session
def func1():
  b1 = Book[123]
  b1.year = 1990
  commit()
  # Первая транзакция закреплена
  # Начало новой транзакции
  b2 = Book[456]
  b2.name = 'Война и мир'
```

2.9.4 Вложенные db_session

Что произойдет, если Вы введете рекурсивную область видимости *db_ session*, например, вызвав функцию, декорированную *db_session*, из другой функции, которая тоже декорирована *db_session*? PonyORM не будет создавать новую сессию, а вместо этого обе функции будут использовать одну. Сессия будет закрыта после выхода из области внешнего декоратора или внешнего менеджера контекста.

PonyORM дает возможность настраивать сессии путем определения значений параметров декоратора *db_ session*. Основные параметры сессии:

• **allowed_exceptions(list)**. Список исключений, при возникновении которых не вызывается откат транзакции;

• **immediate(bool)**. Сообщает PonyORM о начале транзакции с базой данных. Некоторые базы данных (например, SQLite, PostgreSQL) запускают транзакцию только тогда, когда в базу данных отправляется модифицирующий запрос (UPDATE, INSERT, DELETE) и не запускают ее для SELECT. Если Вам нужно начать транзакцию с SELECT, вам следует установить значение True;

• **optimistic(boll)**. По умолчанию – *True*. Когда *optimistic = False*, оптимистические проверки не будут добавляться к запросам в этой *db_session*;

• **retry(int)**. Указывает количество попыток запуска текущей транзакции. Декорированная функция не должна явно вызывать функции *commit ()* или *rollback ()*. Когда этот параметр указан, PonyORM перехватывает исключение *TransactionError* (и всех его потомков) и перезапускает текущую транзакцию;

• **retry_exceptions** определяет список исключений, которые приведут к перезапуску транзакции. По умолчанию этот параметр равен [*Transac-tionError*];

• **strict**. При значении *True* кэш будет очищен при выходе из *db_session*. Если Вы попытаетесь получить доступ к объекту после окончания сеанса, Вы получите исключение *pony.orm.core.DatabaseSessionIsOver*. Обычно PonyORM настоятельно рекомендует Вам работать с объектами сущностей только внутри *db_session*. Но некоторые пользователи PonyORM хотят получить доступ к извлеченным объектам в режиме «Только для чтения» даже после того, как *db_session* закончится;

• **sql_debug**. Когда *sql_debug = True*, записываются операторы SQL в консоль или в файл журнала;

• **show_values**. При значении *True* параметры запроса будут регистрироваться в дополнение к тексту SQL.

Что произойдет, если внутренний *db_session* будет иметь настройки, отличные от внешнего *db_session*? Например, внешний *db_session* имеет настройки по умолчанию, а внутренний определен как *db_session(optimistic= False)*.

В настоящее время PonyORM проверяет внутренние параметры *db_session* и выполняет одно из следующих действий:

1) если внутренний *db_session* имеет несовместимые значения опций со значениями внешнего *db_session*, то PonyORM создаст исключение;

2) для опции *sql_debug* PonyORM использует новое значение во внутреннем *db_session* и восстанавливает его при возврате во внешний *db_session*;

3) другие опции (*strict*, *optimistic*, *immediate*, *retry*) игнорируются во внутреннем *db_session*.

Если будет вызван *rollback* из внутреннего *db_session*, то произойдет откат транзакции и во внешнем *db_session*.

Некоторые СУБД поддерживают вложенные транзакции, но в настоящее время PonyORM их не реализует.

2.9.5 Кэш db_session

Для увеличения производительности PonyORM кэширует данные в несколько этапов. Он кэширует:

• результаты трансляции генераторов выражений. Если один и тот же запрос в виде генератора выражений используется несколько раз, то он будет транслирован в SQL только один раз; Этот кэш является глобальным для всей программы, а не только для сессии;

• созданные или загруженные из БД объекты. PonyORM сохраняет эти объекты в карте присутствия. Этот кэш очищается, когда *db_session* заканчивается, или вызывается *rollback*;

• результаты запроса. PonyORM возвращает результаты запроса из кэша, если точно такой же запрос уже исполнялся. Этот кэш очищается после того, как экземпляры сущности изменятся или удалятся. А также кэш очистится, если произойдет откат транзакции или выход из области *db_session*.

2.9.6 Работа с несколькими базами данных

PonyORM позволяет работать с несколькими базами данных одновременно. Следующий пример показывает, как PostgreSQL используется для хранения пользовательской информации, а MySQL – для хранения информации об адресах.

```
db1 = Database()

class User(db1.Entity):
...

db1.bind('postgres', ...)

db2 = Database()

class Address(db2.Entity):
...

db2.bind('mysql', ...)

@db_session
def do_something(user_id, address_id):
  u = User[user_id]
  a = Address[address_id]
...
```

При выходе из функции *do_something* PonyORM будет выполнять *commit* или *rollback* для обеих баз данных. Если Вам необходимо зафиксировать

изменения в одной базе данных перед выходом из функции, Вы можете использовать методы *db1.commit()* и *db2.commit()*.

2.9.7 Функции для работы с транзакциями

Существуют три функции верхнего уровня, которые Вы можете использовать для работы с транзакциями:

- *commit()* – закрепляет изменения в рамках транзакции в БД;
- *rollback()* – откатывает изменения в рамках транзакции;
- *flush()* – сохраняет все изменения из кэша *db_session* без их фиксации (commit).

Также существуют три аналогичные функции в пространстве имен *Data-base* объекта:

- *Database.commit()*;
- *Database.rollback()*;
- *Database.flush()*.

2.10 Работа с экземплярами сущностей

2.10.1 Создание экземпляров сущностей

Создание экземпляров сущностей в PonyORM выполняется аналогично созданию обычных объектов в Python. Например,

```
author = Author(first_name="Иван", last_name="Иванов",
                patronymic="Иванович")
```

При создании объекта в PonyORM все параметры должны быть указаны как ключевые аргументы. Если атрибут имеет значение по умолчанию или опционален, то он может быть опущен.

Все созданные экземпляры принадлежат текущему *db_session*. В некоторых ORM Вам необходимо вызвать метод *save()* для сохранения объекта в БД. Это неудобно, поскольку программист должен отслеживать, какие объекты были созданы или обновлены, и не забыть вызвать метод *save()* для каждого из них. PonyORM следит, какой объект был создан или обновлен и сохраняет или обновляет его в базе данных автоматически, когда текущая сессия завершается. Если необходимо сохранить вновь созданный объект перед выходом из сессии, то Вы можете сделать это, вызвав функции *flush()* или *commit()*.

2.10.2 Загрузка объектов из базы данных

Получение объектов по первичному ключу

Рассмотрим простейший случай получения объекта по значению его первичного ключа. Для этого необходимо указать значение первичного ключа в квадратных скобках после названия класса. Например, для извлечения автора с первичным ключом 123 Вы можете написать следующее выражение:

```
author = Author[123]
```

Подобный синтаксис также работает с составными ключами, только в этом случае необходимо передать кортеж значений составного ключа.

PonyORM генерирует исключение *ObjectNotFound*, если объект с указанным первичным ключом не был найден в БД.

Получение объекта по уникальному набору атрибутов

Если Вы хотите получить один объект не по первичному ключу, а по комбинации атрибутов, Вы можете использовать метод сущности *get()*. В большинстве случаев он используется для получения объекта по суррогатному уникальному ключу, но также может использоваться для поиска по любой другой комбинации атрибутов. В качестве параметров метода *get()* Вы должны указывать имена атрибутов и их значения. Например, если Вы хотите получить сотрудника по его полному имени, и Вы считаете, что БД содержит только одного сотрудника с указанным именем, то Вы можете написать:

```
employee = Emlpoyee.get(first_name='Петр', last_name='Петров',
                        patronymic='Петрович')
```

Если объект не был найден, то метод *get()* вернет *None*. Если будет найдено несколько объектов, то будет создано исключение *MultipleObjectsFound-Error*. Вы можете использовать метод *get()* для получения объекта по значению его первичного ключа. В этом случае, если объект не будет найден, но не будет брошено исключение, то метод *get()* вернет *None*.

Получение нескольких объектов

Чтобы извлечь несколько объектов из базы данных, Вы должны использовать метод сущности *select()*. Его аргумент – это лямбда-функция, которая имеет единственный параметр, определяющий экземпляр объекта в базе данных. Внутри этой функции Вы можете написать условия, по которым Вы хотите выбирать объекты. Например, если требуется найти всех читателей с фамилией «Иванов», то можете написать:

```
readers = Reader.select(lambda r: r.last_name == "Иванов")
```

Эта лямбда-функция не будет выполняться интерпретатором Python. Вместо этого она будет транслирована в следующий SQL-запрос:

```
SELECT "r"."id", "r"."first_name", "r"."second_name", "r"."patronymic",
"r"."classtype"
FROM "Person" "r"
WHERE "r"."last_name" = 'Иванов' AND "r"."classtype" IN ('Reader')
```

Метод *select()* возвращает экземпляр класса *Query*. Если Вы начнете перебирать этот объект, SQL-запрос будет отправлен в базу данных и Вы получите последовательность экземпляров сущности. Например, вот как Вы можете показать полные имена читателей:

```
for r in Reader.select(lambda r: r.last_name == "Иванов"):
  print(r.full_name)
```

Если Вы не хотите перебирать объекты, работая с экземпляром *Query*, а просто хотите получить сразу список объектов, Вы можете сделать это следующим образом:

```
readers = Reader.select(lambda r: r.last_name == "Иванов")[:]
```

или

```
readers = list(Reader.select(lambda r: r.last_name == "Иванов"))
```

Использование параметров в запросе

Вы можете использовать переменные в запросах. PonyORM передаст эти переменные в качестве параметров в SQL-запрос. Одним из важных преимуществ декларативного синтаксиса запросов в PonyORM является то, что он предлагает полную защиту от SQL-инъекций, поскольку все внешние параметры будут должным образом экранированы.

Например:

```
  x = 'Иванов'
  readers = Reader.select(lambda r: r.last_name == x)[:]
```

PonyORM преобразует данное выражение в следующий SQL-запрос:

```
SELECT "r"."id", "r"."first_name", "r"."second_name",
"r"."patronymic", "r"."classtype"
FROM "Person" "r"
WHERE "r"."last_name" = ? AND "r"."classtype" IN ('Reader')
['Иванов']
```

Таким образом, значение x будет передано в качестве параметра запроса SQL, что полностью исключает риск внедрения SQL-кода.

Сортировка результатов запроса

Если Вам нужно отсортировать объекты в определенном порядке, Вы можете использовать метод *Query.order_by()*. Например,

```
  Reader.select(lambda r: r.last_name == 'Иванов')
    .order_by(desc(Reader.first_name))
```

В этом примере мы получим читателей с фамилией «Иванов» в порядке убывания их имен. Вот SQL-запрос, сгенерированный для предыдущего примера:

```
SELECT "r"."id", "r"."first_name", "r"."second_name", "r"."patronymic",
"r"."classtype"
FROM "Person" "r"
WHERE "r"."last_name" = 'Иванов' AND
"r"."classtype" IN ('Reader') ORDER BY "r"."first_name" DESC
```

Метод *Query.order_by()* также может ожидать от пользователя лямбда-функцию в качестве параметра:

```
    Reader.select(lambda r: r.last_name == 'Иванов')
    .order_by(lambda r: desc(r.first_name))
```

Передача лямбда-функции методу позволяет использовать сложные выражения для сортировки. Например, вот как Вы можете отсортировать книги в порядке убывания их популярности среди читателей:

```
Book.select().order_by(lambda b: desc(count(b.event_logs)))
```

Чтобы отсортировать результат по нескольким атрибутам, Вам необходимо разделить их запятой. Например, если Вы хотите отсортировать читателей сначала по фамилии, затем при их совпадении по имени, а затем по отчеству, то это необходимо записать следующим образом:

```
  Reader.select().order_by(Reader.last_name, Reader.first_name,
    Reader.patronymic)
```

Тот же запрос, но с использованием лямбда-функции будет выглядеть так:

```
  Reader.select().order_by(lambda r: (r.last_name, r.first_name,
    r.patronymic))
```

Обратите внимание, что в соответствии с синтаксисом Python, если Вы возвращаете более одного элемента из лямбда-функции, вам нужно поместить их в круглые скобки (возвратить кортеж).

Ограничение количества выбранных элементов

Можно ограничить количество объектов, возвращаемых запросом, используя метод *Query.limit()* или более компактную нотацию фрагмента Python, работая со срезами. Например, вот как Вы можете получить десять самых популярных книжек:

```
Book.select().order_by(lambda b: desc(count(b.event_logs)))[:10]
```

Результатом среза является не объект запроса, а список экземпляров сущности.

Обход отношения

В PonyORM можно просматривать объектные отношения:

```
employee = Employee[123]
position = enmployee.position
print(position.name)
```

PonyORM пытается минимизировать количество запросов, отправляемых в базу данных. В приведенном выше примере, если запрошенный объект *Employee* уже был загружен в кэш, PonyORM вернет объект из кэша без отправки запроса СУБД. Но если объект еще не был загружен, PonyORM все равно не отправит запрос немедленно. Вместо этого он сначала создаст Proxy-объект. Proxy-объект – это объект, инициализированный только первичным ключом. PonyORM не знает, как этот объект будет использоваться, и всегда есть вероятность, что нужен только первичный ключ. В приведенном выше примере PonyORM получает объект типа *Position* из базы данных в третьей строке при доступе к атрибуту *name*, используя концепцию «ленивой загрузки» (Lazy Load).

Обход возможен и в направлении «ко-многим». Например, мы хотим узнать все жанры книги с идентификатором 12:

```
book = Book[12]
for genre in book.genres:
  print(genre.name)
```

2.10.3 Обновление объектов

Когда Вы присваиваете новые значения атрибутам объекта, не нужно сохранять каждый обновленный объект вручную. Изменения будут автоматически сохраняться в базе данных после выхода из области действия *db_session*.

Например, чтобы изменить название книги с первичным ключом 123, можно написать следующий код:

```
with db_session:
  Book[123].name = 'Война и мир'
```

Следующий пример показывает, как можно изменить несколько атрибутов одного и того же объекта:

```
with db_session:
  author = Author[12]
  author.first_name = 'Лев'
  author.last_name = 'Толстой'
  author.patronymic = 'Николаевич'
```

А также для изменения значений атрибутов сущности можно использовать метод *set()*. Следующий пример кода аналогичен предыдущему:

```
Author[12].set(first_name='Лев', last_name='Толстой',
   patronymic='Николаевич')
```

Метод *set()* может быть незаменим, когда нужно обновить сразу несколько атрибутов объекта из словаря:

```
Author[12].set(**dict_with_new_values)
```

Если нужно сохранить обновления в базе данных до завершения текущего сеанса, можете использовать функции *flush()* или *commit()*.

Необходимо отметить, что PonyORM всегда автоматически сохраняет изменения, накопленные в кэше *db_session*, перед тем, как выполнить следующие методы: *select ()*, *get ()*, *exist ()*, *execute ()* и *commit ()*.

2.10.4 Удаление объектов

Когда вызывается метод *delete ()* экземпляра сущности, PonyORM помечает объект как удаленный. Объект будет удален из базы данных во время следующей фиксации изменений.

Например, вот как мы можем удалить книгу с первичным ключом, равным 12:

```
Book[12].delete()
```

Массовое удаление

PonyORM поддерживает массовое удаление объектов с помощью функции *delete ()*. Таким образом, Вы можете удалить несколько объектов, не загружая их из БД:

```
delete(b for b in Book if b.year < 1800)
```

или

```
Book.select(lambda b: b.year < 1800).delete(bulk=True)
```

Данный код удалит из базы данных книги, изданные ранее 1800 года. PonyORM не будет загружать все книги старше 1800 года, а затем поодиночке их удалять. Будет транслирован генератор списочных выражений или лямбда-функция в соответствующий SQL-запрос, например, такой:

```
DELETE FROM "book" WHERE "book"."year" < 1800;
```

Каскадное удаление

Когда PonyORM удаляет экземпляр сущности, он также должен разорвать его отношения с другими объектами. Отношения между двумя объектами определяются двумя атрибутами отношений. Если другая сторона отношения объявлена как *Set*, то нужно удалить объект из этой коллекции. Если другая сторона объявлена как *Optional*, то нужно установить значение *None*. Если другая сторона объявлена *Required*, то нельзя присвоить *None* этому атрибуту отношения. В этом случае PonyORM попытается выполнить каскадное удаление связанного объекта.

Это поведение по умолчанию можно изменить с помощью опции атрибута *cascade_ delete*. По умолчанию этот параметр имеет значение *True*, если другая сторона отношения объявлена как *Required*, и *False* для всех других типов отношений.

Если связь определена как *Required* на другой стороне отношения и *cascade_delete = False*, то PonyORM вызывает исключение *ConstraintError* при попытке удаления.

Давайте рассмотрим пару примеров.

В приведенном ниже примере возникает исключение *ConstraintError* при попытке удалить должность, связанную с некоторым сотрудником:

```
...
position = Position(name='Директор библиотеки')
employee = Employee(first_name='Лидия', last_name='Иванова',
  patronymic='Семеновна', position=position)

...

position.delete()
```

В следующем примере, если объект *Person* имеет связанный объект *Passport*, то, если Вы попытаетесь удалить объект *Person*, объект *Passport* также будет удален из-за каскадного удаления:

```
class Person(db.Entity):
  ...
  passport = Optional("Passport", cascade_delete=True)

class Passport(db.Entity):
  number = Required(str)
  seriaes = Required(str)
  person = Required("Person")
```

2.10.5 Проблемы сохранения объектов в базе данных

Обычно программисту не стоит беспокоиться о сохранении экземпляров сущностей в базе данных вручную. PonyORM автоматически фиксирует все изменения в базе данных после выхода из контекста *db_session*. Это очень удобно. Но в некоторых случаях может потребоваться вызов функций *flush()* или *commit()* перед выходом из текущего сеанса базы данных.

Если нужно получить значение первичного ключа для вновь созданного объекта, можно вызвать *commit ()* вручную в *db_session()*:

```
  @db_session
  def my_fun():
    author = Author(first_name='Иван', last_name='Иванов',
                    patronymic='Иванович')
    # author.id в данном месте имеет значение None,
    # так как это значение определяется на уровне базы данных
    commit()
    # После вызова commit и создание автора в базе данных
    # author.id уже имеет значение. Выведем его в консоль
    print(author.id)
```

Порядок сохранения объектов

Обычно PonyORM сохраняет объекты в базе данных в том же порядке, в котором они были созданы или изменены. В некоторых случаях PonyORM может изменить порядок операторов INSERT, если это требуется для сохранения объектов. Давайте рассмотрим следующий пример:

```
  from pony.orm import *

  db = Database()

  class TeamMember(db.Entity):
    name = Required(str)
    team = Optional('Team')

  class Team(db.Entity):
    name = Required(str)
```

```
    team_members = Set(TeamMember)

  db.bind('sqlite', ':memory:')
  db.generate_mapping(create_tables=True)
  set_sql_debug(True)

  with db_session:
    john = TeamMember(name='John')
    mary = TeamMember(name='Mary')
    team = Team(name='Tenacity', team_members=[john, mary])
```

В приведенном примере мы определили две сущности: *Команда* (*Team*) и *Член Команды* (*TeamMember*). Далее пытаемся создать двух членов команды, а затем объект команды, определяя ее состав. Схема базы данных будет иметь следующий вид:

```
CREATE TABLE "Team" (
  "id" INTEGER PRIMARY KEY AUTOINCREMENT,
  "name" TEXT NOT NULL
)

CREATE TABLE "TeamMember" (
  "id" INTEGER PRIMARY KEY AUTOINCREMENT,
  "name" TEXT NOT NULL,
  "team" INTEGER REFERENCES "Team" ("id")
)
```

Когда PonyORM создает объекты *john*, *mary* и *team*, он понимает, что ему следует переупорядочить операторы *INSERT* и сначала создать экземпляр объекта *Team* в базе данных, так как перед созданием членов команды необходимо знать его первичный ключ. Поэтому порядок операторов *INSERT* будет следующий:

```
INSERT INTO "Team" ("name") VALUES (?)
[u'Tenacity']

INSERT INTO "TeamMember" ("name", "team") VALUES (?, ?)
[u'John', 1]

INSERT INTO "TeamMember" ("name", "team") VALUES (?, ?)
[u'Mary', 1]
```

Циклические цепочки при сохранении объектов

Теперь предположим, что необходимо назначить капитана в команду. Для этого нужно добавить пару атрибутов в наши сущности: *Team.captain* и обратный атрибут *Team-Member.captain_of*:

```
class TeamMember(db.Entity):
  name = Required(str)
  team = Optional('Team')
  captain_of = Optional('Team')

class Team(db.Entity):
  name = Required(str)
  team_members = Set(TeamMember)
  captain = Optional(TeamMember, reverse='captain_of')
```

Далее напишем код для создания экземпляров сущностей с капитаном, назначенным в команду:

```
        with db_session:
        john = TeamMember(name='John')
        mary = TeamMember(name='Mary')
        team = Team(name='Tenacity', team_members=[john, mary], captain=mary)
```

Когда PonyORM пытается исполнить приведенный выше код, возникает следующее исключение:

```
pony.orm.core.CommitException: Cannot save cyclic chain:
TeamMember -> Team -> TeamMember
```

PonyORM видит, что для сохранения объектов *john* и *mary* в базе данных ему нужно знать идентификатор команды и пытается изменить порядок операторов вставки. Но для сохранения объекта команды с назначенным атрибутом *captain* необходимо знать идентификатор объекта *mary*. В этом случае PonyORM не может разрешить эту циклическую цепочку и выдает исключение.

Чтобы сохранить такую циклическую цепочку, нужно помочь PonyORM, добавив вызов функции *flush()*:

```
  with db_session:
    john = TeamMember(name='John')
    mary = TeamMember(name='Mary')
    flush()
    team = Team(name='Tenacity', team_members=[john, mary], captain=mary)
```

В этом случае PonyORM сначала сохранит объекты *john* и *mary* в базе данных, а затем выполнит оператор UPDATE для построения отношений с объектом *team*:

```
INSERT INTO "TeamMember" ("name") VALUES (?) [u'John']

INSERT INTO "TeamMember" ("name") VALUES (?)[u'Mary']

INSERT INTO "Team" ("name", "captain") VALUES (?, ?) [u'Tenacity', 2]
```

```
UPDATE "TeamMember"
SET "team" = ?
WHERE "id" = ?
[1, 2]

UPDATE "TeamMember"
SET "team" = ?
WHERE "id" = ?
[1, 1]
```

2.11 Запросы

PonyORM предоставляет очень удобный способ создания запросов к базе данных, используя синтаксис выражения генератора. Программисты работают с объектами, которые хранятся в базе данных, как если бы они хранились в памяти, используя собственный синтаксис Python.

Для написания запросов Вы можете использовать выражения генератора Python или лямбда-функции.

2.11.1 Использование генераторов выражений

PonyORM позволяет использовать генераторы выражений как естественный способ написания запросов к базе данных. Есть функция *select ()*, которая принимает генератор Python, переводит его в SQL-код и возвращает объекты из базы данных.

Вот пример запроса:

```
query = select(b for b in Book if len(b.event_logs) > 10)
```

Данный пример возвращает книги, которые выдавали более 10 раз.

К полученному запросу можно применить *filter*:

```
query1 = query.filter(lambda book: book.year > 2000)
```

Будет сформирован новый запрос *query1* на основе *query*, который будет возвращать запрос, фильтрующий книги по году издания.

Также вы можете сделать новый запрос на основе другого запроса:

```
query2 = select(book.genres for book in query1 if 'мир' in book.name)
```

Запрос *query2* будет возвращать жанры книг, удовлетворяющие условиям:

1) выдавались более 10 раз;
2) были изданы позже 2000 года;

3) в названии фигурирует слово «мир».

Приведем SQL-код, в который будет транслирован запрос *query2*:

```
SELECT DISTINCT "genre"."id", "genre"."name"
FROM (
  SELECT "b"."id" AS "book-id"
  FROM "Book" "b"
  LEFT JOIN "EventLog" "eventlog"
  ON "b"."id" = "eventlog"."book"
  WHERE "b"."year" < 2000
  GROUP BY "b"."id"
  HAVING COUNT(DISTINCT "eventlog"."id") > 10
) "t-1", "Book" "book", "Book_Genre" "t-2", "Genre" "genre"
WHERE "book"."name" LIKE '%мир%'
AND "t-1"."book-id" = "book"."id"
AND "book"."id" = "t-2"."book"
AND "t-2"."genre" = "genre"."id"
```

Функция *select ()* возвращает экземпляр класса *Query*, и затем Вы можете вызвать методы объекта *Query* для получения результата, например, получить только первое значение:

```
        first_genre = query2.first()
```

2.11.2 Использование лямбда-функций

Вместо того, чтобы использовать генератор, можно писать запросы, используя лямбда-функцию:

```
  query = Book.select(lambda b: len(b.event_logs) > 10)
```

С точки зрения перевода запроса в SQL нет разницы, используете ли Вы генератор или лямбда-функцию. Единственное отличие состоит в том, что используя лямбда-функцию, Вы можете возвращать только экземпляры сущностей – нет способа вернуть список определенных атрибутов сущности или список кортежей.

2.11.3 Примеры запросов

Рассмотрим типовые запросы в рамках нашей предметной области «Абонентский отдел библиотеки».

```
# Получить всех читателей с фамилией 'Иванов'
Reader.select(lambda r: r.last_name == 'Иванов')
```

```python
# Возвращает кортежи, элементы которых являются должность и количество
# сотрудников, работающих в этой должности
select((e.position.name, count(e)) for e in Employee)
```

```python
# Год издания самой раритетной книги
select(b.year for b in Book).min()
```

```python
# Год издания самой раритетной книги в жанре "Роман"
min(b.year for b in Book
          for g in b.genres if g.name == 'Роман')
```

Стоит заметить, что данный запрос будет транслирован в SQL-код, в котором не будет использован JOIN. Вот полученный SQL-код:

```sql
SELECT MIN("b"."year")
FROM "Book" "b", "Book_Genre" "t-1", "Genre" "g"
WHERE "g"."name" = 'Роман'
AND "b"."id" = "t-1"."book"
AND "t-1"."genre" = "g"."id"
```

```python
  # Три самые старые книги
  Book.select().order_by(desc(Book.year))[:3]
```

```python
  # Читатели, которые не брали книг
  Reader.select(lambda r: len(r.event_logs) == 0)
```

```python
  # Наиболее популярная книга
  Book.select().order_by(lambda b: desc(len(b.event_logs))).first()
```

```python
  # Авторы, которые не написали ни одной книги
  Author.select(lambda a: not a.books)
```

```python
  # Книги, написанные в нескольких жанрах
  Book.select(lambda b: count(b.genres) > 1)
```

```python
  # Три самых трудолюбивых сорудника
  Employee.select().order_by(lambda e: desc(count(e.event_logs)))[:3]
```

```python
  # Читатели, которые заказали несколько разных книг в жанре "Повесть"
  select(r for r in Reader for e in r.event_logs
    if e.action.name == 'Выдача' and 'Повесть' in e.book.genres.name and
        count(e) > 1)
```

2.11.4 Использование даты и времени в запросах

Вы можете выполнять арифметические операции с датой и временем в запросах.

Если выражение может быть вычислено в Python, PonyORM передаст результат вычисления в качестве параметра в запрос:

```
select(e for e in EventLog if e.created_at >= datetime.now() -
   timedelta(days=3))[:]
```

Этот запрос вернет все записи в журнале выдачи/возврата книг, которые были внесены за последние 3 дня, и преобразуется он в следующий SQL-запрос:

```
SELECT "e"."id", "e"."book", "e"."reader",
   "e"."employee", "e"."action", "e"."created_at"
FROM "EventLog" "e"
WHERE "e"."created_at" >= ?
['2020-08-09 10:13:28.745379']
```

Сгенерированный SQL будет варьироваться в зависимости от базы данных.

С PonyORM Вы можете использовать атрибуты *datetime*, такие как месяц, час и т. д. В зависимости от базы данных он будет переведен в свой SQL-код, который извлекает значение этого атрибута. В следующем примере мы получаем все записи в журнале, которые были внесены в декабре:

```
select(e for e in EventLog if e.created_at.month == 12)
```

Это результат трансляции этого запроса для SQLite-код:

```
SELECT "e"."id", "e"."created_at"
FROM "EventLog" "e"
WHERE cast(substr("e"."created_at", 6, 2) as integer) = 12
```

2.11.5 Дубликаты

PonyORM пытается избежать дубликатов в результате запроса, автоматически добавляя ключевое слово *DISTINCT* там, где это необходимо, так как полезные запросы с дубликатами очень редки. Когда кто-то хочет получить объекты с определенными критериями, он обычно не ожидает, что один и тот же объект будет возвращен более одного раза.

Давайте рассмотрим несколько примеров:

Использование критериев

```
Person.select(lambda p: p.id == 1)
```

В этом примере запрос не возвращает дубликаты, потому что результат содержит столбец первичного ключа *Person*. Поскольку дубликаты здесь невозможны, ключевое слово *DISTINCT* не нужно и PonyORM его не добавляет:

```
SELECT "p"."id", "p"."first_name", "p"."last_name",
 "p"."patronymic", "p"."classtype", "p"."position"
FROM "Person" "p"
WHERE "p"."classtype" IN ('Employee', 'Author', 'Reader', 'Person')
AND "p"."id" = 1
```

Получение атрибутов объекта

Рассмотрим пример:

```
select(p.first_name for p in Person)
```

Результат этого запроса возвращает не объекты, а его атрибут. Этот результат запроса может содержать дубликаты, поэтому PonyORM добавит *DISTINCT* к этому запросу:

```
SELECT DISTINCT "p"."first_name"
FROM "Person" "p"
WHERE "p"."classtype" IN ('Author', 'Employee', 'Reader', 'Person')
```

Результат такого запроса обычно используется для выпадающего списка, где дубликаты не ожидаются. Трудно придумать реальный вариант использования, когда Вы хотите иметь здесь дубликаты.

Если Вам нужно сосчитать людей с одинаковыми именами, лучше использовать агрегированный запрос:

```
select((p.first_name, count(p)) for p in Person)
```

Но если необходимо получить имена всех людей, включая дубликаты, Вы можете сделать это с помощью метода *Query.without_distinct()*:

```
select(p.name for p in Person).without_distinct()
```

Получение объектов, используя соединения таблиц

Рассмотрим пример:

```
select(b for b in Book for g in b.genres if g.name in ("Роман", "Повесть"))
```

Этот запрос может содержать дубликаты, поэтому PonyORM устраняет их, используя *DISTINCT*:

```
SELECT DISTINCT "b"."id", "b"."name", "b"."year", "b"."number"
FROM "Book" "b", "Book_Genre" "t-1", "Genre" "g"
WHERE "g"."name" IN ('Роман', 'Повесть')
AND "b"."id" = "t-1"."book"
AND "t-1"."genre" = "g"."id"
```

Без использования DISTINCT возможны дубликаты, поскольку в запросе используются три таблицы (*Book*, *Genre* и *Book_ Genre*), но в разделе *SELECT* фигурирует только одна таблица. Приведенный выше запрос возвращает только книги, и поэтому, как правило, нежелательно получать одну и ту же книгу более одного раза.

Но если по какой-то причине Вам не нужно исключать дубликаты, Вы всегда можете добавить к запросу *without_ distinct()*.

Пользователь, вероятно, хотел бы видеть дубликаты объектов *Book*, если результат запроса содержит и жанры, в которых написана книга. В этом случае запрос PonyORM будет другим:

```
select((b, g) for b in Book for g in b.genres if g.name in
   ("Роман", "Повесть"))
```

И в этом случае PonyORM будет добавлять ключевое слово *DISTINCT* в SQL-запрос.

```
SELECT DISTINCT "b"."id", "g"."id"
FROM "Book" "b", "Book_Genre" "t-1", "Genre" "g"
WHERE "g"."name" IN ('Роман', 'Повесть')
AND "b"."id" = "t-1"."book"
AND "t-1"."genre" = "g"."id"
```

Подведем итоги.

1 Принцип «все запросы не возвращают дубликаты по умолчанию» прост для понимания и не приводит к неожиданностям. Такое поведение – это то, что пользователи хотят в большинстве случаев.

2 PonyORM не добавляет *DISTINCT*, если в запросе не должно быть дубликатов.

3 Метод запроса *without_ distinct ()* может быть использован для того, чтобы PonyORM не удалял дубликаты.

2.11.6 Использование сырых SQL-запросов

PonyORM позволяет использовать «сырой» SQL в Ваших запросах. Есть два варианта, как Вы можете его использовать.

1 Использование функции *raw_ sql ()* для записи только части генератора или лямбда-запроса с использованием «сырого» SQL.

2 Написание полного SQL-запроса, используя методы *Entity.select_by_sql ()* или *Entity.get_by_sql ()*.

Использование функции *raw_sql()*

Рассмотрим примеры использования функции *raw_sql ()*.

Результат *raw_sql()* можно рассматривать как логическое выражение. Вот, например, код, возвращающий книги, изданные позже 2000 года:

```
select(b for b in Book if raw_sql('"b"."year" > 2000'))
```

Результат *raw_sql()* можно использовать в операторах сравнения:

```
q = Book.select(lambda x: raw_sql('abs("x"."year") - 1900') > 25)
print(q.get_sql())
```

```
SELECT "x"."id", "x"."name", "x"."year", "x"."number"
FROM "Book" "x"
WHERE abs("x"."year") - 1900 > 25
```

Кроме того, в приведенном выше примере мы используем *raw_sql ()* в лямбда-функции и распечатываем полученный SQL. Как видите, «сырая» часть SQL становится частью всего запроса.

raw_sql () может принимать параметры, например:

```
x = 2000
select(b for b in Book if raw_sql('"b"."year" > $x'))
```

Вы можете динамически изменять содержимое функции *raw_sql ()* и по-прежнему использовать параметры внутри:

```
x = 2000
s = 'b.year > $x'
select(b for b in Book if raw_sql(s))
```

Также PonyORM предоставляет возможность использовать различные типы внутри запроса SQL:

```
x = datetime.now() - timedelta(days=30)
select(e for e in EventLog if raw_sql('e.created_at > $x'))
```

Параметры внутри «сырой» части SQL можно объединить:

```
x = 100
y = 1900
select(b for b in Book if raw_sql('b.year > $(x + y)'))
```

Вы даже можете вызывать функции Python:

```
select(e for e in EventLog if raw_sql('e.created_at < $date.today()'))
```

В итоге будут возвращены все записи журнала, созданные раньше текущего дня.

Функция *raw_sql ()* может использоваться не только в блоке условия, но также в блоке возврата результата запроса:

```
first_names = select(raw_sql('UPPER(p.first_name)') for p in Person)[:]
print(first_names)
```

В данном случае возвращаются все имена людей в верхнем регистре.

Но когда Вы возвращаете данные с помощью функции *raw_sql ()*, Вам может потребоваться указать тип результата, потому что PonyORM не всегда имеет представление о нем. Например,

```
dates = select(raw_sql('(e.created_at)') for e in EventLog)[:]
print(dates)

['2020-08-20 05:09:27.629535', '2020-08-21 15:08:17.736292']
```

В этом примере будут возвращены даты и времена в текстовом виде. Если Вы хотите получить результат в виде списка дата-время, Вам нужно указать *result_type*:

```
dates = select(raw_sql('(e.created_at)', result_type=datetime)
  for e in EventLog)[:]
print(dates)

[datetime.datetime(2020, 8, 20, 5, 9, 27, 629535),
 datetime.datetime(2020, 8, 21, 15, 8, 17, 736292)]
```

Функцию *raw_sql ()* можно использовать и в *Query.filter ()*:

```
x = 2000
select(b for b in Book).filter(lambda b: b.year > raw_sql('$x'))
```

В этом примере будут возвращены книги, изданные позже 2000 года.

Данную функцию можно использовать и внутри *Query.filter ()* без лямбда-функции. В этом случае Вы должны использовать первую букву имени объекта в нижнем регистре в качестве псевдонима. Перепишем предыдущий пример:

```
x = 2000
Book.select().filter(raw_sql('b.year > $x'))
```

Можно использовать несколько выражений *raw_sql ()* в одном запросе, например, для получения всех людей с именем «Иван» или фамилией «Иванов»:

```
x = 'ИВАНОВ'
y = 'ИВАН'
Person.select(lambda p: raw_sql("UPPER(p.first_name)") == y
              or raw_sql("UPPER(p.last_name)") == x)
```

Можно использовать *raw_sql ()* в разделе *Query.order_by ()*:

```
x = 3
Person.select().order_by(lambda p: raw_sql('SUBSTR(p.first_name, $x)'))
```

Этот запрос вернет всех людей, отсортированных в алфавитном порядке по первым трем символам имени.

2.11.7 Использование select_by_sql() и get_by_sql()

Хотя PonyORM может транслировать практически любое условие, написанное на Python, в SQL, иногда возникает необходимость использовать, например, «сырой» SQL для вызова хранимой процедуры или использования функции диалекта конкретной системы базы данных. В этом случае PonyORM позволяет пользователю написать запрос на «сыром» SQL, поместив его в функцию *Entity.select_by_sql ()* или *Entity.get_by_sql ()* в виде строки:

```
Book.select_by_sql("SELECT * FROM Book")
```

В отличие от метода *Entity.select()*, метод *Entity.select_by_sql ()* возвращает не объект *Query*, а список экземпляров объекта.

Параметры передаются с использованием следующего синтаксиса: «$имя переменной» или «$(выражение в Python)». Например:

```
x = 1900
y = 2000
Book.select_by_sql("SELECT * FROM Book WHERE year > $x AND year < $(y - 50)")
```

Когда PonyORM встречает параметр в «сыром» запросе SQL, он получает значение переменной из текущей области видимости (из глобальных и локальных) или из словарей, которые могут быть переданы в качестве параметров:

```
Book.select_by_sql("SELECT * FROM Book WHERE year > $x AND year < $(y - 50)",
                   globals={'x': 1900}, locals={'y': 2000})
```

Переменные и более сложные выражения, указанные после знака $, будут автоматически вычислены и перенесены в запрос в качестве параметров, что делает невозможным SQL-инъекции. PonyORM автоматически заменяет $x в строке запроса на «?».

Если Вам нужно использовать знак $ в запросе (например, в имени системной таблицы), Вы должны написать два знака $ подряд: $$.

2.12 Работа с отношениями между сущностями

2.12.1 Отношение «один-к-одному»

Давайте посмотрим, как установить связь типа «один-к-одному», например, между человеком и паспортом. Создадим объекты этих сущностей:

```
person = Person(first_name='Иванов', last_name='Иван', patronymic='Иванович')
passport = Passport(series='1234', number='123456', person=person)
```

При создании человека мы можем не указывать паспорт, так как соответствующий атрибут имеет тип *Optional*. Однако при создании паспорта необходимо указать, кому он принадлежит (*person* – атрибут типа *Required*).

Как только мы создали паспорт и указали, чей он, так сразу у объекта *person* атрибут *passport* будет ссылаться на созданный экземпляр класса *Passport*. Проверим это утверждение:

```
print(person.passport)
 -> Passport[1]
```

Пусть наш Иванов Иван Иванович достиг возраста, при котором необходимо заменить паспорт на новый:

```
  del passport
  new_passport = Passport(series="4321", number='987654', person)
```

Мы сначала удалили старый паспорт, а затем создали новый, сказав, кому он будет принадлежать. Проверим, не остался ли наш человек без паспорта:

```
  print(person.passport)
  -> Passport[2]
```

2.12.2 Отношение «один-ко-многим»

Между сущностями *Должность* и *Сотрудник* мы определили ассоциацию «Работает в». При этом любой сотрудник работает только в одной должности, а в одной должности могут работать несколько сотрудников. Таким образом, это отношение «один-ко-многим». Приступим к созданию объектов. Уточним, что атрибут *position* класса *Сотрудник* имеет тип

Required, а значит при создании объекта класса *Сотрудник* необходимо указать его должность:

```
first_position = Position('Библиотекарь')
first_employee = Employee(last_name='Сидорова', first_name='Людмила',
                          patronymic='Сергеевна', position=first_position)
```

Мы создали два объекта и установили между ними связь путем указания должности у сотрудника. Проверим наличие связи с обеих сторон:

```
print(first_position.employees)
-> EmployeeSet([Employee[1]])

print(first_employee.position)
-> Position[1]
```

Можем еще создать сотрудника, который работает в той же должности, и повысить в должности первого сотрудника:

```
second_employee = Employee(last_name='Бирюкова', first_name='Светлана',
                           patronymic='Леонидовна', position=first_position)

print(first_position.employees)
-> EmployeeSet([Employee[1], Employee[2]])

second_position = Position('Главный библиотекарь')
first_employee.position = second_position

print(first_employee.position)
-> Position[2]

print(first_position.employees)
-> EmployeeSet(Employee[2]])

print(second_position.employees)
-> EmployeeSet(Employee[1]])
```

Из примера видим, что изменяя атрибут *position* класса *Employee*, значения атрибута *employees* класса *Position* автоматически синхронизируются.

2.12.3 Отношение «многие-ко-многим»

Давайте создадим объекты нескольких сущностей, например:

```
book = Book(name="Война и мир", year=1988, number=10)
author = Author(first_name='Лев', last_name='Толстой',
                patronymic='Николаевич')
```

Сразу после того, как мы создали экземпляры *book* и *author*, они не имеют установленных отношений. Давайте проверим значения атрибутов отношений:

```
print(book.authors)
-> AuthorSet([])

print(author.books)
-> BookSet([])
```

Теперь давайте установим связь между этими двумя объектами:

```
book.authors.append(author)

print(book.authors)
-> BookSet([Book[1]])

print(author.books)
-> AuthorSet([Author[1]])
```

Мы видим, что как только мы добавили книге автора, так сразу у данного автора появилась книга в его коллекции. Аналогично для установления связи мы могли добавить автору книгу:

```
author.books.append(book)

print(author.books)
-> AuthorSet([Author[1]])

print(book.authors)
-> BookSet([Book[1]])
```

2.12.4 Операции с коллекциями

Атрибут *Book.authors* представлен в виде списка, и поэтому мы можем использовать обычные операции, применимые к спискам: *add()*, *remove()*, *in*, *len()*, *clear()*.

Вы можете добавлять или удалять отношения между объектами, используя методы *Set.add()* и *Set.remove()*:

```
book.authors.remove(author)
print(book.authors)
-> AuthorSet([])

book.author.add(author)
print(book.authors)
-> AuthorSet([Author[1]])
```

Вы можете проверить, написал ли автор книгу:

```
author in book.authors
-> True
```

Или убедитесь, что данный человек не является автором книги:

```
 author not in p1.cars
 -> False
```

Определить количество авторов:

```
 len(book.authors)
 -> 1
```

Если Вам нужно создать экземпляр книги и сразу назначить ему автора (автор до этого момента не существовал), есть несколько способов сделать это. Один из вариантов – вызвать метод *create()* атрибута *collection*:

```
 book.authors.create(first_name='Александр', last_name='Пушкин',
                     patronymic='Сергеевич')
 commit()
```

Теперь мы можем проверить, что новый экземпляр *Author* был добавлен в атрибут коллекции *Book.authors*:

```
 print(book.authors)
 -> AuthorSet([Author[2], Author[1]])
 book.authors.count()
 -> 2
```

Вы можете перебрать элементы списка:

```
for a in book.authors:
 print(a.full_name, end='', sep='; ')
 -> 'Пушкин Александр Сергеевич'; 'Толстой Лев Николаевич'
```

2.12.5 Параметры атрибута коллекции

Вот список параметров, которые Вы можете применить к атрибутам коллекции:

• **cascade_delete**. Управляет каскадным удалением связанных объектов. Если выставлено значение *True*, то это означает, что PonyORM всегда делает каскадное удаление, даже если другая сторона определена как *Optional*. Если выставлено значение *False*, то PonyORM никогда не делает каскадное удаление для этих отношений. Если связь определена как *Required* на другом конце и *cascade_ delete* = *False*, то PonyORM вызывает исключение *ConstraintError* при попытке удаления;

• **columns**. Задает имена столбцов в таблице базы данных, которые используются для сопоставления составного атрибута;

• **lazy**. Когда данный атрибут принимает значение *True*, то PonyORM откладывает загрузку значения атрибута при загрузке объекта. Значение не будет загружено, пока Вы не попытаетесь получить доступ к этому атрибуту напрямую. По умолчанию для *lazy* установлено значение *True* для *LongStr* и *LongUnicode* и *False* для всех других типов;

• **reverse**. Указывает имя атрибута на другом конце, которое должно использоваться для связи. Это необходимо, если между двумя объектами существует более одного отношения;

• **reverse_columns**. Используется для симметричных отношений, если у объекта есть составной первичный ключ. Позволяет указать имя столбца базы данных для промежуточной таблицы;

• **table**. Используется только для связи «многие-ко-многим», чтобы указать имя промежуточной таблицы.

2.12.6 Запросы атрибутов коллекции и другие методы

Для получения данных из коллекции отношений можно использовать следующие методы:

• **select()**. Выбирает объекты из коллекции по заданному критерию в виде lambda-функции;

• **random(limit)**. Возвращает заданное количество случайных объектов из коллекции;

• **page (pagenum, pagesize=10)**. Возвращает страницу с номером *pagenum*. Количество объектов в странице определяется параметром *pagesize*. Часто используется в пагинации;

• **order_by(attr|lambda)**. Возвращает отсортированную коллекцию. Способ упорядочивания определяется либо атрибутом, либо lambda-функцией;

• **load**. Загружает все связанные объекты из базы данных;

• **filter()**. Аналог *select()*.

Приведем несколько примеров.

В данном примере выбираются все книги автора с идентификатором 101, написанные с 1867 по 1890 гг.

```
author = Author[101]
author.books.filter(lambda book:  book.year >= 1867 and book.year <= 1890)[:]
```

Следующий запрос может использоваться для отображения второй страницы списка книг автора с идентификатором 101, упорядоченного по атрибуту *name*:

```
author.books.order_by(Book.name).page(2, pagesize=3)
```

Этот же запрос может быть записан в следующей форме:

```
author.books.order_by(lambda b: b.name).limit(3, offset=3)
```

Следующий запрос возвращает две случайные книги автора:

```
 author.books.random(2)
```

2.13 Агрегатные функции

Вы можете использовать следующие шесть агрегатных функций в своих запросах:

- *sum()*;
- *count()*;
- *min()*;
- *max()*;
- *avg()*;
- *group_ concat()*.

Давайте рассмотрим несколько примеров простых запросов с использованием этих функций.

1 Количество всех экземпляров книг в жанре «Роман»:

```
sum(b.number for b in Book if 'Роман' in b.genres.name)
```

2 Количество читателей с именем «Иван»:

```
count(r for r in Reader if r.first_name = 'Иван')
```

3 Самая первая изданная книга Л. Н. Толстого:

```
author = Author[1]
min(b.year for b in Book if author in b.authors)
```

4 Дата и время последней выдачи книги:

```
max(e.created_at for e in EventLog if e.action.name == 'Выдача')
```

5 Среднее количество экземпляров книг, изданных в период с 1950 по 2000 года:

```
avg(b.number for b in Book if b.year >= 1950
    and b.year <= 2000)
```

6 Список фамилий всех сотрудников, разделенных запятой:

```
group_concat(e.last_name for e in Employee)
```

Агрегатные функции также могут использоваться внутри запроса, например, если Вам нужно найти самую редкую книгу. Для этого можно написать следующий запрос:

```
select(b for b in Book if b.number == min(b.number for b in Book))
```

Или, например, чтобы получить все книги, написанные в четырех жанрах:

```
select(b for b in Book if count(b.genres) == 4)
```

2.13.1 Использование агрегатных функций через объект запроса

Вы можете вызвать аналогичный список агрегатных функций в виде методов класса *Query*. Например, следующие два выражения эквивалентны:

```
select(count(b) for b in Book if b.year == 1950)
```

```
select(b for b in Book if b.year == 1950).count()
```

Вот список агрегатных функций в классе Query:
1) *Query.avg()*;
2) *Query.count()*;
3) *Query.min()*;
4) *Query.max()*;
5) *Query.sum()*;
6) *Quety.group_concat()*.

2.13.2 Несколько агрегатных функций в одном запросе

SQL позволяет включать несколько агрегатных функций в один запрос. Например, нам может потребоваться получить как самую старую, так и самую молодую книгу, которые хранятся в библиотеке. При этом необходимо вывести эти книги для каждого жанра. В SQL такой запрос будет выглядеть так:

```
SELECT "g"."name", MIN("b"."year"), MAX("b"."year")
FROM "Genre" "g" INNER JOIN "Book_Genre" "t-1" ON
  "g"."id" = "t-1"."genre" INNER JOIN "Book" "b" ON
  "b"."id" = "t-1"."book"
GROUP BY "g"."name"
```

С PonyORM Вы можете использовать тот же подход:

```
select((g.name, min(b.year), max(b.year)) for g in Genre for b in g.books)
```

2.13.3 Функция count

Агрегатные запросы часто должны вычислять количество чего-либо. Вот как мы получаем количество книг, написанных автором с фамилией «Толстой»:

```
    count(e for e in Employee if 'Толстой' in b.authors.last_name)
```

Число сотрудников для каждой должности:

```
    select((p, count(p.employees)) for p in Position
           if p.name=='Библиотекарь')
```

или

```
select((e.position, count(e)) for e in Employee
       if e.position.name == 'Библиотекарь')
```

В первом примере агрегатная функция *count()* получает коллекцию, и PonyORM преобразует ее в подзапрос. Этот подзапрос будет оптимизирован PonyORM с использованием соединения таблиц LEFT JOIN. Во втором примере функция *count ()* получает один объект вместо коллекции. В этом случае PonyORM добавит раздел GROUP BY к запросу SQL, и группировка будет выполнена по атрибуту *e.position*.

Если Вы используете функцию *count ()* без аргументов, она будет переведена в SQL COUNT (*). Если вы укажете аргумент, он будет переведен в COUNT(DISTINCT *аргумент*).

2.13.4 Условный count

Есть еще один способ использования функции *count()*. Давайте предположим, что требуется получить три числа для каждой должности:

1) количество сотрудников с именем «Лидия»;
2) количество сотрудников с именем «Галина»;

3) количество сотрудников с именем «Ирина».

Запрос может быть построен следующим образом:

```
select((p, count(p for p in p.employees if p.first_name == 'Лидия'),
  count(p for p in p.employees if p.first_name == 'Галина'),
  count(p for p in p.employees if p.first_name > 'Ирина'))
  for p in Position)
```

Приведем SQL-код данного запроса:

```
SELECT "p"."id", (
SELECT COUNT(*)
FROM "Person" "p-2"
WHERE "p"."id" = "p-2"."position"
AND "p-2"."first_name" = 'Лидия'
), (
SELECT COUNT(*)
FROM "Person" "p-2"
WHERE "p"."id" = "p-2"."position"
AND "p-2"."first_name" = 'Галина'
), (
SELECT COUNT(*)
FROM "Person" "p-2"
WHERE "p"."id" = "p-2"."position"
AND "p-2"."first_name" > 'Ирина'
)
FROM "Position" "p"
```

Хотя этот запрос будет работать, он довольно длинный и не очень эффективный. Каждый *count* будет переведен в отдельный подзапрос. Для повышения эффективности PonyORM предоставляет синтаксис условного COUNT. Предыдущий запрос перепишем с использованием условного COUNT следующим образом:

```
select((e.position, count(e.first_name == 'Лидия'),
 count(e.first_name == 'Галина'),
 count(e.first_name  == 'Ирина')) for e in Employee)
```

Полученный SQL-код:

```
SELECT "e"."position", COUNT(case when "e"."first_name" = 'Лидия'
   then 1 else null end),
  COUNT(case when "e"."first_name" = 'Галина' then 1 else null end),
  COUNT(case when "e"."first_name" = 'Ирина' then 1 else null end)
FROM "Person" "e"
WHERE "e"."classtype" IN ('Employee')
GROUP BY "e"."position"
```

Таким образом, мы помещаем наше условие в функцию *count()*. Этот запрос не будет иметь подзапросов, что делает его более эффективным.

2.13.5 Запросы с HAVING

Оператор SELECT имеет два раздела, которые используются для условий: WHERE и HAVING. Раздел WHERE используется чаще и содержит условия, которые будут применяться к каждой строке. Если запрос содержит агрегатные функции, такие как MAX или SUM, оператор SELECT также может содержать разделы GROUP BY и HAVING. Условия раздела HAVING применяются после группировки результатов SQL-запроса. Обычно условия раздела HAVING всегда содержат агрегатные функции, тогда как условия в разделе WHERE могут содержать только агрегатные функции внутри подзапроса.

Когда Вы формируете запрос, который содержит агрегатные функции, PonyORM необходимо определить, будет ли полученный SQL содержать разделы GROUP BY и HAVING, и куда он должен поместить каждое условие из запроса Python. Если условие содержит статистическую функцию, PonyORM помещает условие в раздел HAVING, в противном случае он помещает условие в раздел WHERE.

Рассмотрим следующий запрос, который возвращает кортежи:

```
select((e.position, count(e)) for e in Employee
  if e.position.name == 'Главный библиотекарь'
    and count(e) < 5)
```

В этом запросе у нас есть два условия. Первым условием является *e.position == 'Главный библиотекарь'*. Поскольку в него не включена агрегатная функция, PonyORM поместит это условие в раздел WHERE. Второе условие *co-unt(e) < 5* содержит агрегатную функцию *count* и будет помещено в раздел HAVING.

Другой вопрос, какие столбцы PonyORM следует добавить в раздел GROUP BY. В соответствии со стандартом SQL любой неагрегированный столбец, помещенный в оператор SELECT, также должен быть добавлен в раздел GROUP BY. Давайте рассмотрим следующий запрос:

```
SELECT A, B, C, SUM(D), MAX(E), COUNT(F)
FROM T1
WHERE ...
GROUP BY ...
HAVING ...
```

В соответствии со стандартом SQL нам необходимо включить столбцы *A*, *B* и *C* в раздел GROUP BY, поскольку эти столбцы появляются в списке SELECT и не заключаются в какие-либо агрегатные функции. Если агрегированный запрос возвращает кортеж с несколькими выражениями, любое неагрегированное выражение будет помещено в раздел GROUP BY. Давайте снова рассмотрим тот же запрос PonyORM:

```
  select((e.position, count(e)) for e in Employee
    if e.position.name == 'Главный библиотекарь'
      and count(e) < 5)
```

Далее приведем SQL-запрос, соответствующий запросу на языке Python:

```
SELECT "e"."position", COUNT(DISTINCT "e"."id")
FROM "Person" "e", "Position" "position"
WHERE "e"."classtype" IN ('Employee')
AND "position"."name" = 'Главный библиотекарь'
AND "e"."position" = "position"."id"
GROUP BY "e"."position"
HAVING COUNT(DISTINCT "e"."id") < 5
```

Этот запрос возвращает кортежи (Должность, Количество сотрудников). Первый элемент кортежа не агрегируется, поэтому он будет помещен в раздел GROUP BY. Второй аргумент встречается в функции *count()*, поэтому условие $count(e) < 5$ было помещено в раздел HAVING запроса.

Иногда условие, которое следует поместить в раздел HAVING, содержит несколько неагрегированных столбцов. Такие столбцы будут добавлены в раздел GROUP BY, поскольку в соответствии со стандартом SQL запрещено использовать неагрегированный столбец внутри раздела HAVING, если он не был добавлен в список GROUP BY. Например:

```
    select((book.name, book.number, min(book.year))
      for book in Book if book.year > 500 + min(book.year))
```

Этот запрос имеет следующее условие: $book.year > 500 + min(book.year)$, который содержит агрегатную функцию и должен быть добавлен в раздел HAVING. Но *book.number* не агрегируется. Следовательно, он будет добавлен в раздел GROUP BY для удовлетворения требований SQL.

2.13.6 ORDER_BY

Агрегатные функции можно использовать внутри функции *Query.order_by ()*. Вот пример:

```
    select((e.position, count(e.event_logs)) for e in Employee)
      .order_by(lambda _, e: desc(e))[:]
```

Другой способ упорядочения по агрегированному значению – указание номера позиции в методе *Query.order_by ()*:

```
select((e.position, count(e.event_logs)) for e in Employee)
  .order_by(-2)[:]
```

Минус означает, что упорядочивание будет по убыванию, а 2 означает, что сортировка будет проводиться по второму аргументу в блоке SELECT.

2.14 Проектный практикум по PonyORM

2.14.1 Основные положения

Основной целью выполнения практических занятий является получение практических навыков работы с PonyORM путем выполнения заданий, связанных с написанием программного кода на языке Python, используя функционал PonyORM.

Практические задания выполняются в рамках предоставляемого Python-проекта «Абонентский отдел библиотеки». Исходный код проекта содержит описание классов модели предметной области.

Перед тем, как начать выполнение практических заданий, рекомендуется ознакомиться с теоретическим материалом учебного пособия.

Защита практических заданий осуществляется в форме демонстрации исходных текстов проекта на языке Python и результатов их исполнения.

2.14.2 Практические задания

Модификация модели предметной области

1 Добавить следующие свойства книги: вид издания, библиотечный шифр ББК.

2 Реализовать хранение многотомных изданий книг.

3 Реализовать хранение, выдачу и возврат периодических изданий.

Работа с экземплярами сущностей

1 Напишите запросы выборки, вставки, обновления, удаления для многотомных и периодических изданий, журнала выдачи/возврата периодических изданий.

2 Оценить эффективность операций выборки, вставки, обновления и удаления экземпляров сущностей средствами PonyORM. Для этого замерить скорость выполнения указанных операций, используя низкоуровневые

библиотеки работы с выбранной СУБД и средствами PonyORM. Оценить, во сколько производительность PonyORM ниже производительности низкоуровневых библиотек. Выполнить эксперименты для разных объемов данных. Для операции поиска используйте различные по сложности условия выборки, а для операции удаления поэкспериментируйте с каскадным удалением и без него.

3 Придумайте пример и реализуйте его в коде (возможно потребуется модификация модели предметной области), при котором PonyORM изменяет порядок сохранения связанных друг с другом объектов.

Простые запросы

1 Реализуйте следующие запросы:

- поиск книг издания, введенного пользователем;
- поиск книг по заданному пользователем шифру;
- поиск многотомных книг;
- поиск книг, состоящих из двух томов;
- поиск многотомных книг, вторые тома которых никому не выдавались;
- поиск всех томов книг, которые были выданы указанным пользователем библиотекарем, но еще не были возвращены.

2 Запрограммируйте вышеуказанные запросы данного пункта, используя в том числе сырые SQL-запросы и методы *select_by_sql* и *get_by_sql*.

Работа с отношениями

1 Проведите исследование качества генерируемого PonyORM сложного SQL-запроса по выборке данных с несколькими соединениями таблиц. Сделайте выводы.

2 Придумайте и реализуйте пример (возможно потребуется модификация модели предметной области), в котором используется самоссылаемость симметричного и асимметричного типа. Например, организовать справочник ББК в иерархическом виде.

3 Продемонстрируйте синхронизацию объектов, связанных отношением. Между объектами классовых сущностей создайте отношение с разных сторон. Со стороны «один-» воспользуйтесь методом *create ()* .

Агрегатные функции

Реализуйте следующие запросы, в которых должны быть использованы агрегатные функции.

1 Получить количество томов по заданному идентификатору книги.

2 Получить дату первого возврата тома заданной пользователем книги.

3 Определить среднее количество книг, выданных в заданный пользователем период времени.

4 Вывести количество книг для каждого вида издания.

5 Получить в алфавитном порядке все жанры, количество книг которых было выдано не более десяти раз.

ЗАКЛЮЧЕНИЕ

Развитие программных средств работы с реляционными базами данных в виде ORM привело к сглаживанию разрыва между различными парадигмами: объектно-ориентированной парадигмы программирования с одной стороны и реляционной парадигмы хранения данных с другой. PonyORM является одним из ярких представителей реализации паттерна ORM для языка Python, позволяющий строить запросы в Python-стиле, используя генераторы выражений и lambda-функции. Это одно из его достоинств, выделяющих его из ряда других ORM, например, Django ORM, SQL Alchemy. Другим достоинством PonyORM является его высокая производительность и качество трансляции запросов в SQL-код. Можно сказать, что PonyORM был проверен временем (первый релиз был датирован 2013 годом) и в настоящее время, как показывает официальный репозиторий на github, продолжает развиваться (коммиты вносятся достаточно регулярно). Однако, как и у всех, PonyORM не обделен недостатками. Можно выделить следующие:

1) отсутствие системы миграции, необходимой в крупных проектах, в которых с завидной регулярностью происходит изменение структуры хранения данных;

2) развитый жизненный цикл объектов сущностей по типу того, как это организовано в Hibernate;

3) к некоторых крупным СУБД, например, MS SQL, отсутствуют адаптеры;

4) нет возможности связывать объекты из разных баз данных или использовать композитные поля.

Будем надеяться, что в будущем PonyORM обзаведется широким кругом почитателей и будет использован в полноценном фреймворке, как это произошло, например, с ActiveRecord и Ruby On Rails.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы / Ф. Брукс, Ч. Хилл ; пер. с англ. – Москва : Символ-Плюс, 2010. – 304 с.

2 Фаулер Мартин. Архитектура корпоративных программных приложений / Мартин Фаулер ; пер. с англ. – Москва : Издательский дом «Вильямс», 2006. – 544 с.

3 Бауэр К. Java Persistence API и Hibernate / К. Бауэр, Г. Кинг , Г. Грегори ; пер. с англ. Д. А. Зинкевича ; под науч. ред. А. Н. Киселева. – Москва : ДМК Пресс, 2017. – 632 с.

4 Александер К. Язык шаблонов. Города. Здания. Строительство / К. Александр, С. Исикава, М. Силверстайн ; пер. с англ. – Москва : Издательство Студии Артемия Лебедева, 2014. – 1096 с.

5 Приемы объектно-ориентированного проектирования. Паттерны проектирования / Дж. Ральф, Х. Ричард, В. Джон, Г. Эрих ; пер. с англ. – Санкт-Петербург : Питер, 2016. – 366 с.

6 PonyORM : официальный сайт. – URL: https://ponyorm.org (дата обращения: 01.12.2020).

7 Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем / Э. Эванс ; пер. с англ. – Москва : «Вильямс», 2011. – 448 с.

Учебное издание

Черепанов Олег Сергеевич

Маер Алексей Владимирович

ВВЕДЕНИЕ В PONYORM

Учебное пособие

Редактор А. С. Темирова

Подписано в печать 04.02.21	Формат 60x84 1/16	Бумага 80 г/м^2
Печать цифровая	Усл. печ. л. 4.88	Уч.-изд. л. 4.88
Заказ № 14	Тираж 100	

Библиотечно-издательский центр КГУ.
640020, г. Курган, ул. Советская, 63/4.
Курганский государственный университет.