Санкт-Петербургский государственный университет

Кафедра системного програмирования

Группа 22.Б15-мм

Перенос и адаптация моделей базы данных Desbordante на Python с использованием SQLAlchemy

Нурмухаметов Рафик

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры информационно-аналитических систем, Чернышев Г.А.

Санкт-Петербург
2023

Оглавление

Введение

В современном мире объем создаваемых и обрабатываемых данных постоянно увеличивается, подчеркивая важность процесса профилирования данных. Профилирование данных — это комплексный процесс, позволяющий извлекать из имеющихся данных метаданные, которые могут включать в себя различные детали, такие как размер файлов, время создания, авторство, а также раскрывать не очевидные, скрытые в глубине данных закономерности.

Различают классическое и наукоемкое профилирование. Классическое профилирование направлено на выявление простых статистических характеристик, например, нахождение минимального и максимального значений для каждого столбца таблицы, нахождение суммы всех его ячеек и других показателей. С другой стороны, наукоемкое профилирование нацелено на обнаружение сокрытых в данных закономерностей, которые могут быть описаны различными примитивами, такими как функциональные зависимости [13], ассоциативные правила [1] и условные функциональные зависимости [12]. Примитив, как понятие, представляет собой некоторое правило, работающее с данными и формулируемое с применением математических методов.

Процесс профилирования данных имеет большое значение в анализе данных, обнаружении в них ошибок и недочетов, а также управлении информацией в различных типах хранилищ данных. Выявленные при профилировании данных закономерности могут представлять особую ценность, например, для ученых, которые на её основе могут построить некоторую гипотезу; специалистам по машинному обучению, дата-инженерам, а также всем специалистам, работающим с большим массивом данных.

Одним из наиболее продвинутых инструментов, позволяющих проводить качественный анализ и обработку данных, является Desbordante — наукоёмкий профилировщик данных, обладающий высокими показателями производительности и поддерживающий поиск трех примитивов: функциональные зависимости, ассоциативные правила и условные

функциональные зависимости. На момент начала работы Desbordante представлял собой консольное приложение, целиком написанное на C++, и его веб-версию.

Веб-приложение Desbordante использует клиент-серверную архитектуру, где клиентская часть приложения разработана с использованием библиотеки React, а серверная часть реализована на платформе Node.js. Основные функции сервера заключаются в хранении и обработки данных из базы, создании новых задач, предоставлении информации о их текущем состоянии и результатах работы алгоритма. В настоящее время инициирован процесс переписывания серверной части приложения на язык программирования Python, с пересмотром её текущей архитектуры.

Одним из ключевых компонентов серверной части является база данных, хранящая в себе сведения о пользователях, сессиях, загружаемых файлах и конечно же задачах. Переписывание серверной части на Python влечет за собой потребность в пересмотре и переработке моделей базы данных. Текущая архитектура базы данных Desbordante характеризуется избыточностью связей и компонентов, что делает структуру ненужно сложной, тяжеловесной, и затрудняющей решение стандартного спектра задач. В связи с этим, стоит задача не просто адаптировать существующую архитектуру базы данных под новый языковой стек, но и предложить новое, более оптимизированное и эффективное организационное решение.

1. Постановка задачи

Целью работы является пересмотр текущей архитектуры базы данных с её дальнейшим переписыванием на язык Python. Для этого были поставлены следующие задачи:

1. Провести анализ существующего решения;
2. Спроектировать новую архитектуру базы данных, учитывая найденные при анализе недостатки;
3. Переписать модели базы данных на язык Python, с учетом внесенных в архитектуру изменений;
4. Подготовить стартовое тестовое окружение для дальнейшего тестирования кода, взаимодействующего с базой данных.

2. Общие сведения о существующем решении

2.1. Архитектура серверной части

Серверная часть веб-приложения Desbordante подразделяется на несколько компонент: сервис, предоставляющий API для клиента, база данных, оркестратор и исполнитель алгоритма. Веб-сервер разработан на платформе Node.js и выполняет множество критически важных функций, а именно:

1. Управление данными пользователей, что подразумевает хранение информации и разграничивание доступных им прав и функциональных возможностей;

2. Управление процессом создания задач, в том числе инициализация новых, а также аккумулирование и предоставление информации по активным или уже завершенным задачам;

3. Хранение данных о загруженных файлах, задачах и их конфигурациях.

Известно, что веб-серверы не предназначены для проведения сложных математических операций. Такие операции, как выполнение алгоритмов для нахождения закономерностей в данных потребляют много системных ресурсов. Именно поэтому, во втором пункте данного списка под процессом “создания задач” понимается не непосредственное выполнение алгоритма на заданных пользователем конфигурациях и загруженных файлах, а процесс инициализации задачи и последующей её отправки в специализированную очередь. Обработка очереди обеспечивается брокером сообщений Apache Kafka [2] —распределенной системой обмена сообщениями, предназначенной для эффективного и надежного обмена данными между различными компонентами приложения. Таким образом, Apache Kafka выступает некоторым посредником между веб-сервером и оркестратором.

Оркестратор — элемент, реализованный на языке программирования Python — отвечает за дальнейшую работу с задачами: получение их из очереди, запуск Docker-контейнеров для их выполнения, передача идентификаторов задач исполнителю алгоритма. Оркестратор контролирует состояние контейнера и процесс выполнения задачи, и в случае возникновения сбоев в работе алгоритма, способен обновлять статус выполнения задачи в базе данных.

Содержательная часть решения задачи, то есть непосредственное выполнение самого алгоритма, производится специализированным исполнителем, написанным на C++ внутри запущенного оркестратором контейнера. Исполнитель получает из базы данных необходимые конфигурации и входные данные, а затем приступает к работе, по завершению которой записывает результаты обратно в базу данных.

Стоит отметить, что в веб-приложении Desbordante применяется язык запросов GraphQL [5], который обеспечивает эффективное взаимодействие с сервером путем оптимизации передачи данных и точечного запроса только необходимых ресурсов.

2.2. Архитектура базы данных

Для хранения данных в базе и взаимодействия с ними в Desbordante используется реляционная система управления базами данных PostgreSQL. Реляционные СУБД представляют собой разновидность баз данных, информация в которых организована в виде взаимосвязанных между собой таблиц. Они построены на принципе использования отношений между данными, что обеспечивает структурированное и легко управляемое хранение информации.

В контексте работы с базой данных PostgreSQL в веб-приложении Desbordante, используются два инструмента: объектно-реляционное отображение Sequelize [10] и библиотека pgtools. Объектно-реляционное отображение (object-relational mapping, ORM) — это техника программирования, позволяющая смотреть на базу данных как на набор объектов и их связей, вместо обычных таблиц с данными и SQL запросов. По-

добный подход упрощает понимание устройства базы данных для разработчиков и сильно снижает риск возникновения ошибок. Sequelize является одной из библиотек предоставляющих объектно-реляционное отображение для Node.js. Она сильно облегчает взаимодействия с базами данных PostgreSQL, предоставляя гибкие методы для работы с данными, миграциями, запросами и валидацией.

Библиотека pgtools предоставляет множество инструментов для управления PostgreSQL, среди которых: создание, удаление, восстановление данных. Однако большая часть этих операций может быть легко выполнена с использованием методов из Sequelize. Поэтому в контексте использования ORM принимение pgtools снижаетя до специфических сценариев, где это действительно нужно.

Архитектурно базу данных Desbordante можно разделить на три предметные области:

1. Таблицы, хранящие данные, которые связаны с пользователем, его правами, устройствами, сессиями и прочее;

2. Таблицы, хранящие встроенные и загруженные пользователями файлы с данными, предназначенные для профилирования;

3. Таблицы, хранящие сведения об активных и завершенных задачах, подзадачах, их настройки и результат работы.

2.2.1. Пользовательские таблицы

К пользовательским таблицам относятся таблицы User, Device, Role, Permission, Code, Feedback и Session, которые изображены ниже на рисунке 1.

Таблица User является одной из ключевых и хранит в себе данные о пользовательском аккаунте — имя и фамилия, адрес электронной почты, хеш пароля, статус и другие сведения.

Таблица Role соотносит каждого пользователя с некоторыми ролями, которых может быть несколько. Каждая роль содержит список

идентификаторов из таблицы “Permission”, позволяющих определить перечень разрешений, предоставляемых пользователю. Сама же таблица Permission статична, инициализируется при запуске приложения и в процессе его работы не подвержена изменениям.

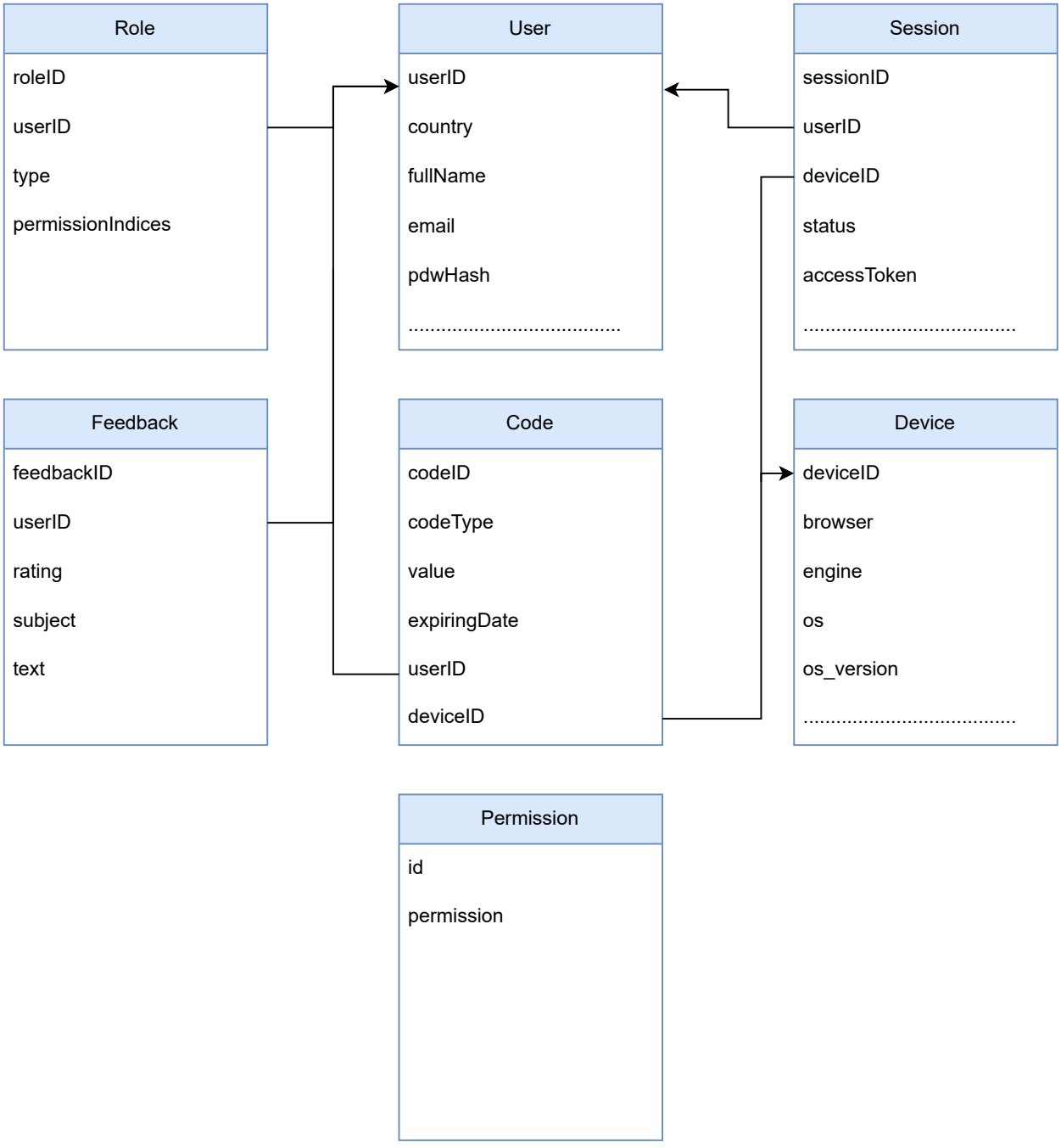

Рис. 1: Пользовательские таблицы

В таблице Session содержатся данные об активных и завершенных сессиях, включая access token, refresh token и другие сведения. Сессия

представляет собой период времени и последовательность взаимодействий, в течение которых пользователь поддерживает активное соединение с веб-приложением. Access token и refresh token, представляют собой уникальные идентификаторы, которые позволяют приложению удостоверить легитимность запросов на доступ к данным и ресурсам. Access token используется для аутентификации пользователя, в то время как refresh token позволяет обновлять access token без повторной аутентификации. Таким образом каждая строка таблицы Session представляет собой активные или завершенные входы пользователя в систему. Каждая сессия определяется пользователем из таблицы User и устройством, на котором она запущена из таблицы Device. Сама таблица Device хранит информацию о различных устройствах, с помощью которых осуществляются сеансы пользователя.

Таблица Code содержит сгенерированные и отправленные пользователю коды для подтверждения адреса электронной почты или изменения пароля. Эти данные играют важную роль в процессе подтверждения и аутентификации пользователей, а также обеспечивают безопасность и надежность операций с учетной записью.

Что касается таблицы Feedback, она служит для сбора обратной связи от пользователей о работе приложения. Эта информация может включать в себя отзывы и предложения по улучшению приложения.

2.2.2. Файловые таблицы

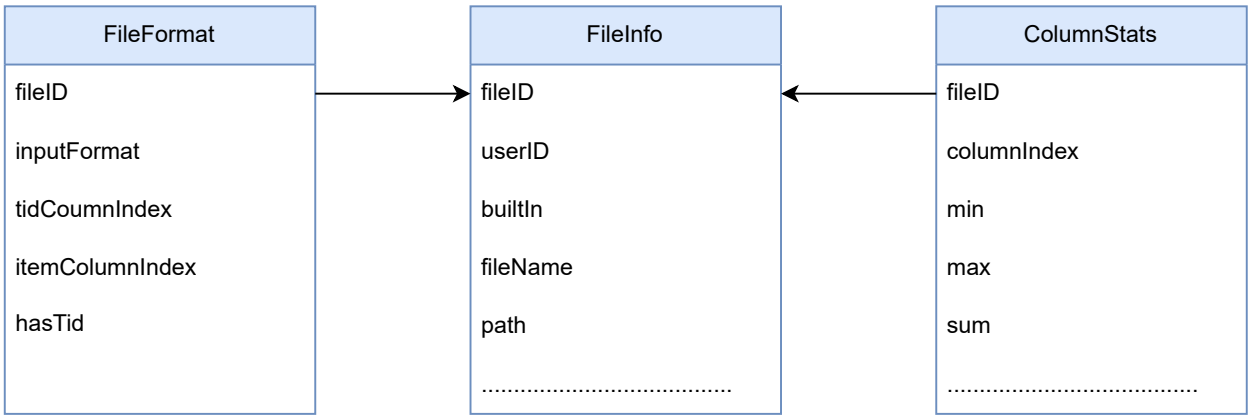

Рис. 2: Таблицы, хранящие сведения о файлах

Информация о файлах, содержащих данные для профилирования хранится в трёх таблицах: FileInfo, FileFormat, ColumnStats.

Таблица базы данных под наименованием FileInfo содержит в себе содержательную часть информации о файле, такую как название файла, путь к файлу, количество строк и столбцов, встроенный ли файл и прочее. Таблица FileFormat содержит сведения о формате транзакционного набора данных, применяемого в анализе ассоциативных правил. Для остальных наборов данных таблица не содержит соответствующей информации. Таблица ColumnStats содержит в себе статистические данные о столбцах таблицы, которая подвергается процессу профилирования. Примером статистик является минимальное и максимальное значения, сумма всех значений столбца и прочие данные.

Все таблицы этой категории в качестве первичного ключа используют идентификатор fileID, который для FileFormat и ColumnStats также является внешним.

2.2.3. Таблицы, хранящие сведения о задачах

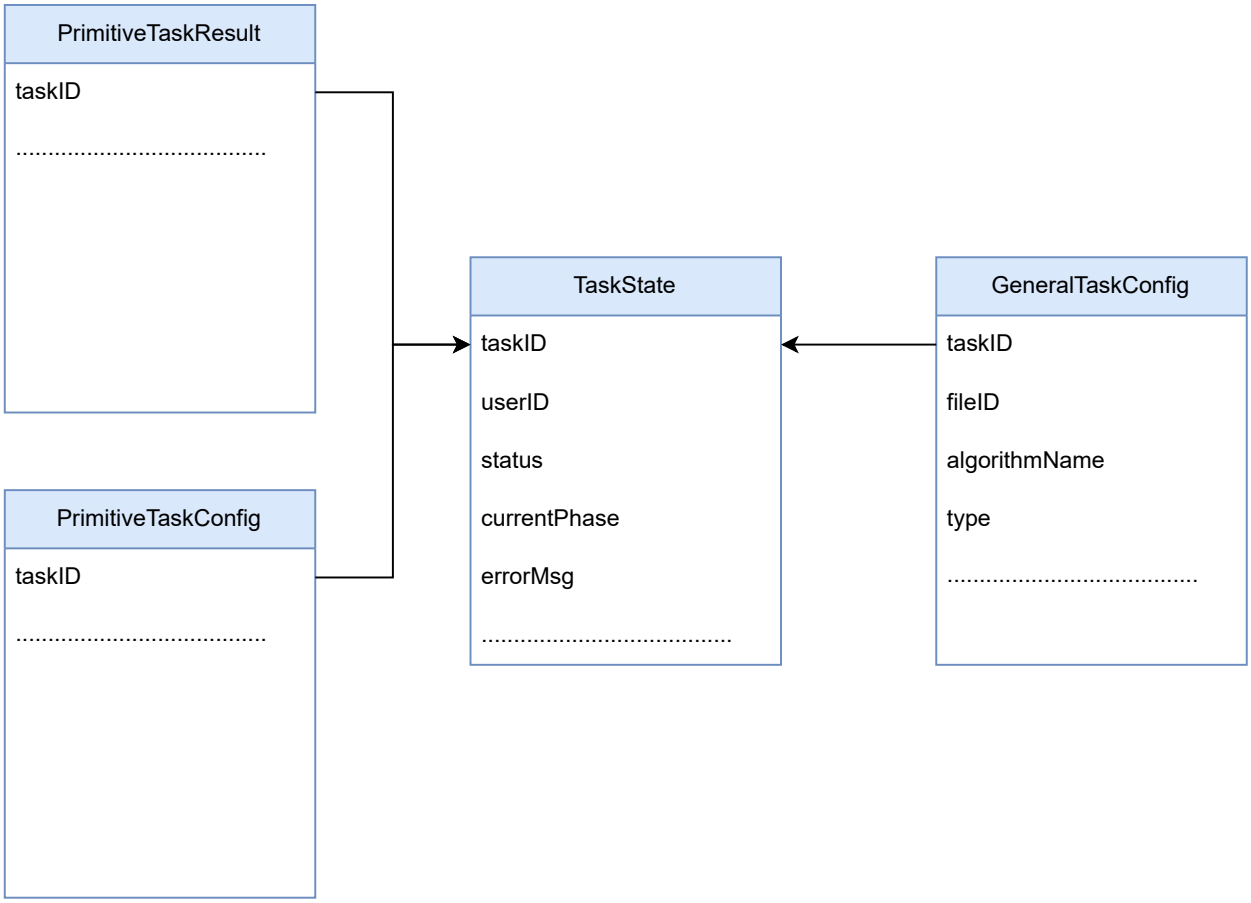

Рис. 3: Таблицы, хранящие сведения о задачах

Часть базы данных, предназначенная для хранения информации об активных и завершенных задачах представлена таблицами TaskState, GeneralTaskConfig, а также множеством таблиц вида PrimitiveTaskConfig и PrimitiveTaskResult, каждая из которых соответствует определенному примитиву. Например, для ассоциативных правил это будут таблицы ARTaskConfig и ARTaskResult. Во всех таблицах связанных с хранением сведений о задачах первичным ключом выступает идентификатор taskID из таблицы TaskState.

Таблица TaskState хранит в себе базовую информацию о каждой задаче: её статус, фазу выполнения, прогресс, номер попытки, время начала выполнения, завершилась ли она, сообщение об ошибке в случае неудачного выполнения алгоритма, какому пользователю она принадлежит и прочее. Каждая задача имеет конфигурацию, в соответствии

с которой она будет выполняться и результат выполнения. Хранение подобной информации реализовано с помощью таблицы GeneralTaskConfig и таблиц вида PrimitiveTaskConfig и PrimitiveTaskResult.

Таблица GeneralTaskConfig содержит общие настройки для задачи, такие как название алгоритма, типы примитивов, идентификаторы файлов, на данных которых будет выполнен алгоритм и прочие общие сведения. А таблицы типа PrimitiveTaskConfig содержат специфические настройки для каждого примитива, которые не применимы к другим. Множество таблиц вида PrimitiveTaskResult содержат данные о результатах выполнения каждой задачи соответствующей данному примитиву. Все эти таблицы связаны с таблицей TaskState следующим образом: у каждой записи в таблице TaskState есть только одна соответствующая запись в таблицах с конфигурациями и результатами, и, наоборот, каждая запись в этих таблицах связана только с одной записью из таблицы TaskState.

2.3. Переписывание веб-сервера на Python

На момент начала работы над проектом, переписывание серверной части Desbordante с использованием языка Python было только начато, и никакие её компоненты ещё не были реализованы. Однако, уже были сделаны определенные шаги в выборе технологий. Для реализации серверной части было решено использовать веб-фреймворк FastAPI [3], ориентированный на создание быстрых и эффективных API. Фреймворк имеет множество особенностей, среди которых: поддержка типизации данных, автоматическая генерация интерфейса OpenAPI [6] и Swagger [11], интеграция с системами автоматической документации.

3. Реализация

В данном разделе описываются мотивация пересмотра и реализации базы данных Desbordante.

3.1. Пересмотр архитектуры

После проведения детального анализа действующей архитектуры базы данных Desbordante, перед началом её переписывания на Python, было принято решение внести в неё некоторые корректировки.

Модель, описывающую таблицу пользователя решено наследовать от готовых моделей в библиотеке FastAPI Users [4], в которых уже реализовано большинство полей и методов. FastAPI Users — это библиотека, спроектированная для обеспечения механизмов аутентификации и авторизации в веб-приложениях, разрабатываемых на FastAPI. Такое решение может быть полезно в будущем, когда нужно будет реализовывать аутентификацию и авторизацию пользователей.

Таблицы, хранящие сведения о файлах, такие как FileInfo, FileFormat, ColumnStats также подверглись некоторым изменениям. В частности, ColumnStats является результатом выполнения задачи по вычислению базовых статистических показателей, а именно поэтому, для поддержания согласованности с структурой базы данных, должна быть отнесена к таблицам вида PrimitiveTaskState и подвергнута аналогичным модификациям, описанным ниже. В очередь, таблица FileFormat, теперь содержит свой собственный первичный ключ, отличный от FileInfo. Однако при этом связь между таблицами остаётся неизменной. Подобный подход представляет собой более удобное и понятное решение по сравнению с ситуацией, когда несколько таблиц используют один и тот же идентификатор в качестве первичного ключа, что способствует избежанию путаницы в структуре данных.

Наибольшим изменениям подверглись таблицы, связанные с хранение сведений о задачах. В предыдущей архитектуре отмечалось избыточное количество таблиц и отношений между ними, что приводило к неоправданной запутанности архитектуры базы данных. Был пере-

смотрен подход к хранению конфигураций задач и результатов их выполнения. Теперь имеется всего одна модель, отвечающая за хранение задач — “Task”, включающая в себя всё те же поля, что и её предыдущий аналог “TaskState”, а также два новых JSON-поля: “config” и “result”. Таким образом, информация о конфигурации задачи и результат её выполнения будут храниться в одной таблице, в виде JSON-строки. Такой подход позволит нам уменьшить число отношений между таблицами, упростить архитектуру базы данных, а также избежать дублирования типов, так как ранее типы полей для конфигураций задач, определяемые в GraphQL-схеме, дублировались в отношениях.

Рис. 4: Таблица, хранящая сведения о задачах, после изменений

В дополнении к пересмотру структуры базы данных были пересмотрены типы столбцов в некоторых таблицах и внесены изменения в названия с целью повышения ясности.

3.2. Детали переписывания моделей

После проведения процесса реинжиниринга базы данных, наступает этап переписывания и адаптирования её моделей под язык программирования Python. В качестве системы управления базами данных было решено продолжить использование СУБД PostgreSQL, также, как в первоначальной реализации. Этот выбор обусловлен обширным набо-

ром функциональных возможностей PostgreSQL для хранения и обработки данных, включая поддержку расширенных типов данных, механизмы транзакций, а также возможность масштабирования для работы с большими объемами информации.

3.2.1. Конфигурирование SQLAlchemy

Для обеспечения эффективного и удобного взаимодействия с базой данных необходимо использовать библиотеки, предоставляющие объектно-реляционное отображение. Одной из таких библиотек является SQLAlchemy [9], разработанная на языке Python специально для работы с реляционными СУБД с применением технологии ORM. Выбор библиотеки SQLAlchemy обусловлен наличием в ней обширного набора инструментов, качественной документации и широкой популярностью в сообществе разработчиков. Она предоставляет механизмы для создания выразительных и мощных запросов к базе данных, что в значительной мере упрощает процесс разработки и уменьшает вероятность возникновения ошибок.

Конфигурирование SQLAlchemy включает в себя установку библиотеки, импортирование и подготовку базовых классов и методов, а также реализацию функций, обеспечивающих удобное соединение с базой данных. В листинге 1 приведен пример конфигурирования SQLAlchemy.

```
1 engine = create_engine(url=settings.postgres_dsn.unicode_string())
2 SessionLocal = sessionmaker(bind=engine, autoflush=False)
3
4
5 def get_session() -> Session:
6     with SessionLocal() as session:
7         yield session
```

Листинг 1: Конфигурирование SQLAlchemy

На первой строке create_engine() представляет собой функцию, которая принимает в качестве аргумента URL-адрес базы данных и создает экземпляр класса “Engine”, служащий для соединения с базой. При этом вызов данной функций не приводит к мгновенному установлению

физического соединения с базой данных: созданный объект Engine будет устанавливать соединение по мере фактического обращения к базе данных. Sessionmaker — функция, которая создает фабрику сессий, предназначенную для создания экземпляров класса Session. Session в SQLAlchemy представляет собой контекст взаимодействия с базой данных, обеспечивающий отслеживание изменений, выполнение запросов и управление транзакциями. После этого на 5 строке определяется генератор сессий get_session, каждый вызов которого генерирует новую сессию. При этом использование контекстного менеджера позволяет работать с сессией, гарантируя, что она будет корректно закрыта после завершения кода в блоке.

3.2.2. Миграции Alembic

Миграции — это механизм, обеспечивающий изменение и согласованность схемы базы данных со временем. Можно представить миграции как инструмент контроля версий для баз данных. Они позволяют эффективно управлять изменениями в структуре и данных в базе, тем самым обеспечивая её целостность и согласованность.

Для работы с миграциями было принято решение использовать Alembic — инструмент, разработанный специально для взаимодействия с SQLAlchemy, что делает его привлекательным выбором для проектов, использующих эту библиотеку. Кроме того, он основан на Python, что позволяет использовать этот язык для эффективного определения миграций.

Подготовка среды для удобной работы с миграциями включала в себя установку Alembic, его инициализацию, передачу пути к базе данных, а также прописыванию инструкций в Makefile для упрощения процесса миграции базы данных.

3.2.3. Переписывание моделей

SQLAlchemy предоставляет разные подходы для определения таблиц в базе данных, включая декларативное описание моделей. Модель

— это класс на языке Python, который ассоциируется с соответствующей таблицей в базе данных, а его атрибуты отражают столбцы этой таблицы. Для того чтобы SQLAlchemy распознала описанный класс как модель базы данных, его необходимо наследовать от некоторого базового класса, который создается путём вызова функции declarative_base из SQLAlchemy ORM.

```python
from sqlalchemy.orm import declarative_base, Mapped, mapped_column, relationship
from sqlalchemy import ForeignKey
from datetime import datetime

Base = declarative_base()

class Feedback(Base):
    __tablename__ = "Feedback"

    id: Mapped[UUID] = mapped_column(primary_key=True, index=True, default=uuid4)
    user_id: Mapped[UUID | None] = mapped_column(
        ForeignKey("User.id", ondelete="SET NULL", onupdate="CASCADE")
    )
    rating: Mapped[int]
    subject: Mapped[str | None] = mapped_column(default=None)
    text: Mapped[str]
    created_at: Mapped[datetime]

    user = relationship("User")
```

Листинг 2: Переписанная модель Feedback

По листингу 2 можно заметить, что модель базы данных ничем не отличается от обычного класса. В начале указывается название таблицы, а затем описываются её столбцы с использованием mapped_column и Mapped. Функция mapped_column используется для создания столбцов в модели базы данных и предоставляет возможность указывать ограничения и свойства для каждого столбца. Mapped является типом данных, который “оборачивает” другие типы данных и предоставляет абстракцию для их использования в декларативном стиле SQLAlchemy.

Для реализации взаимоотношений между таблицами использовались инструменты SQLAlchemy, такие как функция ForeignKey, позволяющая определять внешний ключ в одной таблице, который связан с первичным ключом в другой таблице, а также функция relationship, предоставляющая механизм для установления отношений между таблицами и обеспечивающая удобный доступ и выполнение запросов к связанным данным.

Рассмотреть отношение “один ко многим” можно на примере таблиц User и Role:

```python
class User(SQLAlchemyBaseUserTableUUID, Base):
    __tablename__ = "User"
    full_name: Mapped[str]
    country: Mapped[str]
    company_or_affiliation: Mapped[str]
    occupation: Mapped[str]
    account_status: Mapped[AccountStatusType]
    created_at: Mapped[datetime]
    deleted_at: Mapped[datetime | None] = mapped_column(default=None)

    sessions = relationship("Session", back_populates="user")
    roles = relationship("Role", back_populates="user")
    feedbacks = relationship("Feedback", back_populates="user")
    tasks = relationship("Task", back_populates="user")
    files = relationship("FileInfo", back_populates="user")

class Role(Base):
    __tablename__ = "Role"

    id: Mapped[UUID] = mapped_column(primary_key=True, default=uuid4)
    user_id: Mapped[UUID] = mapped_column(
        ForeignKey("User.id", ondelete="CASCADE", onupdate="CASCADE")
    )
    type: Mapped[RoleType]
    permission_indices: Mapped[str]

    user = relationship("User")
```

Листинг 3: Переписанные модели User и Role

Для установления отношения “один ко многим” в SQLAlchemy внешний ключ родительского класса передаётся в дочерний класс. В представленном случае User выступает в роли родительского класса, а Role в роли дочернего. Таким образом, для установления связи, таблица Role должна содержать столбец, который для каждой роли хранит идентификатор пользователя из таблицы User с использованием функции ForeignKey. Дополнительно, функция relationship устанавливает связь между таблицами и добавляется атрибуты в модели. Таким образом, на 12 строке добавляется атрибут role к модели User, который позволяет получить все роли конкретного пользователя.

Отношение “один к одному” реализовано между большинством таблиц. В качестве примера, можно рассмотреть таблицы FileInfo и FileFormat:

```
1  class FileInfo(Base):
2      __tablename__ = "FileInfo"
3
4      id: Mapped[UUID] = mapped_column(primary_key=True, default=uuid4)
5      created_at: Mapped[datetime] = mapped_column(
6          nullable=False, server_default=text("now()")
7      )
8      user_id: Mapped[UUID | None] = mapped_column(
9          ForeignKey("User.id", ondelete="SET NULL", onupdate="CASCADE"),
       default=None
10     )
11     ..........................................................
12     file_format = relationship("FileFormat", uselist=False)
13     ..........................................................
14
15 class FileFormat(Base):
16     __tablename__ = "FileFormat"
17     id: Mapped[UUID] = mapped_column(primary_key=True, default=uuid4)
18     file_id: Mapped[UUID] = mapped_column(
19         ForeignKey("FileInfo.id", onupdate="CASCADE")
20     )
21     ..........................................................
22     tabular_has_tid: Mapped[bool | None] = mapped_column(default=None)
```

Листинг 4: Переписанные модели FileInfo и FileFormat

Отношение “один к одном” в SQLAlchemy устанавливается также, как и отношение “один ко многим”, за исключением того, что в функцию relationship необходимо передать параметр uselist со значением False. Благодаря этому параметру создаваемый функцией атрибут будет возвращать не список значений, а одно значение.

3.2.4. Описывание структуры JSON-полей, хранящих конфигурацию и результат задачи

В целях уменьшения количества таблиц и отношений между ними, а также упрощения доступа к конфигурациям и результатам задач, было принято решение сохранять их в JSON-полях в таблице Task. Однако, перед этим шагом необходимо описать структуру данных, которые будут храниться в этих JSON-полях. Для данной задачи было принято решение использовать Pydantic [7] — библиотеку для разбора, валидации и анализа данных, построенную на основе классов данных языка Python.

Структуры данных описываются при помощи Pydantic-моделей, которые являются классами Python, наследующиеся от базового класса BaseModel из Pydantic. Атрибуты модели описываются при помощи стандартных типов данных Python, а также типов, предоставляемых другими библиотеками. Примеры Pydantic-моделей, описывающих структуру для общей конфигурации задач, от которой наследуются все специфичные и структуру результатов задач поиска функциональных зависимостей можно увидеть в листинге 5.

```
class BaseTaskConfig(BaseModel):
    algorithmName: str
    primitive_type: DBTaskPrimitiveType
    created_at: datetime
    deleted_at: datetime | None = None

class FDTaskConfig(BaseTaskConfig):
    error_threshold: Annotated[float, Field(ge=0, le=1)]
    max_lhs: Annotated[int, Field(ge=1, le=10)]
    threads_count: Annotated[int, Field(ge=1, le=8)]
```

```
11
12 class FDTaskResult(SpecificTaskWithDepsResult):
13     pk_column_indices: list[int] | None
```

Листинг 5: Примеры Pydantic-моделей, описывающих структуру кофигураций и результатов задач

Благодаря использованию функции Field из библиотеки Pydantic и типа Annotated из стандартной библиотеки Python, мы обретаем возможность накладывать на атрибуты нашей модели некоторые ограничения. Annotated позволяет прикреплять дополнительную информацию к аннотациям типов в Python, в то время, как Field используется для настройки различных аспектов атрибутов модели. Например, в листинге 5, для атрибута error_threshold мы задаём тип float, на который накладываем ограничение, что он должен находиться в диапазоне между 0.0 и 1.0.

С использованием подобных Pydantic-моделей облегчается разбор JSON-данных, содержащих конфигурации и результаты задач. Также легко сформировать JSON-представление из экземпляра нашего класса.

3.3. Подготовка тестового окружения

По мере разработки приложения возникает необходимость тестирования разнообразных компонентов приложения, которые постепенно к нему добавляются. В веб-приложениях значительная часть этих компонент так или иначе взаимодействует с базой данных. В связи с этим становится крайне важным предварительно подготовить некоторые стартовые инструменты для дальнейшего написания тестов, в которых осуществляется взаимодействие с базой данных.

Для разработки тестов и вспомогательных функций было принято решение использовать pytest [8] — фреймворк для тестирования в языке программирования Python. Он является очень гибким, простым в настройке и удобным в использовании, благодаря чему способствует эффективной реализации тестового окружения для веб-приложения.

Для подготовки базы данных к проведению тестов, а также для предоставления тестовым сценариями активных сессий применяются механизмы, предоставляемые pytest, которые известны как фикстуры. Фикстуры позволяют нам предварительно настроить окружение для тестирования и обеспечивают создание контролируемого контекста перед выполнением тестов. В рамках тестирования веб-приложений фикстуры позволяют нам гарантировать стандартизированные условия перед проведением тестов.

Для проведения дальнейшего тестирования веб-приложения Desbordante были разработаны специализированные фикстуры, которые обеспечивают выполнение предварительных действий перед тестированием, таких как создание и очистка базы данных, наполнение её тестовыми данными, предоставление активных сессий, а также удаление базы данных после того, как тестирование будет завершено. Все эти фикстуры различаются по области применения и степени автоматизации. Например, фикстуры, отвечающие за создание и очистку базы данных, работают в контексте session и являются автоматическими, то есть выполняются единожды перед запуском тестов. Фикстуры также могут работать в контексте function, то есть запускаться перед каждым тестовым сценарием. Подобное разграничение обеспечивает достаточную гибкость в процессе тестирования функциональностей, взаимодействующих с базой данных.

Процесс тестирования веб-приложения не должен каким-либо образом воздействовать на основную базу данных. Именно поэтому, написана фикстура prepare_db, предназначенная для создания отдельной тестовой базы данных, с которой будут взаимодействовать тесты и очистки таблиц базы данных, в случае, если она уже существует. Это обеспечивает изоляцию тестовых операций и предотвращает нежелательное вмешательство в основную базу данных при выполнении тестовых сценариев. Ознакомиться с фикстурой можно в листинге 6.

```
1 import pytest
2 from sqlalchemy_utils.functions import create_database, database_exists
3 from app.db import Base
```

```

test_engine = create_engine(test_settings.db_url, pool_pre_ping=True)

@pytest.fixture(scope="session", autouse=True)
def prepare_db():
    if not database_exists(test_settings.db_url):
        create_database(test_settings.db_url)
    Base.metadata.drop_all(bind=test_engine)
    Base.metadata.create_all(bind=test_engine)
```

Листинг 6: Подготовка тестовой базы данных

Фикстура, предоставляющая активную сессию работает также в контексте session и является автоматической. Таким образом, она выполняется в самом начале, перед запуском всех тестов, создает фабрику сессий с помощью sessionmaker, а затем перед выполнением отдельного тестового сценария или фикстуры, которая его запрашивает, будет отдавать активную сессию. Пример такой фикстуры, можно увидеть в листинге 7.

```
import pytest

@pytest.fixture(scope="session", autouse=True)
def get_test_session():
    Session = sessionmaker(test_engine, expire_on_commit=False)
    yield Session
```

Листинг 7: Фикстура, предоставляющая активную сессию

В листинге 8 можно рассмотреть фикстуру clean_tables, которая предназначена для очистки базы данных перед каждым тестовым сценарием с целью предотвращения влияния действий одного теста на результаты другого. Условная конструкция в начале используется для предотвращения автоматического применения фикстуры к тестовым сценариям, вложенным в рамках одного параметризованного теста, если явным образом не указано противоположное.

```
import pytest

@pytest.fixture(scope="function", autouse=True)
def clean_tables(request, session_test):
```

```
    if "fixture_name" in request.fixturenames:
        yield
    else:
        with session_test() as session:
            for table_name in tables:
                table = Base.metadata.tables[table_name]
                session.query(table).delete()
            yield
```

Листинг 8: Фикстура, очищающая таблицы между тестами

3.4. Тестирование

Для тестирования работоспобности и целостности базы данных был написан параметризованный тест, который проверяет идентичность отправленных и полученных из базы данных.

Заключение

В результате работы были выполнены все поставленные задачи, а именно:

- Проведен подробный анализ и обзор устройства серверной части веб-приложения Desbordante, в том числе архитектуры базы данных;
- Спроектирована новая архитектура базы данных;
- Переписаны все модели базы данных на язык Python, с использованием библиотеки SQLAlchemy;
- Подготовлено тестовое окружение, для удобного взаимодействия с базой данных при тестировании самого веб-приложения, а также протестировано взаимодействие с базой данных;

Код реализации доступен в GitHub репозитории[1].

[1] https://github.com/toadharvard/desbordante-server

Список литературы

[1] Aggarwal Charu C., Han Jiawei. Frequent Pattern Mining. — 2014. — URL: https://dl.acm.org/doi/book/10.5555/2677098.

[2] Documentation of Apache Kafka. — URL: https://kafka.apache.org.

[3] Documentation of FastAPI. — URL: https://fastapi.tiangolo.com.

[4] Documentation of FastAPI Users. — URL: https://fastapi-users.github.io/fastapi-users/12.1/.

[5] Documentation of GraphQL. — URL: https://graphql.org/learn/.

[6] Documentation of OpenAPI. — URL: https://learn.openapis.org.

[7] Documentation of Pydantic. — URL: https://docs.pydantic.dev/latest/.

[8] Documentation of Pytest. — URL: https://docs.pytest.org/en/7.1.x/contents.html.

[9] Documentation of SQLAlchemy. — URL: https://docs.sqlalchemy.org/en/20/.

[10] Documentation of Sequelize. — URL: https://sequelize.org/api/v6/identifiers.

[11] Documentation of Swagger. — URL: https://swagger.io.

[12] Fan Wenfei, Geerts Floris et al. Conditional functional dependencies for capturing data inconsistencies. — 2008. — URL: https://dl.acm.org/doi/10.1145/1366102.1366103.

[13] Papenbrock Thorsten et al. Functional dependency discovery: an experimental evaluation of seven algorithms. — 2015. — URL: https://dl.acm.org/doi/10.14778/2794367.2794377.