ORM, миграции, репликации, шардинг

Теоретическая часть

- ORM;
- Миграции;
- Репликация;
- Шардирование.

Практическая часть

- Создание классов для работы с ORM;
- Написание менеджера и миграций.

Object-Relational Mapping (ORM)

Классы

- Классы определяют сущность;
- Классы содержат данные и методы;
- Классы могут наследовать данные и методы др. классов;
- В качестве данных классы могут содержать экземпляры других классов, в том числе и списков.

Базы данных

- Основным элементом является таблица;
- Таблицы содержат простые типы данных;
- Данные могут содержать массивы и списки;
- Таблицы могут быть связаны внешними ключами.

Объектно-реляционное отображение

Прослойка между БД и кодом, которая позволяет записывать/читать данные из БД в виде объектов.

- Позволяет приложениям БД работать с объектами вместо таблиц или SQL;
- Операции выполняются над объектами, а потом прозрачно транслируются в команды БД при помощи ORM;

ORM и Flask

SQLAlchemy — это программное обеспечение с открытым исходным кодом для работы с базами данных при помощи языка SQL.

- Реализует технологию программирования ORM;
- Позволяет описывать структуры БД и способы взаимодействия с ними прямо на языке Python;

ORM и Flask

Устанавливаем

```
# Для работы с ORM.
pip install flask_sqlalchemy
# Для работы с менеджером.
pip install flask_script
# Для работы с миграциями.
pip install flask_migrate
```

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
    'postgresql://username:password@localhost/dbname'
db = SQLAlchemy(app)
```

- SQLALCHEMY_DATABASE_URI - путь/URI базы данных, который будет использоваться для подключения;

Создание модели

```
class Member(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)

    def __init__(self, username):
        self.username = username
```

- Создали новую модель Member, унаследованную от db.Model;
- Далее определяем поля при помощи db.Column;
- db.Column принимает название колонки, тип.

Создание модели. One-to-Many

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    addresses = db.relationship('Address', backref='person', lazy=True)

class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), nullable=False)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'),
        nullable=False)
```

- Вместо db.Column делаем db.relationship;
- `backref` -- объявили, что у Address будет поле person;
- `lazy` определяет, когда SQLAlchemy загружит данные из БД.

lazy

Существует несколько типов, чему может быть равно lazy:

- `'select'/True` ; Загрузит данные, используя select;
- `'joined'/False` ; Загрузит данные, используя join;
- `'subquery'` ; То же, что и joined, но использует подзапрос;
- `'dynamic'` ; Будет полезно, если в дальнейшем захотите применить дополнительные фильтры;

Создание модели. Many-to-Many

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'), primary_key=True),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'), primary_key=True)
)

class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags, lazy='subquery',
        backref=db.backref('pages', lazy=True))

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

- Нужно использовать вспомогательную таблицу для реализации Many-to-Many;

Миграции

Flask-Migrate

pip install Flask-Migrate

- Alembic поддерживает репозиторий миграции, который является каталогом, в котором хранится его сценарии миграции;
- Генерирует скрипт перехода между предыдущим состоянием и следущим;
- Папку migrations нужно добавить в систему контроля версий;
- Скрипты "перехода" между состояниями хранятся в migrations/versions.

С чего начать?

Можно попробовать так...

```
from flask_migrate import Migrate
...
db = SQLAlchemy(app)
# Создадим класс миграции.
migrate = Migrate(app, db)
```

... но в таком случае придётся делать отдельно.

Flask-Script - расширение, предоставляющее поддержку для написания внешних скриптов.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///app.db'

db = SQLAlchemy(app)
migrate = Migrate(app, db)

manager = Manager(app)
manager.add_command('db', MigrateCommand)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128))

if __name__ == '__main__':
```

Репликация

Определение репликации

Репликация (replication) - хранение копий одних и тех же данных на нескольких машинах. Причины репликации данных:

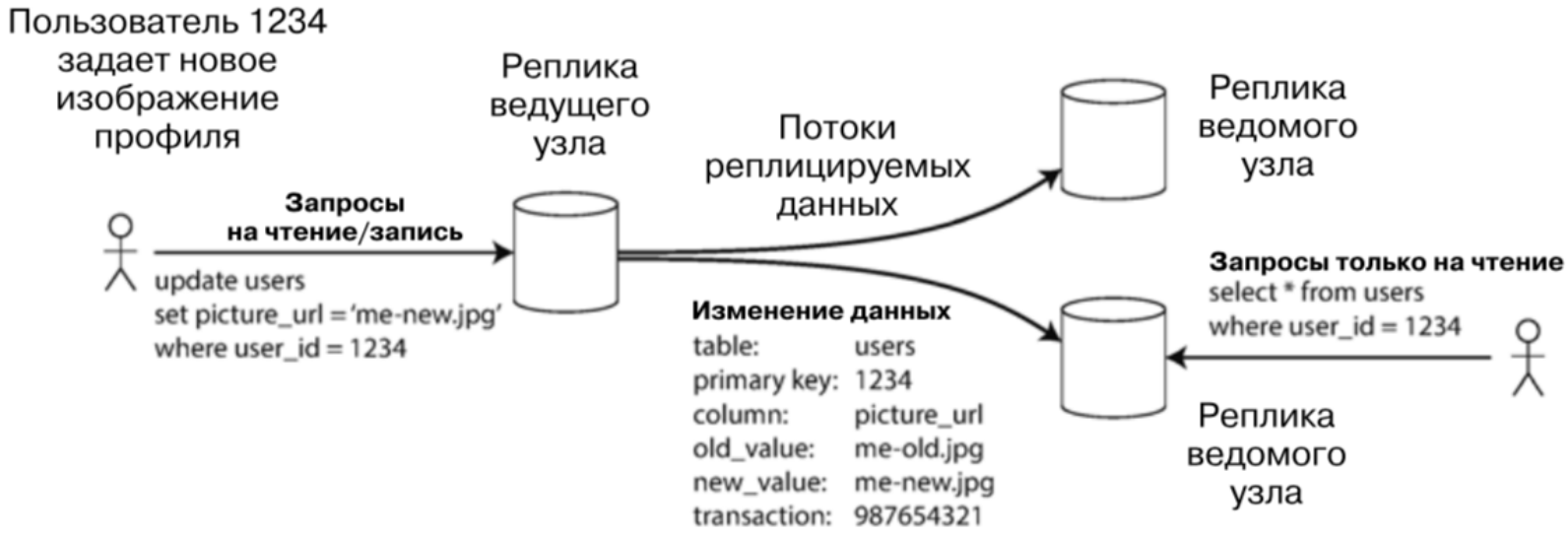
- ради хранения данных географически близко к пользователям (и сокращения, таким образом, задержек);
- чтобы система могла продолжать работать при отказе некоторых ее частей (и повышения, таким образом, доступности);
- для горизонтального масштабирования количества машин, обслуживающих запросы на чтение (и повышения, таким образом, пропускной способности по чтению).

Виды репликации

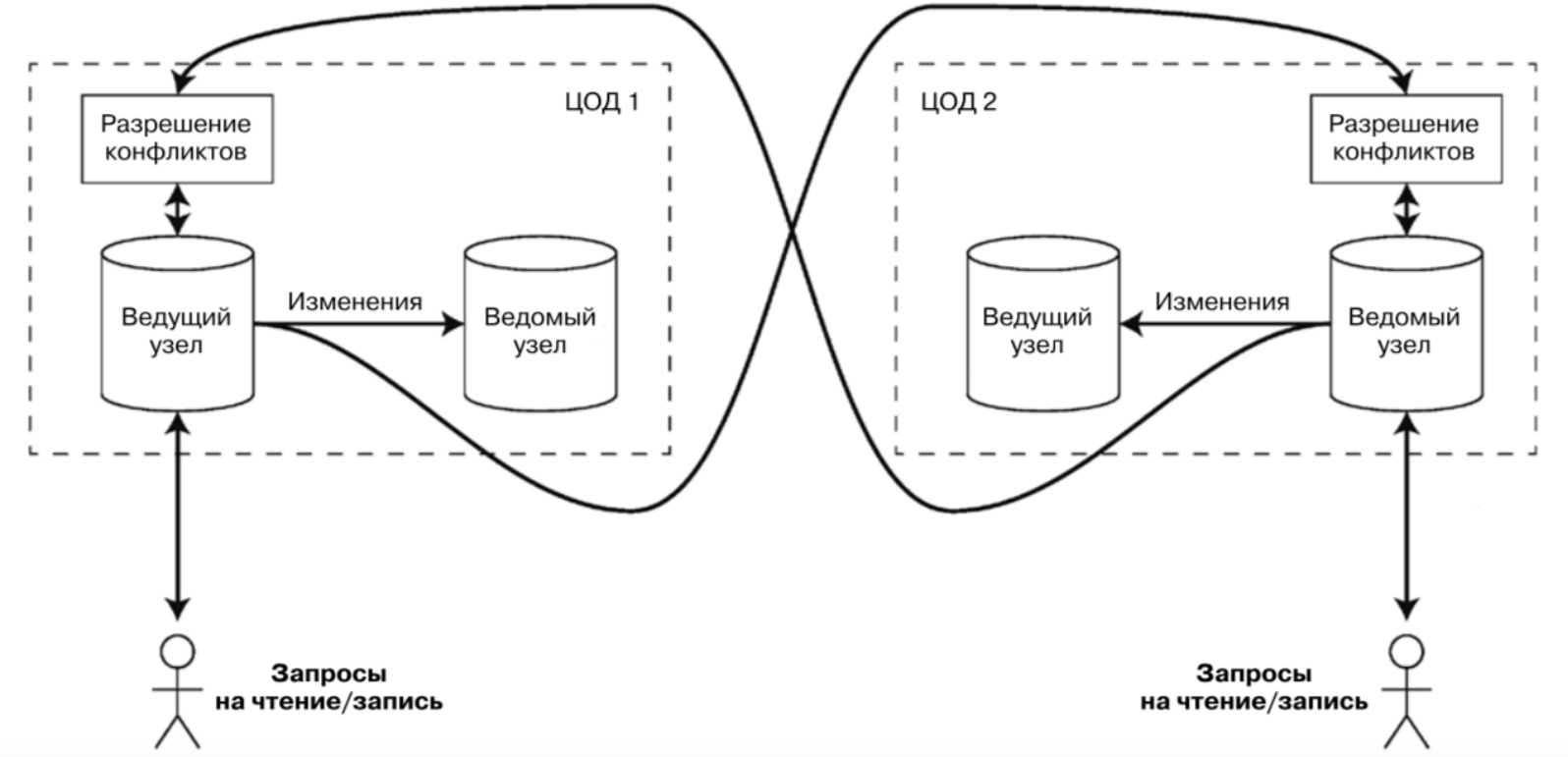
Если реплицируемые данные не меняются с течением времени, то репликация не представляет сложности: просто нужно однократно скопировать их на каждый узел и всё. Основные сложности репликации заключаются в том, что делать с изменениями реплицированных данных.

- С одним ведущим узлом (single-leader);
- С несколькими ведущими узлами (multi-leader);
- Без ведущего узла (leaderless).

С одним ведущим узлом (single-leader)



С несколькими ведущими узлами



Виды репликации

- Производительность.
- Устойчивость к перебоям в обслуживании ЦОДов
- Устойчивость к проблемам с сетью

Без ведущего узла (leaderless)

Клиенты отправляют информацию о каждой из операций записи одному из нескольких узлов и читают из нескольких узлов параллельно, чтобы обнаружить узлы с устаревшими данными и внести поправки.

Шардирование

Что это такое?

В случае очень больших наборов данных или объёмов обрабатываемой информации репликаций недостаточно: необходимо разбить данные на секции (partitions), иначе говоря, выполнить шардинг (sharding) данных. Секционирование (partitioning), представляет собой способ умышленного разбиения большого набора данных на меньшие.

Когда нужно применять шардирование?

- Когда функциональное разбиение и репликация не помогают;
- Разбиваем данные на маленькие кусочки и храним на многих серверах;
- “Единственное” решение для крупного масштаба;
- Нужно аккуратное планирование.

Подходы к секционированию

- Секционирование по диапазонам значений ключа (ключи сортируются и секция содержит все ключи, начиная с определенного минимума до определенного максимума);
- Хеш-секционирование (вычисляется хеш-функция каждого ключа и к каждой секции относится определенный диапазон хешей);
- “Единственное” решение для крупного масштаба;
- Нужно аккуратное планирование.

Домашнее задание

- Переписать БД из первого семестра на модели (4 балла)
- Сделать миграцию и закоммитить её (2 балла);
- Переписать методы, пищущие в БД, с использование ORM (4 балла);

Срок сдачи: следующее занятие.