Урок 6

Работа с базами данных в Python

Особенности использования баз данных в Python и Django.

Python DB-API и реляционные базы данных

- ***В чем суть использования Python DB-API при взаимодействии с реляционными базами данных?***

Использование **DB-API** является одним из подходов, на основе которых реализуется взаимодействие Python-кода с реляционными базами данных. **DB-API** представляет собой не конкретную библиотеку, а набор правил или стандарт, которому подчиняются модули, реализующие взаимодействие с базами данных. Существует несколько таких модулей, и в каждом реализован Python DB-API, а также механизм взаимодействия Python-кода с определенной базой данных.

Стандарт Python DB-API может иметь отличия применительно к различным базам данных, но существуют общие принципы, благодаря которым при работе с разными БД используется один подход.

Шаблон взаимодействия с базами данных в Python по стандарту **DB-API** имеет следующую структуру:

1. **Импорт библиотеки**, соответствующей определенному типу базы данных.
2. **Создание объекта соединения базы данных**. При этом используется метод **connect()** импортированного модуля. В качестве аргумента методу передается ссылка на файл базы данных, с которым предполагается вести работу.
3. **Создание объекта курсора.** Используется метод **cursor()** объекта соединения, созданного на предыдущем шаге.
4. **Выполнение операций с базой данных.** Все операции с данными строятся вокруг объекта курсора. Чтобы выполнить запрос, достаточно вызвать один из трех методов (**execute**, **executescript** или **executemany**) объекта «курсор». Сам запрос представляет собой стандартную SQL-инструкцию и передается в качестве аргумента перечисленных методов.
5. **Подтверждение всех незавершенных транзакций.** Используется метод **commit()** объекта соединения.
6. **Закрытие соединения.** Используется метод **close()** объекта соединения.

Пример шаблона для реализации взаимодействия с БД **SQLite**:

```
import sqlite3

conn = sqlite3.connect('test.sqlite')

cur = conn.cursor()

cur.execute("текст_запроса_изменяющего_содержимое_БД")

conn.commit()

conn.close()
```

Метод **execute()** позволяет выполнить обычный SQL-запрос, который передается в качестве аргумента запроса в кавычках. При написании запросов на обновление БД этому методу может передаваться второй аргумент, содержащий кортеж значений, которые необходимо записать в поля БД. При этом поля, для которых необходимо выполнить запись значения, идентифицируются с помощью заполнителя, обозначаемого символом **?**.

Примеры использования метода **execute()**:

```
cursor.execute('SELECT * FROM table_test')
cursor.execute("
    SELECT field_1 FROM table_test
    WHERE field_2=? AND field_3=?",
    (field_2_val, field_3_val)
)
cursor.execute("
    INSERT INTO table_test VALUES (?, ?, ?)",
    (field_1_val, field_2_val, field_3_val)
)
```

Метод **executescript()** запускает последовательность SQL-запросов (SQL-сценарий). На начало сценария и его окончание указывает конструкция **"""**.

Примеры использования метода **executescript()**:

```
cur.executescript("""
    create table users(
        login,
        password,
        email
    );
    create table vendors(
        name,
        phone,
        address
    );
    insert into users(login, password, email)
    values ('user_1', 'user_1', 'user_1@mail.ru');
""")
```

Метод **executemany()** обеспечивает множественную вставку записей (запрос типа **insert**), «пробегая» по коллекции данных.

Примеры использования метода **executemany()**:

```
users = [
    ("user_1", "user_1", "user_1@mail.ru"),
    ("user_2", "user_2", "user_1@mail.ru"),
]
cur.executemany("insert into users(login, password, email) values (?, ?, ?)",
users)
```

Если запрос предполагает выборку данных, после его запуска необходимо выполнить обработку результата. При этом используются методы **fetchone()** и **fetchall()** объекта «курсор». Метод **fetchone()** возвращает одну запись БД в форме кортежа, содержащего значения полей БД для данной записи. Метод **fetchall()** возвращает все отобранные записи в форме кортежа кортежей.

Модули, реализующие Python DB-API

- ***Какие существуют модули, реализующие Python DB-API и обеспечивающие взаимодействие с различными СУБД?***

Чтобы обеспечивать взаимодействие Python-кода с реляционными базами данных, используются четыре модуля, в каждом из которых реализован стандарт Python DB-API:

1. **sqlite3**. Обеспечивает взаимодействие с СУБД SQLite. Это единственный модуль, который уже интегрирован в стандартную библиотеку Python и не требует дополнительной установки.
2. **psycopg2**. Модуль, который используется для организации взаимодействия с СУБД PostgreSQL. Наряду с остальным модулями не входит по умолчанию в библиотеку Python и требует установки. Установка:

```
pip install psycopg2
```

Для корректной работы модуля в ОС Linux может потребоваться установка зависимостей этого пакета:

```
apt-get build-dep python-psycopg2
```

3. **mysql-connector**. Модуль обеспечивает связь Python-кода с СУБД MySQL. Установка:

```
pip install mysql-connector
```

4. **pyodbc**. Модуль реализует взаимодействие с СУБД MSSQL Server. Установка:

```
pip install pyodbc
```

Преимущества SQLite

- ***В чем преимущества СУБД SQLite? Каковы ее основные команды?***

SQLite считается одной из наиболее популярных СУБД при разработке небольших и учебных проектов на Python и Django. У нее есть ряд преимуществ перед другими СУБД:

- Возможность установки и эффективного использования практически на всех типах операционных систем;

- Отсутствие необходимости в настройке или администрировании. SQLite распространяется с нулевой конфигурацией;
- Экономичность. Файлы баз данных SQLite имеют очень компактный формат и требуют минимального размера дискового пространства;
- Данная СУБД не требует наличия отдельной серверной процессорной системы;
- СУБД SQLite распространяется в качестве автономного приложения и не требует установки дополнительных зависимостей;
- Свободная лицензия, что позволяет распространять SQLite вместе с собственным программным продуктом;
- Безопасность хранения. Все данные хранятся в одном файле, правами доступа к которому можно управлять стандартными средствами ОС;
- Возможность использования с различными языками программирования. Доступ к базе данных SQLite можно организовать средствами не только Python, но и Java, C#, Ruby и других языков.

В модуле SQLite предусмотрены стандартные SQL-команды взаимодействия с реляционными базами данных:

- **SELECT** — выполнить выборку данных из таблицы по определенном условию);
- **CREATE** — создать таблицу базы данных;
- **INSERT** — добавить одну или нескольких записей в таблицу;
- **UPDATE** — обновить значения для столбцов таблицы;
- **DELETE** — удалить одну или нескольких записей из таблицы;
- **DROP** — удалить таблицу.

Приложения для работы с базами данных

- ***Какие существуют оконные приложения для работы с базами данных?***

Это стандартные программы с графическим пользовательским интерфейсом. Они называются браузерами баз данных и позволяют просматривать и редактировать таблицы с данными. Благодаря браузерам баз данных можно вручную создавать таблицы и соответствующие им записи. Рассмотрим браузеры для самых популярных СУБД Python- и Django-проектов.

Для СУБД SQLite:

- **SQLite Database Browser**. Свободно распространяемый кроссплатформенный визуальный инструмент на русском языке, совместимый с SQLite;
- **SQLiteStudio**. Открытый менеджер баз данных, поставляемый как портативное приложение (не требует установки).

Для СУБД PostgreSQL:

- **pgAdmin**. Открытое кроссплатформенное приложение, предоставляющее графический интерфейс для взаимодействия с СУБД.

Работая с данными программами, необходимо подтверждать выполняемые операции. Например, после добавления новой записи (строки) таблицы необходимо подтвердить, что внесены изменения.

Обработка ошибок

- ***Как организовать обработку ошибок при взаимодействии с БД?***

Обработка ошибок позволяет обеспечить устойчивость программы, особенно при записи данных. При реализации обработки ошибок необходимо Python-инструкцию операции с базой данных «обернуть» в блок **try-except-else**. При этом если возникнет ошибка, в зависимости от ее характера может генерироваться исключение.

После оператора **try** необходимо поместить блок, в котором может быть сгенерировано исключение. После оператора **except** с названием исключения надо указать, какие действия должны быть выполнены при генерации указанного исключения. После оператора **else** следует указать инструкции, которые выполняются при отсутствии исключений, то есть когда код оператора **try** выполнен корректно.

Пример

```
try:
    cursor.execute(sql_instr)
    cursor.fetchall()
except sqlite3.DatabaseError as err:
    print("Error: ", err)
else:
    conn.commit()
```

Полный перечень возможных исключений для Python DB-API:

- **Warning** — исключение, создаваемое для важных предупреждений (**warnings**);
- **Error** — базовое класс для исключений типа **error**;
- **InterfaceError** — ошибки, свойственные интерфейсу БД;
- **DatabaseError** — исключения, свойственные базе данных;
- **DataError** — исключения обработки данных (деление на ноль, выход за границы диапазона);
- **OperationalError** — исключения, относящиеся к операциям БД, которые программист не контролирует (неожиданное отключение БД, неизвестное имя источника данных, невозможность выполнить транзакцию, ошибка выделения памяти);
- **IntegrityError** — исключение, создаваемое при нарушении целостности отношений (неудачная проверка внешнего ключа);
- **InternalError** — внутреннее исключение БД (некорректный курсор, ошибка синхронизации транзакции и прочее);

- **ProgrammingError** — программные ошибки (таблица не найдена или уже присутствует, ошибка SQL-синтаксиса, неверное количество параметров);

- **NotSupportedError** — исключение создается при использовании метода или API, который не поддерживается базой данных.

Использование ORM-библиотеки SQLAlchemy

- ***В чем суть использования ORM-библиотеки SQLAlchemy при взаимодействии с реляционными базами данных? Как ее установить?***

Второй подход в реализации взаимодействия Python-кода с реляционными базами данных заключается в использовании ORM-библиотеки SQLAlchemy (ORM — это Object-Relational Mapper). Это библиотека, написанная на языке Python и позволяющая осуществлять взаимодействие с реляционными базами данных с помощью ORM-технологии. В SQLAlchemy реализована возможность описывать структуру базы данных с помощью Python-инструкций без необходимости писать SQL-запросы. В SQLAlchemy предусмотрена возможность работы с базами данных SQLite, PostgreSQL, MySQL.

Автоматическая генерация SQL-кода посредством библиотеки SQLAlchemy — эффективная альтернатива ручному написанию SQL-запросов. Она обеспечивает:

1. **Безопасность**. Минимальная вероятность SQL-инъекций за счет экранирования параметров запросов.

2. **Переносимость**. Код базы данных пишется один раз, независимо от выбранной СУБД. При смене базы данных не надо менять код запросов, достаточно изменить строку соединения.

3. **Высокая производительность.** Проекты, реализованные на базе ORM-библиотеки SQLAlchemy, значительно реже сталкиваются с проблемами при выполнении запросов.

Для установки SQLAlchemy необходимо выполнить следующую инструкцию:

```
pip install SQLAlchemy
```

Или загрузить архив с SQLALchemy с официального сайта библиотеки и выполнить установочный скрипт **setup.py**:

```
python setup.py install
```

Чтобы проверить правильность выполненной установки, необходимо запустить команду отображения версии библиотеки:

```
import sqlalchemy

print("Версия SQLAlchemy:", sqlalchemy.__version__)
```

Чтобы установить соединение с нужной СУБД средствами SQLAlchemy, используют функцию **create_engine()**. В качестве ее параметра передается указание на URL создаваемого Engine-объекта базы данных. Также указывается параметр **echo**, устанавливающий ведение журналирования посредством стандартного модуля **logging**. Если этот параметр установлен в значение **true**, запускается команда отображения всех создаваемых SQL-запросов. Результатом выполнения функции является экземпляр класса **Engine**, который представляет способ выполнения запроса с помощью DB-API.

Пример создания подключения к СУБД SQLite средствами SQLALchemy:

```
from sqlalchemy import create_engine

engine = create_engine('sqlite:///:memory:', echo=True)
```

В приведенном фрагменте кода определено, что файл базы данных должен создаваться не в физической (на диске), а в оперативной памяти вычислительного устройства. Использование такого подхода дает значительный рост производительности.

Рассмотрим примеры установки соединений с другими СУБД.

PostgreSQL:

```
from sqlalchemy import create_engine

engine = create_engine(
'postgresql+psycopg2://user:password@/dbname'
)
```

MySQL:

```
from sqlalchemy import create_engine

engine = create_engine(
'mysql+pymysql://user:pass@127.0.0.1:3306/db', pool_recycle=3600
)
```

Параметр **pool_recycle** позволяет задать автоматическую переустановку соединения через каждые два часа. Это может потребоваться в связи с автоматическим обрывом соединения при простое в течение 8 часов и более.

Для выполнения запросов к базе данных используется метод **execute()** объекта соединения. Аргументом метода выступает текст SQL-запроса, например:

```
engine = create_engine('sqlite:///:memory:', echo=True)
result = engine.execute("select * from test_table")
print(result.fetchall())
```

Традиционный подход к созданию таблиц, классов и отображений в SQLALchemy

- ***В чем суть использования традиционного подхода к созданию таблиц, классов и отображений в ORM SQLAlchemy?***

В соответствии с традиционным подходом для создания таблицы в SQL Alchemy используется конструктор **Table()**. Как его параметры передаются название таблицы, объект коллекции **MetaData** и необходимое количество объектов-столбцов с названием и типом данных в качестве параметров. Например:

```
from sqlalchemy import Table, Column, Integer, String, MetaData

metadata = MetaData()
vendors_table = Table('vendors', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('phone', String),
    Column('address', String)
)
```

Второй параметр конструктора указывает на метаданные (набор определений таблиц). Объекты **Table** и **Column** реализуются в области метаданных, предоставляющей эти объекты. В представленном выше примере создаются четыре столбца, которые могут содержать данные указанного в скобках типа. Допустимая длина содержимого каждой колонки не указана. Это вполне допустимо для СУБД SQLite, но если работа ведется с СУБД PostgreSQL или MySQL, то необходимо явно указать длину содержимого столбца, например:

```
Column('address', String(100))
```

После описания структуры таблицы следует перейти непосредственно к ее созданию. Для этого предназначен метод **create_all()**, содержащий указание на объект соединения (**engine**) с базой данных. Метод первым делом запускает проверку наличия указанной таблицы и при отсутствии создает ее:

```
metadata.create_all(engine)
```

Когда таблица создана, необходимо перейти к созданию класса — то есть объекта, через который данные будут передаваться в таблицу. Создадим класс для только что подготовленной таблицы:

```
class Vendor:

    def __init__(self, name, phone, address):
        self.name = name
        self.phone = phone
        self.address = address

    def __repr__(self):
        return "<Vendor('%s','%s', '%s')>" %
                                        (self.name, self.phone, self.address)
```

В этом примере представлена функция **__repr__**, которая позволяет определить строковое представление объекта.

Теперь чтобы созданный класс передавал данные в соответствующую таблицу, необходимо выполнить привязку таблицы **vendors** и класса **Vendor**. Для этого применяется функция **mapper** пакета **sqlalchemy.orm**:

```
from sqlalchemy.orm import mapper

mapper(Vendor, vendors)
vendor = Vendor(
"ООО ‘Компани’ ", "8(495)77-77-77", "г. Москва, ул. Бажова, д. 9"
)
```

Функция **mapper()** отвечает за создание отображения между таблицей **vendors** и классом **Vendor**. Чтобы добавить новый объект **Vendor**, следует вызвать конструктор соответствующего класса и передать в него необходимые параметры.

Декларативный подход к созданию таблиц, классов и отображений в SQLALchemy

- ***В чем суть использования декларативного подхода к созданию таблиц, классов и отображений в ORM SQLAlchemy?***

В предыдущем разделе был рассмотрен пример традиционного подхода к использованию SQLAlchemy. Он ценится за разделение задач, однако у большинства приложений отсутствуют жесткие требования по этому критерию, и для них SQLAlchemy предлагает альтернативный стиль — декларативный.

Важным элементом данного подхода является функция **declarative_base()**, определяющая новый класс-родитель для всех необходимых пользовательских ORM-классов. Рассмотрим уже знакомый пример создания таблицы **vendors**, реализованный уже в декларативном стиле:

```
Base = declarative_base()

class Vendor(Base):
    __tablename__ = 'vendors'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    phone = Column(String)
    address = Column(String)

    def __init__(self, name, phone, address):
        self.name = name
        self.phone = phone
        self.address = address

    def __repr__(self):
        return "<Vendor('%s','%s', '%s')>" %
                                        (self.name, self.phone, self.address)
```

Описание таблицы и определение ее класса находятся в рамках одного класса. Этот пример демонстрирует лаконичность декларативного подхода работы с SQLALchemy по сравнению с классическим.

Добавление данных в таблицу в SQLALchemy

- ***Как добавлять данные в таблицу в SQLAlchemy?***

Когда создана таблица базы данных и соответствующий ей класс, а также добавлен экземпляр этого класса и в него переданы данные, необходимо реализовать запись данных в таблицу. Для доступа к базе данных используется механизм сессии **Session**.

При создании соединения с базой следует определить класс **Session**, выступающий фабрикой объектов сессий:

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
```

Если в этот момент Engine-объект базы данных не определен, создать сессию можно с помощью следующей инструкции:

```
Session = sessionmaker()
```

Когда с помощью метода **create_engine()** будет установлено подключение к базе данных, необходимо связать это подключение с сессией с помощью метода **configure()**:

```
Session.configure(bind=engine)
```

Класс **Session** отвечает за генерацию Session-объектов, привязанных к базе данных. Когда требуется установить связь с базой данных, необходимо создать объект этого класса:

```
session = Session()
```

Для добавления созданного объекта **Vendor** необходимо включить его в имеющеюся сессию, выполнив вызов метода **add()** сессии или **add_all()** для добавления нескольких объектов:

```
vendor = Vendor(
"ООО ‘Компани’ ", "8(495)77-77-77", "г. Москва, ул. Бажова, д. 9"
)
session.add(vendor)
```

После вызова метода **add()** новая запись в базе данных еще не появится. Чтобы изменения вступили в силу, нужно выполнить команду **commit()**, которая означает завершение транзакции.

```
session.commit()
```

Чтобы получить, например, идентификатор новой записи базы данных, теперь необходимо обратиться к стандартному полю **id** объекта **vendor**:

```
print('Vendor ID:', vendor.id)
```

Взаимодействие с базами данных в Django

- ***Как взаимодействие с базами данных реализовано в Django?***

Взаимодействие с базами данных в Django-проектах выполнено с помощью механизма объектно-ориентированного отображения. Эта функциональность реализована не только в Django, но и в других современных веб-фреймворках. Преимуществом использования системы ORM является автоматизация множества основных взаимодействий с базой данных — благодаря использованию объектно-ориентированного подхода как альтернативы SQL-инструкциям.

ORM-система представляет собой промежуточное звено между базой данных и кодом, который пишет разработчик. Благодаря этой системе у разработчика нет необходимости вручную писать SQL-запросы, чтобы взаимодействовать с базой данных. Достаточно создать объект и сохранить его в БД. Чтение данных из базы также выполняется посредством этого объекта.

У Django-разработчиков есть возможность вместо встроенной ORM-системы использовать популярный пакет SQLAlchemy. Это мощный инструмент, который при этом требует больших усилий программиста при написании кода. Встроенная ORM-система в Django в этом отношении более привлекательна. Кроме того, она предусматривает автоматическую генерацию интерфейса администрирования.

Миграции, их назначение и создание

- ***Для чего в Django используются миграции, и как они создаются?***

Миграции в Django-проектах — это средство синхронизации созданных моделей с файлом базы данных. Миграции позволяют перенести изменения в моделях на таблицы базы данных. Физически они представляют собой файлы, содержащие Python-классы.

Чтобы в файле базы данных проекта появились таблицы, соответствующие моделям, недостаточно только создать сами модели. Необходимо выполнить команду создания миграций и команду их применения к базе данных. После этого изменения, внесенные в модели, будут отражены в соответствующих таблицах.

Миграции являются своего рода системой контроля версий для базы данных проекта. При этом команда **makemigrations**, отвечающая за создание файлов миграции, позволяет сохранить состояние моделей в этих файлах. А команда **migrate** отвечает за применение миграции к базе данных.

Файлы миграций добавляются в каталоге **migrations** каждого созданного приложения. Они являются элементами приложения и распространяются вместе с остальным его кодом. Важно, что Django создает миграции при любых изменениях, вносимых в модель, — даже тех, которые не влияют на базу данных.

Для работы с миграциями используются две основные инструкции. Первая:

```
python manage.py makemigrations
```

Запускает сканирование всех моделей проекта и их сравнение с версиями, содержащимися в файлах миграций. Создает новые миграции, если обнаружены изменения. После запуска команды необходимо проверять их вывод, чтобы отслеживать, как команда видит изменения, внесенные в модель.

Вторая:

```
python manage.py migrate
```

Запускает применение к файлу базы данных, созданных на предыдущем шаге миграций.

Предыдущие команды указывают Django, что необходимо сканировать изменения в моделях и вносить их в таблицы применительно ко всему проекту. Чтобы создать и применить миграции для отдельного приложения, следует к представленным выше командам добавить название приложения, например:

```
python manage.py makemigrations catalog
python manage.py migrate catalog
```

Для работы с миграциями в Django может применяться еще одна команда — **sqlmigrate**. Пример:

```
python manage.py sqlmigrate catalog
```

Данная команда выполняется после создания миграций. Она позволяет вывести на экран SQL-запросы, которые будут генерироваться для Django командой применения миграций к базе данных (то есть к таблицам, соответствующим указанному приложению).

SQLite или PostgreSQL

- ***На каких Python-проектах эффективнее использовать СУБД SQLite и PostgreSQL?***

На собеседованиях работодатель часто требует знания особенностей работы с PostgreSQL. Многие проекты реализуются на основе этой СУБД:

- **Приложения, рассчитанные на значительную нагрузку.** Если доступ к приложению должны иметь более 30–40 пользователей и они должны использовать одну базу данных, то оптимальным вариантом будет PostgreSQL.

- **Приложения, требующие выполнения операций записи данных в больших объемах.** PostgreSQL поддерживает использование массовых операций записи данных в любой момент.

При разработке учебных проектов, как правило, возможностей СУБД SQLite оказывается достаточно, и нет необходимости подключать проект к другим СУБД. Но и в условиях реальных проектов SQLite также может использоваться в качестве основной. Это могут быть:

- **Встроенные приложения.** Это программы, не требующие расширения, —например, мобильные или игровые приложения;

- **Небольшие десктопные приложения.** Аналитические, расчетные системы, программы учета, рассчитанные на малые группы пользователей;

- **Приложения для тестирования** — например, логики работы бизнес-приложений.

Среди разработчиков существует мнение, что СУБД SQLite предназначена исключительно для тестирования приложений и решения учебных задач. Но на деле SQLite вполне допустима в коммерческих проектах. Ее возможностей достаточно для обеспечения до 100 тысяч ежедневных просмотров веб-ресурса. SQLite идеальна для проектов с невысокой нагрузкой, где есть повышенные требования к мобильности, простоте, скорости, а также критичен объем потребляемой памяти. Высоконагруженные проекты принято разрабатывать на базе полнофункциональных СУБД (MySQL и PostgreSQL).

MongoDB, ее преимущества и недостатки

- ***Что такое MongoDB, каковы ее преимущества и недостатки?***

MongoDB представляет собой документо-ориентированную СУБД, в которой — в отличие от привычных реляционных СУБД PostgreSQL и MySQL — не используется табличный способ представления данных со связями через внешние ключи. В MongoDB заложен принцип хранения документов в формате **BSON** (Binary JSON). Главное, что необходимо понять относительно этой СУБД: *хранение данных не в таблицах, а с помощью коллекций*. При этом данные, хранящиеся в коллекции MongoDB, не привязаны к определенной схеме. Таким образом, в MongoDB JSON-документы хранятся в коллекциях.

Преимущества:

- Использование документо-ориентированного принципа хранения (в MongoDB реализована простая и мощная JSON-подобная схема данных);
- Наличие гибкого языка для формирования запросов, поддержка динамических запросов, профилирования запросов, поддержка индексов;
- Наличие эффективной системы хранения двоичных данных больших объемов — например, видеофайлов, изображений;
- Наличие системы логирования операций, приводящих к изменениям в базе данных;
- Поддержка механизмов отказоустойчивости и масштабируемости данных, возможность реализации доступа к данным, расположенным на нескольких серверах.

Недостатки:

- Запросы строятся не на SQL, а на собственном языке, изучение которого может потребовать времени и сил;
- Несовершенные инструменты перевода SQL-запросов в MongoDB. Они доступны, но в качестве основного средства перевода их рассматривать нельзя.
- Продолжительное время установки, требовательность к ресурсам вычислительного устройства (памяти, объему дискового пространства).
- Данная СУБД уступает конкурентам в решении сложных и многокомпонентных задач.

Области применения MongoDB

- ***Когда следует применять MongoDB?***

1. Высокопроизводительные распределенные веб-приложения, где выдвигаются особые требования по гибкости структуры и необходимо работать с большими объемами данных. Примерами таких ресурсов могут быть сервисы amazon.com, ebay.com, netflix.com.
2. Приложения с пропорциональным разделением нагрузки — например, если задачи чтения и записи данных из базы находятся в соотношении 1:1.
3. Приложения с минимальным объемом транзакций. MongoDB идеально подходит для приложений, в которых не нужно выполнять операции с данными, а необходимо оперативно получить и отобразить данные.
4. Проекты, в которых взаимосвязь объектов до конца не определена и существует вероятность дальнейших изменений.

Масштабирование и его реализация в MongoDB

- ***Что такое масштабирование баз данных, для чего оно применяется и как реализовано в MongoDB?***

Задача масштабирования возникает перед разработчиками в связи с ростом объема используемых данных и выполняемых над ними операций. Классическая стратегия с использованием одного

сервера базы данных утрачивает свою эффективность, поскольку сервер в этом случае перестает справляться с нагрузкой. Необходимо масштабировать данные, то есть разделять их на группы и отдельные сервера. Основные стратегии масштабирования — репликация и шардинг.

Особенности использования стратегий масштабирования применительно к MongoDB:

- **Репликация в MongoDB.** Представляет собой синхронизацию данных между кластером серверов. Помимо масштабируемости повышает доступность и сохранность данных. Позволяет восстанавливать данные после аппаратных сбоев. В MongoDB реализованы две формы репликации: реплисеты (**Replica Sets**), «ведущий-ведомый» (**Master-Slave**).
- **Шардинг в MongoDB.** Предусматривает хранение отдельных частей данных на разных серверах. Позволяет решить проблему горизонтального масштабирования. Пример — хранение данных пользователей с фамилиями, начинающимися на буквы A–M, на одном сервере, а остальных — на другом.

Установка MongoDB и ее использование в Python

- ***Как установить MongoDB и организовать работу с базой в Python?***

Установка MongoDB в ОС Windows не представляет сложности и выполняется в несколько шагов:

- **Загрузка установщика** в формате **msi** с официального сайта или других ресурсов.
- **Запуск установщика**, выполнение этапов установки приложения.
- **Создание каталога C:\data\db**, поскольку в ОС Windows по умолчанию MongoDB сохраняет базы данных по этому пути.
- **Запуск сервера.** Необходимо перейти в каталог **C:\Program Files\MongoDB\bin** из консольного приложения и запустить команду **mongod**. Это приложение, отвечающее за запуск сервера MongoDB.
- **Запуск оболочки.** Операции с базами данных в MongoDB могут выполняться с помощью специальной оболочки, которую предоставляет файл **mongo.exe**. Он располагается в каталоге **C:\Program Files\MongoDB\bin**.

Для работы с MongoDB из Python используется библиотека **pymongo**. Она не встроена в стандартную библиотеку Python и требует отдельной установки:

```
pip install pymongo
```

Один экземпляр MongoDB реализует поддержку нескольких независимых баз данных. В отдельной БД может храниться несколько коллекций, каждая из которых является эквивалентом таблицы в реляционных базах данных. Коллекция представляет собой группу JSON-документов. Их можно назвать аналогами записей в таблицах реляционных баз данных. Далее приведены примеры операций с базой данных MongoDB.

Устанавливаем соединение с сервером базы данных:

```
import pymongo

# объект_соединения = pymongo.MongoClient('host', port)
# например,

con = pymongo.MongoClient('localhost', 27017)
```

Получаем список всех баз данных:

```
# переменная_список_баз_данных = pymongo.MongoClient().database_names()
# например,

dbs = pymongo.MongoClient().database_names()
```

Подключаемся к одной из баз данных:

```
# объект_БД = объект_соединения['имя_БД']
# например,

db = con['test']
```

Выводим все коллекции для выбранной базы данных:

```
# переменная_список_коллекций = объект_БД.collection_names()
# например,

colls = db.collection_names()
```

Получаем доступ к определенной коллекции базы данных:

```
# переменная_коллекции = объект_БД['имя_коллекции']
# например,

col = db['users']
```

Вставляем документ в коллекцию:

```
# объект_БД.имя_коллекции.insert({"ключ":"значение"})
# например,

db.users.insert({"name":"John"})
```

Получаем документ:

```
# переменная_документа = переменная_коллекции.find_one({"ключ": "значение"})
# например,

doc = col.find_one({"name": "John"})
```

Получаем все документы из коллекции:

```
# for документ_коллекции in переменная_коллекции.find():
    # print(документ_коллекции)
# например,

for obj in col.find():
    print(obj)
```

Обновляем документ:

```
# объект_БД.имя_коллекции.update(
# {"ключ":"старое_значение"},{"ключ":"новое_значение"}
# )
# например,

db.users.update({"name":"John"}, {"name":"Jack"})
```

Удаляем документ:

```
# объект_БД.имя_коллекции.remove({"ключ":"значение"})
# например,

db.users.remove({"name":"Jack"})
```

Практическое задание

Практическое задание шестого и седьмого урока — разработать десктопное приложение с графическим интерфейсом пользователя, предназначенное для ведения простого складского учета. Это приложение с оконным интерфейсом будет реализовано в привязке к СУБД SQLite и позволит заносить в базу данных сведения о номенклатуре товаров, поставщиках и сотрудниках предприятия.

На данном этапе работы с проектом выполним первую часть практическое задания: подготовим фрагменты программного кода, отвечающие за создание таблиц базы данных. В седьмом уроке отображение этих таблиц реализуем в специальных виджетах. Ниже приведено описание необходимых блоков программного кода.

Создать файл базы данных

С помощью одного из менеджеров баз данных (например, SQLiteStudio) создать пустой файл SQLite-базы данных.

Создать подключение к базе данных

Выполнить импорт модуля с Python DB-API для реализации взаимодействия с СУБД SQLite. Создать подключение к базе данных, путь к которой записан в переменную **db_path**. Создать объект-курсор для выполнения операций с данными.

Создать вспомогательные таблицы

- ***Категории товаров.*** Написать запрос создания таблицы **categories** (с проверкой ее существования). Таблица должна содержать два поля: **category_name** (категория), **category_description** (описание). Все поля должны быть не пустыми. Поле **category** должно быть первичным ключом.

- ***Единицы измерения товаров.*** Написать запрос создания таблицы **units** с проверкой ее существования. Таблица должна содержать одно поле — **unit** (единица измерения). Оно должно быть не пустым и выступать первичным ключом.

- ***Должности***. Написать запрос создания таблицы **positions** (с проверкой ее существования). Таблица должна содержать одно поле — **position** (должность). Оно должно быть не пустым и выступать первичным ключом.

Создать основные таблицы

- ***Товары.*** Написать запрос создания таблицы **goods** с проверкой ее существования. Таблица должна содержать четыре поля: **good_id** (номер товара — первичный ключ), **good_name** (название товара), **good_unit** (единица измерения товара — внешний ключ на таблицу **units**), **good_cat** (категория товара — внешний ключ на таблицу **categories**).

- ***Сотрудники.*** Написать запрос создания таблицы **employees** с проверкой ее существования. Таблица должна содержать три поля: **employee_id** (номер сотрудника — первичный ключ), **employee_fio** (ФИО сотрудника), **employee_position** (должность сотрудника — внешний ключ на таблицу **positions**).

- ***Поставщики.*** Написать запрос создания таблицы **vendors** с проверкой ее существования. Таблица должна содержать шесть полей: **vendor_id** (номер поставщика — первичный ключ), **vendor_name** (название поставщика), **vendor_ownerchipform** (форма собственности поставщика), **vendor_address** (адрес поставщика), **vendor_phone** (телефон поставщика), **vendor_email** (email поставщика).

Дополнительные материалы

1. ORM, или как забыть о проектировании БД.
2. SQLAlchemy — старт.
3. SQLAlchemy Quickstart.
4. Создание приложения Django и подключение к базе данных.
5. Django: SQLite или PostgreSQL.
6. Курс молодого бойца PostgreSQL.

7. Типы полей моделей.
8. MongoDB и Python.
9. За и против: когда стоит и не стоит использовать MongoDB.
10. Ответы на вопросы на собеседование MongoDB.
11. Подходящее применение для SQLite.

Используемая литература

1. Python: Работа с базой данных, часть 1/2: Используем DB-API.
2. Python: модуль sqlite3. Перевод документации с примерами.
3. Документация Django.
4. Модели Django.
5. Эффективная работа с моделями Django.
6. 20 вопросов и ответов на знание базы данных SQLite.
7. SQLAlchemy.
8. Django учебник Часть 3: Использование моделей.
9. SQLAlchemy.
10. SQLAlchemy ORM.
11. Установка и начало работы с MongoDB на Windows.
12. Руководство по PyMongo.