GeekBrains

Методы сбора и обработки данных из сети Интернет

Системы управления базами данных MongoDB и SQLite в Python

Введение в основные принципы работы с реляционными и нереляционными базами данных. Основные операции и методы для формирования запросов. Работа с данными внутри баз.

SQL и noSQL

- SQL
- NoSQL

MongoDB и почему именно она

- Документная модель данных
- Произвольные запросы
- Другие преимущества

Работа с MongoDB

- Установка MongoDB
- Запуск сервера
- Работа с MongoDB из консоли
- Работа с MongoDB в Python

Операции CRUD
Операции Create
Добавление одного документа
Добавление нескольких документов
Операции Read
Запрос всех документов в коллекции
Запрос документов по заданному условию
Запрос документов по нескольким критериям
Запрос документов по составным условиям (оператор $and)
Запрос документов по составным условиям (оператор $or)
Запрос документов по составным условиям (совместное использование $or и $and)
Проекции
Операции Update
Обновление одного документа
Обновление нескольких документов
Замена одного документа
Операции Delete
Удаление одного документа
Удаление нескольких документов
SQLAlchemy. Работа с SQLite
Установка и подключение к БД
Создание таблицы в базе данных
Определение класса Python для отображения в таблицу
Настройка отображения
Декларативное создание таблицы, класса и отображения за один раз
Создание сессии
Добавление новых объектов
Сохранение в базу
Практическое задание
Используемая литература

При написании руководства были использованы ОС: Windows 10 x64, Python v3.7.2, PyCharm Edu v2019.3.2, MongoDB v4.2.0

SQL и noSQL

СУБД (DBMS) расшифровывается как система управления базами данных (Database Management System). Это набор команд, прикладных и инфраструктурных приложений и библиотек, которые позволяют управлять и обслуживать базу данных — по сути, как API для программиста.

База данных (БД) — структурированный массив данных. Он может располагаться как на жёстком диске, так и в оперативной памяти. Существует множество типов СУБД, но сейчас актуальны два типа — реляционные (SQL) и нереляционные (noSQL).

SQL

Реляционные БД называются так потому, что они основаны на реляционной модели данных. Само понятие «реляционность» происходит от английского слова relation, то есть «отношение». Реляционная структура имеет зависимости, явные связи между собой. Для управления данными и программирования логики используются язык SQL 1992 года выпуска и процедурная надстройка над ним. У разных производителей свои реализации: например, у Oracle это PL/SQL (Procedural Language / Structured Query Language), у Microsoft — Transact-SQL (T-SQL), а у IBM — SQL PL.

Есть два основных способа организации БД: OLTP (Online Transaction Processing) и OLAP (Online Analytical Processing). OLTP — транзакционный тип организации БД для проведения маленьких транзакций в режиме реального времени. При разработке БД данного типа организации обычно поддерживает набор правил, который обозначается акронимом ACID:

- **A**tomicity (атомарность) — гарантия того, что все действия, произведенные в атомарной транзакции, будут либо применены, либо отменены (rollback) полностью.
- **C**onsistency (согласованность) — если действия транзакции привели к рассинхронизации данных, она будет отменена.
- **I**solation (изолированность) — во время выполнения транзакции параллельные транзакции не должны влиять на результат выполнения изначальной. Это крайне дорогое требования для работы в режиме реального времени, поэтому есть различные режимы.
- **D**urability (устойчивость) — это свойство легче объяснить на примере. Представьте себе: идёт процесс применения транзакции, и после фиксации транзакции обесточили ЦОД, сервер СУБД выключился. После включения сервера изменения, сделанные в транзакции, останутся зафиксированными.

NoSQL

NoSQL (от англ. not only SQL — не только SQL) — термин, обозначающий ряд подходов, направленных на реализацию систем управления базами данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных СУБД с доступом к данным средствами языка SQL. Применяется к базам данных, в которых делается попытка решить проблемы масштабируемости и доступности за счёт атомарности (англ. atomicity) и согласованности данных.

В NoSQL вместо ACID может рассматриваться набор свойств BASE:

- базовая доступность (***b**asic **a**vailability*) — каждый запрос гарантированно завершается (успешно или безуспешно).
- гибкое состояние (***s**oft state*) — состояние системы может изменяться со временем, даже без ввода новых данных, для достижения согласования данных.
- согласованность в конечном счёте (***e**ventual consistency*) — данные могут быть некоторое время рассогласованы, но приходят к согласованию через некоторое время.

Всего есть 4 типа NoSQL баз данных:

- Ключ-значение (Redis, Berkeley DB).

  Хранилища типа key-value — простейшие БД, чаще всего это in-memory базы данных (то есть они хранятся и работают в оперативной памяти сервера). Такой БД не требуется ни схемы, ни связей.

- Документоориентированные (MongoDB, CouchDB).

  В таких БД данные хранятся в виде документов и имеют сложную иерархическую (древовидную) структуру, между элементами есть связи.

- Графовые (Giraph, Neo4j).

  Такие БД позволяют хорошо реализовать семантические паутины (например, социальные сети), в подобных задачах они более производительны.

- BigTable (HBase, Cassandra).

  В них данные представлены в виде разреженной матрицы. Эти БД похожи на документоориентированные.

MongoDB и почему именно она

MongoDB – документоориентированная СУБД с открытым исходным кодом, не требует схемы таблиц и относится к NoSQL. MongoDB хранит данные в виде документов в формате JSON. Масштабирование в MongoDB реализовано при помощи технологии шардинга.

Преимущества MongoDB:

- скорость разработки;
- нет необходимости в поддержке схемы и в коде, и в БД;
- лёгкая масштабируемость;
- гибкость при смене задачи;
- удобство работы с денормализованными данными.

Почему MongoDB подходит под задачу сбора и анализа данных:

- данные быстро меняются (дополнительные данные из API, динамический контент в HTML-страницах);
- меняя схему, надо менять и приложение, и БД;
- БД нужна лишь до тех пор, пока нужны данные;
- данные постоянно обновляются;
- нормализация не нужна;
- задача не меняется;
- одно приложение.

Документная модель данных

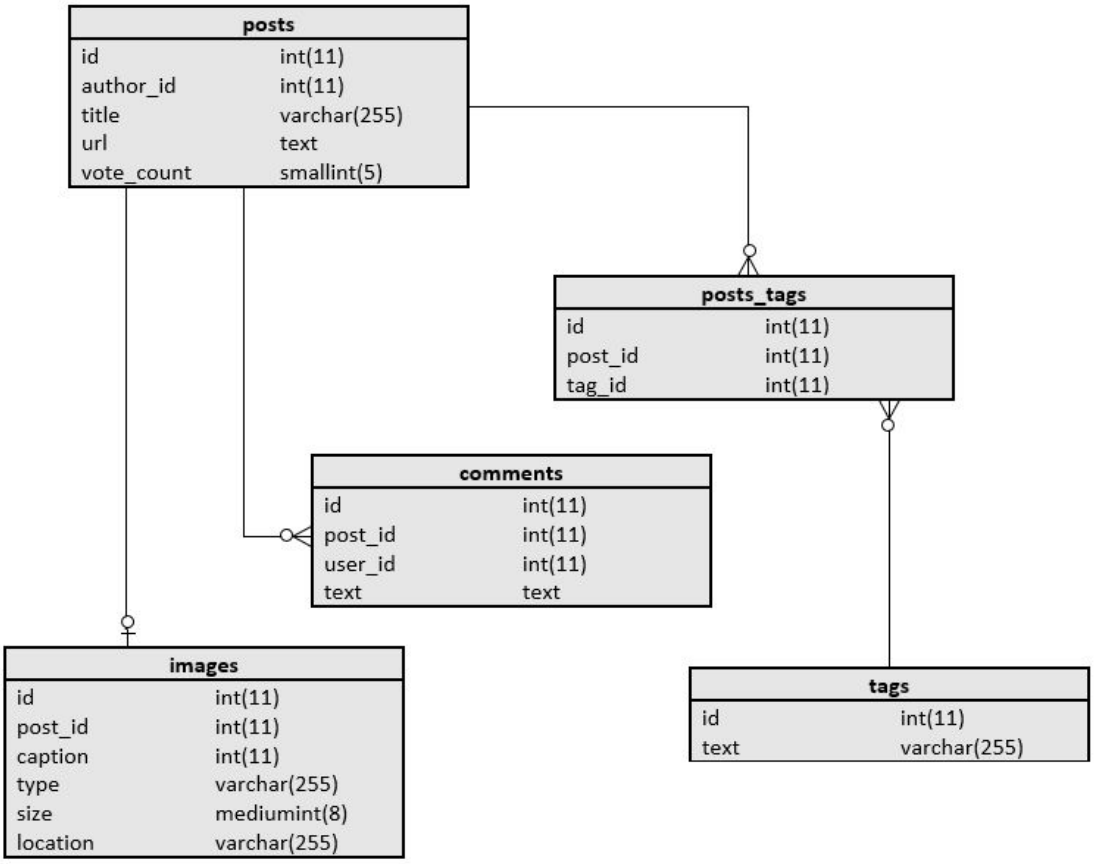
Модель данных MongDB — документо-ориентированная. Для тех, кто не знаком с идеей документа в контексте баз данных, продемонстрировать её проще всего на примере.

```
{_id: ObjectID("4bd9e8e17cefd644108961bb"),    #Поле _id - первичный ключ
 title: "Adventure in Databases",
 url: "http://example.com/databases.txt",
 author: "msmith",
 vote_count: 20,

 tags: ['databases', 'mongodb', 'indexing'],  #Теги хранятся в виде массива
строк

 image:{                                      #Атрибут указывает на другой
элемент
      url: "http://example.com/db.jpg",
      caption:"",
```

```
      type: "jpg",
      size: 75381,
      data: "Binary"
},
  comments: [                       #Комментарии хранятся в виде массива
объектов,
    {                               #представляющих еще один комментарий
      user: "bjones",
      text: "Interesting article!"
    },
    {
      user: "blogger",
      text: "Another related article is at http://example.com/db/db.txt"
    }
  ]
}
```

Здесь приведён пример документа, представляющего статью на социальном новостном сайте. Как видите, документ — это по существу набор, состоящий из имён и значений свойств. Значение может быть представлено простым типом, например, строки, числа и даты. Но может быть также последовательностью и даже другим документом. С помощью таких конструкций можно представлять весьма сложные структуры данных. Так, в нашем примере имеется свойство tags — список, в котором хранятся ассоциированные со статьёй теги. Но еще интереснее свойство comments, которое ссылается на список документов, содержащих комментарии.

Сравним это с представлением тех же данных в стандартной реляционной базе:



Здесь изображён типичный реляционный аналог. Поскольку таблицы по сути своей плоские, для представления связей типа «один-ко-многим» необходимо несколько таблиц. Мы начинаем с таблицы posts, в которой хранится основная информация о каждой статье. Затем создаем ещё три таблицы, каждая из которых содержит поле post_id, ссылающееся на исходную статью.

Вероятно, вы обратили внимание, что документы не только позволяют представлять данные со сложной структурой, но и не нуждаются в заранее определённой схеме. В реляционной базе данных строки хранятся в таблице. У каждой таблицы — строго определённая схема, описывающая, какие столбцы и типы данных допустимы. Если окажется, что необходимо добавить ещё одно поле, то таблицу надо будет явно изменить.

В MongoDB документы группируются в коллекции — контейнеры, не налагающие на данные какую-либо схему. Теоретически у каждого входящего в коллекцию документа может быть своя структура, но на практике документы в одной коллекции похожи друг на друга. Например, у всех документов в коллекции posts имеются поля title, tags, comments и т. д.

Произвольные запросы

При проектировании MongoDB ставилась задача по возможности сохранить выразительную мощь запросов, которая считается принципиально важной особенностью реляционных СУБД. Рассмотрим принцип построения запросов в MongoDB на простом примере статей и комментариев. Пусть

необходимо найти все статьи, помеченные тегом politics, за которые проголосовало более 10 посетителей. SQL-запрос для решения этой задачи выглядел бы так:

```
SELECT * FROM posts
INNER JOIN posts_tags ON posts.id = posts_tags.post_id INNER JOIN tags ON
posts_tags.tag_id == tags.id
WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

Эквивалентный запрос в MongoDB формулируется путем задания документа-образца. Условие «больше» обозначается специальным ключом $gt.

```
db.posts.find(('tags': 'politics', 'vote_count': {'$gt': 10}});
```

Здесь стоит отметить, что в этих запросах предполагаются разные модели данных. SQL-запрос опирается на строго нормализованную модель, в которой статьи и теги хранятся в разных таблицах, тогда как в запросе для MongoDB считается, что теги хранятся внутри документа, описывающего статью. Однако же в обоих случаях имеется возможность формулировать запрос с произвольной комбинацией атрибутов, что и составляет смысл понятия произвольных запросов.

Другие преимущества

По словам создателей, MongoDB проектировалась так, чтобы объединить лучшие черты хранилищ ключей и значений и реляционных баз данных. Хранилища ключей и значений благодаря своей простоте работают чрезвычайно быстро и относительно легко масштабируются. Масштабировать реляционные базы труднее, по крайней мере по горизонтали, зато они предлагают развитую модель данных и мощный язык запросов. Если считать, что MongoDB находится посредине между этими двумя крайностями, то получается СУБД, которая легко масштабируется, позволяет хранить сложные структуры данных и предоставляет развитые механизмы формулирования запросов.

Если говорить о сценариях использования, то MongoDB хорошо подходит в качестве основного хранилища данных для веб-приложений, аналитических приложений, протоколирования и для любых приложений, которым необходим кеш среднего класса. Кроме того, поскольку MongoDB позволяет хранить бессхемные данные, то она удобна, когда структура данных заранее неизвестна.

Работа с MongoDB

Установка MongoDB

MongoDB — кроссплатформенная СУБД. Чтобы скачать дистрибутив, нужно перейти по ссылке: Download Center: Community Server, выбрать свою ОС и нажать кнопку Download.

В случае с ОС Windows, скачается установочный пакет с расширением *.msi. Его необходимо установить, следуя подсказкам мастера установки.

Устанавливается по умолчанию по адресу: C:\Program Files\MongoDB\Server\4.2\bin.

Чтобы сервер было удобнее запускать, добавьте этот каталог в переменную среды PATH.

Запуск сервера

Для старта работы серверной части mongoDB необходимо запустить исполняемый файл **mongod.exe**.

Важно!

Все базы данных MongoDB создает по умолчанию в каталоге C:\data\db\.

Его необходимо заранее создать вручную. Если каталог будет отсутствовать, сервер не запустится и выдаст ошибку:

I STORAGE [initandlisten] exception in initAndListen: NonExistentPath: Data directory C:\data\db\ not found., terminating

При запуске можно указать каталог с базами данных, передав его через параметр **--dbpath**. Например, если базы лежат в D:\bases, нужно запускать:

```
mongod.exe --dbpath D:\bases
```

Сервер запускается и привязывается к адресу localhost (127.0.0.1) на порту 27017. Об этом свидетельствуют две записи из логов:

```
I CONTROL  [initandlisten] ** WARNING: This server is bound to localhost.
I NETWORK  [initandlisten] waiting for connections on port 27017
```

Работа с MongoDB из консоли

Начиная с версии MongoDB 3.2, в качестве графической оболочки поставляется MongoDB Compass. Существуют и другие сторонние проекты, которые предлагают инструменты с GUI для администрирования и просмотра данных.

Для управления и администрирования системы базы данных можно использовать также **mongo**. Это интерактивная оболочка, которая позволяет разработчикам и администраторам просматривать, вставлять, удалять и обновлять данные в базе, настроить репликацию, сегментирование, отключить узлы, выполнять любые запросы к базе данных.

Для старта работы утилиты, необходимо запустить исполняемый файл **mongo.exe**:

```
> mongo.exe
MongoDB shell version v4.2.0
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("43e494b7-805f-4d73-bffa-db3e4c2df69d")
}
MongoDB server version: 4.2.0
```

Чтобы узнать, какие базы данных у вас есть

```
> show dbs
admin               0.000GB
config              0.000GB
local               0.000GB
test                0.000GB
```

Чтобы подключиться к текущей базе данных или создать новую, используем команду use:

```
> use newdb
switched to db newdb
```

Важно!

В MongoDB используется принцип экономии: наша СУБД не будет создавать саму базу данных до тех пор, пока она пустая.

Убедимся, что база данных не появилась в нашем списке:

```
> show dbs
admin               0.000GB
config              0.000GB
local               0.000GB
test                0.000GB
```

Чтобы оперировать с данными в выбранной базе данных, используется синтаксис: объект_db.имя_коллекции.метод.

Создадим документ внутри коллекции docs:

```
> db.docs.insertOne({name:"test doc", size: 1000, author: "Konan Varvar"})

      "acknowledged" : true,
      "insertedId" : ObjectId("5d7a44cf430835d59abbf83f")
}
```

Проверим ещё раз наличие нашей БД в списке:

```
> show dbs
admin               0.000GB
config              0.000GB
local               0.000GB
newdb               0.000GB
test                0.000GB
```

Теперь видим, что новая БД создалась и присутствует в общем списке.

Действует и обратное — если в базе данных не остаётся ни одной коллекции, то она удаляется из общего списка. Удалим только что созданную коллекцию из БД:

```
> db.docs.drop()
true
```

Проверим снова список существующих баз данных:

```
> show dbs
admin               0.000GB
config              0.000GB
local               0.000GB
test                0.000GB
```

Как видим, база данных newdb также удалилась.

Работа с MongoDB в Python

Прежде, чем начать, необходимо установить модуль PyMongo:

```
pip install pymongo
```

Первый шаг при работе с PyMongo – это подключение модуля MongoClient:

```
from pymongo import MongoClient
```

Далее, создаем клиента для подключения к серверу. Сервер в нашем примере запущен локально без изменений параметров по умолчанию:

```
client = MongoClient('localhost', 27017)
```

Один клиент может работать с несколькими независимыми базами данных. Когда работаем с PyMongo, нам необходимо подключиться к базе данных, указав её имя:

```
db = client['test_database']
```

В рамках одной базы данных мы можем работать с несколькими коллекциями. Создадим указатель на коллекцию, чтобы проще было обращаться к ней:

```
collection = db.test_collection
```

Операции CRUD

CRUD-операции — это создание (Create), чтение (Read), обновление (Update), Удаление (Delete)

Операции Create

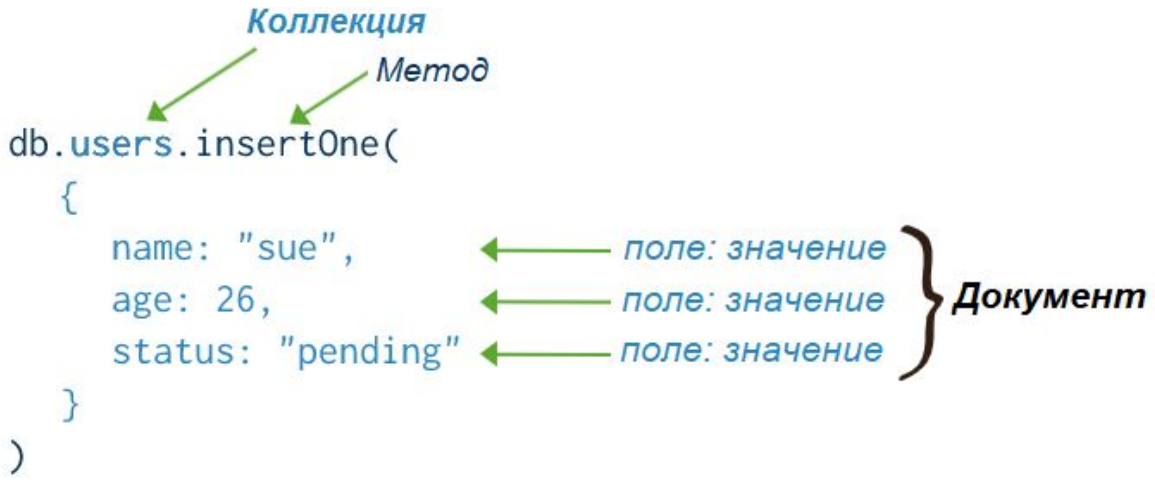
Операции Create (или их иногда называют Insert) добавляют новый документ в коллекцию. MongoDB предоставляет следующие методы для того, чтобы добавить новые документы в коллекцию:

- db.collection.insertOne();
- db.collection.insertMany().

Особенности при создании/добавлении нового документа:

1. Если коллекция, указанная при создании документа, не существует, то она создаётся.
2. В MongoDB каждый документ, записываемый в коллекцию, должен иметь поле _id с уникальным значением, которое выступает как первичный ключ. Если у добавляемого документа нет этого поля, MongoDb автоматически сгенерирует ObjectId для поля _id.
3. В MongoDB операции insert всегда работают только с одной коллекцией. Нельзя вставить документ сразу в несколько коллекций, используя метод один раз. Также важно понимать, что все операции записи в БД атомарны на уровне одного документа.

Общий синтаксис операции:



Добавление одного документа

db.collection.insertOne() добавляет в коллекцию один документ, переданный в качестве параметра метода:

```
db.inventory.insertOne(
   { item: "canvas",
     qty: 100,
     tags: ["cotton"],
     size: { h: 28, w: 35.5, uom: "cm" } }
)
```

Метод **insertOne()** также возвращает свой документ, но уже с новым, добавленным в него полем _id.

Добавление нескольких документов

db.collection.insertMany() добавляет несколько документов в коллекцию. Для этого необходимо передать список документов в качестве параметра метода:

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21,
uom: "cm" } },
   { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" }
},
   { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85,
uom: "cm" } }
])
```

В указанном выше примере ни один из документов также не имеет поля _id. В этом случае MongoDB добавляет его к каждому документу из списка. Метод **insertMany()** также возвращает документ, который содержит все новые добавленные документы с полями _id.

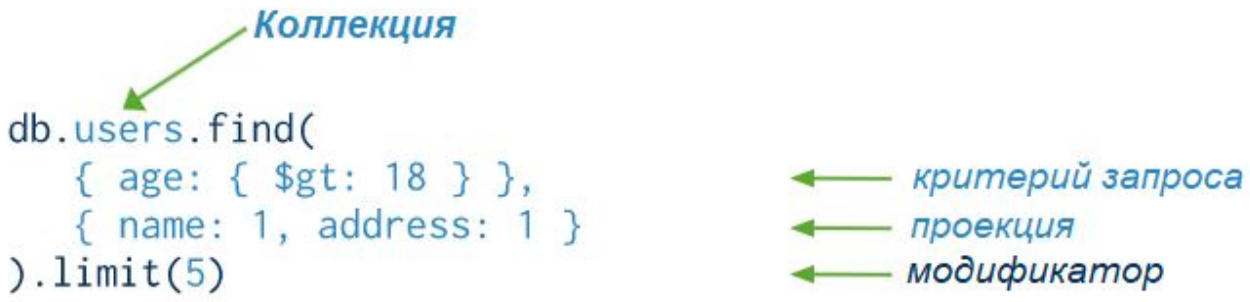
Операции Read

Операции Read запрашивают документы из коллекции по указанным критериям. MongoDB предоставляет всего один метод для того, чтобы получить необходимые документы:

- db.collection.find().

Вы можете также изменять результат работы метода с помощью критериев, проекций и дополнительных методов — модификаторов.

Общий синтаксис операции:



Запрос всех документов в коллекции

Чтобы получить все документы в коллекции, передайте пустой документ в качестве параметра метода **.find()**

```
db.inventory.find( {} )
```

Операция аналогична запросу в реляционной СУБД:

```
SELECT * FROM inventory
```

Запрос документов по заданному условию

Чтобы получить результат по определенному критерию поиска, необходимо передавать пары `{<field1>: <value1>, ... }` в качестве параметра метода .find().Так, следующий пример вернет все документы, значение поля “status” которых равны “D”:

```
db.inventory.find( { status: "D" } )
```

Операция аналогична запросу в реляционной СУБД:

```
SELECT * FROM inventory WHERE status = "D"
```

Запрос документов по нескольким критериям

Чтобы усложнить условия выборки из документов, используют специальные операторы запросов:

1. Операторы сравнения: $eq, $gt, $gte, $in, $lt, $lte, $ne, $nin.
2. Логические операторы: $and, $not, $nor, $or.
3. Операторы элементов: $exists, $type.
4. Операторы оценки: $expr, $jsonSchema, $mod, $regex, $text, $where.

Здесь перечислены основные. Про дополнительные можно прочитать в документации к MongoDB. Синтаксис применения операторов:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

Следующий пример запрашивает все документы из коллекции inventory, у которых поле status равно A или D:

```
db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

Важно!

Запрос можно составить с применением оператора ‘$or’. Результат будет тот же самый. Но когда мы проверяем значения в одном и том же поле у всех документов, рекомендуется использовать оператор $’in’ как более быстрый и правильный при решении этой задачи.

Операция аналогична запросу в реляционной СУБД:

```
SELECT * FROM inventory WHERE status in ("A", "D")
```

Запрос документов по составным условиям (оператор $and)

Вы можете выполнить составной запрос, используя критерии выборки по двум и более полям документов. Обычно для этого используется оператор $and (конъюнкция или логическое И) и дальше в списке указывается две и более пар <поле>:<значение>. Пример ниже запрашивает все документы в коллекции inventory со значениями status = “A” **и** qty меньше 30:

```
db.inventory.find( { $and: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

Но по умолчанию оператор будет применяться ко всем парам, если их просто указать в качестве параметра метода .find():

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

Результат выполнения обоих выражений будет одинаковый.

Операция аналогична запросу в реляционной СУБД:

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

Запрос документов по составным условиям (оператор $or)

Использование оператора $or (дизъюнкция или логическое «ИЛИ») позволяет объединять результаты запроса по каждой паре <поле>:<значение> из списка, переданного в качестве параметра метода

.find(). Чтобы запрос сработал, необходимо по крайней мере одно условие. Пример ниже запрашивает все документы в коллекции inventory со значениями status = “A” **или** qty меньше 30:

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

Операция аналогична запросу в реляционной СУБД:

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

Важно!

Запросы, в которых используются операторы сравнения, работают корректно, только если тип сравниваемого значения и тип значения поля документа одинаковы!

Запрос документов по составным условиям (совместное использование $or и $and)

В этом примере составной запрос возвращает все документы в коллекции, у которых значение поля status равно A **и** также qty меньше ($lt) 30 **или** item начинается с символа p:

```
db.inventory.find( {
     status: "A",
     $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
```

Операция аналогична запросу в реляционной СУБД:

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

Проекции

Проекция — метод, который определяет, какие поля, соответствующие критериям поиска вернутся в документах вернутся в документах. В качестве параметра передаётся документ вида:

```
{ field1: <value>, field2: <value> ... }
```

Значение <value> может быть следующим:

- 1 или true, чтобы включить поле в возвращаемый документ;
- 0 или false, чтобы исключить поле из возвращаемого документа;
- выражение, использующее операторы проекции: $, $elemMatch, $slice, $meta.

Важно!

Для поля _id не обязательно использовать проекцию _id: 1, чтобы включить поле _id. Метод .find() всегда будет возвращать это поле, пока вы не укажете принудительно в проекции _id: 0, чтобы его убрать в выводе.

Пример ниже запрашивает все документы в коллекции inventory со значениями status = A **и** qty меньше 30. Но выводит только поля item и status и _id (выводится по умолчанию):

```
db.inventory.find( { status: "A", qty: { $lt: 30 } }, {item: 1, status: 1} )
```

Операция аналогична запросу в реляционной СУБД:

```
SELECT item, status FROM inventory WHERE status = "A" AND qty < 30
```

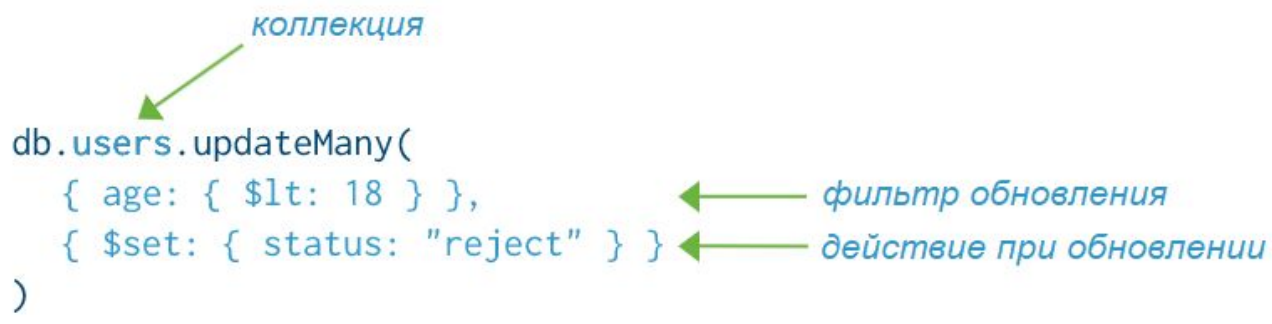
Операции Update

Операции Update изменяют документы целиком или значения их полей в коллекции по указанным критериям. MongoDB предоставляет три метода для данных целей:

- db.collection.updateOne();
- db.collection.updateMany();
- db.collection.replaceOne().

В MongoDB операции update всегда работают только с одной коллекцией. Нельзя обновить документ сразу в нескольких коллекциях, используя метод один раз. Также важно понимать, что все операции обновления в БД атомарны на уровне одного документа.

Общий синтаксис операции:



Обновление одного документа

Метод updateOne() находит в коллекции первый документ, соответствующий параметру <filter>, и применяет указанные в <update> изменения к нему. Его синтаксис:

```
db.collection.updateOne(
```

```
   <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>,
     collation: <document>,
     arrayFilters: [ <filterdocument1>, ... ],
     hint:  <document|string>        # Доступно, начиная с MongoDB версии 4.2.1
   }
)
```

Основные параметры метода:

Параметр	Описание
filter	Критерий выборки для обновления. Используются те же селекторы запроса, что и в методе find(). Передайте пустой документ {}, чтобы обновить первый документ в коллекции.
update	Модификации для изменения документа. Может содержать: ● операторы обновления документа. ● Пайплайн агрегации. Содержит следующие параметры обновления: : 1. $addFields и его алиас $set. 2. $project и его алиас $unset. 3. $replaceRoot и его алиас $replaceWith.
upsert	Опциональный параметр. В состоянии true метод updateOne() выполняет: ● создание нового документа, если в коллекции не найдено ни одного документа, соответствующего параметру filter. ● обновление одного документа, соответствующего параметру filter. Чтобы избежать множественного обновления, убедитесь, что параметр filter содержит поле с уникальным индексом. Значение по умолчанию: false.

Метод .UpdateOne() возвращает документ, который содержит:

- matchedCount — количество документов, которые сравнивались в процессе работы метода;
- modifiedCount — количество изменённых документов.
- upsertedId — содержит _id документов, прошедших через upsert.

Обновление нескольких документов

Метод updateMany() находит в коллекции документы, соответствующие параметру <filter>, и применяет указанные в <update> изменения к ним. Его синтаксис:

```
  <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>,
     collation: <document>,
     arrayFilters: [ <filterdocument1>, ... ],
     hint:  <document|string>        # Доступно, начиная с MongoDB версии 4.2.1
   }
)
```

Основные параметры метода:

Параметр	Описание
filter	Критерий выборки для обновления. Используются те же селекторы запроса, что и в методе find(). Передайте пустой документ {}, чтобы обновить все документы в коллекции.
update	Модификации для изменения документов. Может содержать: ● операторы обновления документа; ● Пайплайн агрегации. Содержит следующие параметры обновления: 1. $addFields и его алиас $set. 2. $project и его алиас $unset. 3. $replaceRoot и его алиас $replaceWith.
upsert	Опциональный параметр. Когда в состоянии true, метод updateMany() выполняет: ● создание нового документа, если в коллекции не найдено ни одного документа, соответствующего параметру filter; ● обновление всех документов, соответствующих параметру filter. Чтобы избежать множественного обновления, убедитесь, что параметр filter содержит поле с уникальным индексом. Значение по умолчанию: false.

Замена одного документа

Метод replaceOne() находит в коллекции первый документ, соответствующий параметру <filter>, и заменяет его на указанные в <replacement>. Его синтаксис:

```
db.collection.replaceOne(
   <filter>,
   <replacement>,
```

```
   {
     upsert: <boolean>,
     writeConcern: <document>,
     collation: <document>,
     hint: <document|string>          # Доступно, начиная с MongoDB версии
4.2.1
   }
)
```

Основные параметры метода:

Параметр	Описание
filter	Критерий выборки для обновления. Используются те же селекторы запроса, что и в методе find(). Передайте пустой документ {}, чтобы обновить первый документ в коллекции.
replacement	Документ для замены. Не может содержать операторы обновления.
upsert	Опциональный параметр. Когда в состоянии true, метод replaceOne() выполняет: ● создание нового документа, если в коллекции не найдено ни одного документа, соответствующего параметру replacement; ● обновление одного документа, соответствующего параметру replacement. Чтобы избежать множественного обновления, убедитесь, что параметр filter содержит поле с уникальным индексом. Значение по умолчанию: false.

Операции Delete

Операции Delete удаляют документ(ы) из коллекции. MongoDB предоставляет следующие методы, чтобы добавить новые документы в коллекцию:

- db.collection.deleteOne();
- db.collection.deleteMany().

В MongoDB операции delete всегда работают только с одной коллекцией. Нельзя удалить документ сразу в нескольких коллекциях, используя метод один раз. Также важно понимать, что все операции удаления в БД атомарны на уровне одного документа.

Можно задавать различные критерии или фильтры для удаления определённых документов в коллекции. Эти фильтры используют тот же синтаксис, что и операции чтения.

Удаление одного документа

Чтобы удалить только один документ, соответствующий заданному критерию (даже если таких документов несколько), необходимо использовать метод db.collection.deleteOne().

Пример удаляет первый документ со статусом D:

```
db.inventory.deleteOne( { status: "D" } )
```

Удаление нескольких документов

Чтобы удалить все документы из коллекции, необходимо передать пустой документ в качестве фильтра в метод db.collection.deleteMany().

```
db.inventory.deleteMany({})
```

Вы также можете задать критерий или фильтр, чтобы удалить только определенные документы из коллекции. Эти фильтры используют тот же синтаксис, что и операции чтения.

Чтобы определить условия для удаления, используйте пары <поле>:<значение> в параметре метода:

```
{ <field1>: <value1>, ... }
```

Также можно использовать операторы запроса, чтобы более сложно настроить выборку для удаления документов.

```
{ <field1>: { <operator1>: <value1> }, ... }
```

Чтобы удалить все документы, соответствующие критерию, необходимо задать фильтр в качестве параметра метода deleteMany().

Следующий пример удаляет все документы со статусом A.

```
db.inventory.deleteMany( { status: "A" } )
```

SQLAlchemy. Работа с SQLite

SQLite — это встраиваемая кроссплатформенная БД, которая поддерживает достаточно полный набор команд SQL и доступна в исходных кодах (на языке C).

Сервера у этой СУБД нет, сервер и есть приложение. Доступ к БД происходит через «подключения» к БД, которые мы открываем через вызов соответствующей функции DLL. При открытии указывается имя файла БД. Если такого нет — он автоматически создаётся.

Допустимо открывать множество подключений к одной и той же БД (через имя файла) в одном или разных приложениях.

Установка и подключение к БД

Происходит стандартным способом:

```
pip install sqlalchemy
```

Чтобы соединиться с СУБД, мы используем функцию create_engine():

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///users.db', echo=True)
```

Здесь users.db — имя базы данных, с которой будем работать. Флаг *echo* включает ведение лога через стандартный модуль logging Python.

Когда он включен, мы увидим все созданные нами SQL-запросы. Если вы хотите убрать отладочный вывод, то просто уберите его, поставив:

```
echo=False
```

Создание таблицы в базе данных

Далее мы пожелаем рассказать SQLAlchemy о наших таблицах. Мы начнем с одиночной таблицы *users*, в которой будем хранить записи о конечных пользователях, которые посещают некий сайт N. Мы определим таблицу внутри каталога *MetaData*, используя конструктор *Table()*, который похож на SQL-ный *CREATE TABLE:*

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey
metadata = MetaData()
users_table = Table('users', metadata,
      Column('id', Integer, primary_key=True),
```

```
      Column('name', String),
      Column('fullname', String),
      Column('password', String)
)
```

Далее мы пошлём базе *CREATE TABLE*, параметры которого будут взяты из метаданных нашей таблицы. Мы вызовем метод *create_all()* и передадим ему объект *engine*, который и указывает на базу. Там сначала будет проверено присутствие такой таблицы перед её созданием, так что можно выполнять это много раз — ничего страшного не случится.

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

Те, кто знаком с синтаксисом SQL и, в частности *CREATE TABLE*, могут заметить, что колонки *VARCHAR* создаются без указания их длины. В *SQLite* и *PostgreSQL* это вполне допустимый тип данных, но во многих других СУБД так делать нельзя. Чтобы выполнить этот урок в *MySQL*, длина должна быть передана строкам, как здесь:

```
Column('name', String(50))
```

Определение класса Python для отображения в таблицу

В то время, как класс *Table* хранит информацию о БД, он ничего не говорит о логике объектов, что используются нашим приложением. SQLAlchemy считает это отдельным вопросом. Для соответствия таблице users создадим элементарный класс *User*. Нужно только унаследоваться от базового класса *Object* (то есть у нас совершенно новый класс).

```
class User(object):
      def __init__(self, name, fullname, password):
      self.name = name
      self.fullname = fullname
      self.password = password

      def __repr__(self):
      return "<User('%s','%s', '%s')>" % (self.name, self.fullname,
self.password)
```

__init__ — это конструктор,

__repr__ же вызывается при операторе *print*.

Они определены здесь для удобства. Они не обязательны и могут иметь любую форму. SQLAlchemy не вызывает *__init__* напрямую.

Настройка отображения

Теперь мы хотим слить вместе таблицу user_table и класс User. Здесь нам пригодится пакет *SQLAlchemy ORM*. Мы применим функцию *mapper*, чтобы создать отображение между *users_table* и *User*.

```
from sqlalchemy.orm import mapper
mapper(User, users_table)
# <Mapper at 0x...; User>
```

Функция *mapper()* создаст новый *Mapper-объект* и сохранит его для дальнейшего применения, ассоциирующегося с нашим классом. Теперь создадим и проверим объект типа *User*:

```
from sqlalchemy.orm import mapper
print mapper(User, users_table)  #Передает класс User и нашу таблицу
user = User("Вася", "Василий", "qweasdzxc")
print user                 #Напечатает <User('Вася', 'Василий', 'qweasdzxc'>
print user.id              #Напечатает None
```

Атрибут *id*, который не определён в *__init__*, все равно существует из-за того, что колонка *id* существует в объекте таблицы *users_table*. Стандартно *mapper()* создает атрибуты класса для всех колонок, что есть в *Table*. Эти атрибуты существуют как Python-дескрипторы и определяют его функциональность. Она может быть очень богатой и включать в себе возможность отслеживать изменения и автоматически подгружать данные в базу, когда это необходимо.

Так как мы не сказали SQLAlchemy сохранить Василия в базу, его *id* выставлено на *None*. Когда позже мы сохраним его, в этом атрибуте будет храниться некое автоматически сформированное значение.

Декларативное создание таблицы, класса и отображения за один раз

Предыдущий подход к конфигурированию, включающий таблицу *Table*, пользовательский класс и вызов *mapper()*, иллюстрируют классический пример использования SQLAlchemy, в которой очень ценится разделение задач.

Большое число приложений, однако, не требуют такого разделения, и для них SQLAlchemy предоставляет альтернативный, более лаконичный стиль: декларативный.

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
class User(Base):
      __tablename__ = 'users'
      id = Column(Integer, primary_key=True)
      name = Column(String)
      fullname = Column(String)
      password = Column(String)

    def __init__(self, name, fullname, password):
      self.name = name
        self.fullname = fullname
      self.password = password
      def __repr__(self):
      return "<User('%s','%s', '%s')>" % (self.name, self.fullname,
self.password)

# Создание таблицы
```

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

Выше — функция *declarative_base()*, что определяет новый класс, который мы назвали *Base*, от которого будет унаследованы все наши ORM-классы.

Обратите внимание: мы определили объекты Column безо всякой строки имени, так как она будет выведена из имени своего атрибута.

Нижележащий объект *Table*, что создан нашей *declarative_base()* версией *User*, доступен через атрибут *__table__*

```
users_table = User.__table__
```

Метаданные MetaData также доступны:

```
metadata = Base.metadata
```

Создание сессии

Теперь мы готовы начать наше общение с базой данных. «Ручка» базы в нашем случае — это сессия *Session*. Когда сначала мы запускаем приложение, на том же уровне нашего *create_engine()* мы определяем класс *Session*, что служит фабрикой объектов *сессий* (*Session*).

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
```

В случае же, если наше приложение не имеет *Engine*-объекта базы данных, можно просто сделать так:

```
Session = sessionmaker()
```

А потом, когда вы создадите подключение к базе с помощью *create_engine()*, соедините его с сессией, используя *configure()*:

```
Session.configure(bind=engine)  # Как только у вас появится engine
```

Такой класс Session будет создавать Session-объекты, которые привязаны к нашей базе. Другие транзакционные параметры тоже можно определить вызовом sessionmaker()'а, но они будут описаны в следующей главе. Так, когда вам необходимо общение с базой, вы создаете объект класса Session:

```
session = Session()
```

Сессия здесь ассоциирована с SQLite, но у неё ещё нет открытых соединений с этой базой. При первом использовании она получает соединение из набора, который поддерживается engine, и удерживает его до тех пор, пока мы не применим все изменения и/или не закроем объект сессии.

Добавление новых объектов

Чтобы сохранить User-объект, нужно добавить его к сессии, вызвав add():

```
vasiaUser = User("vasia", "Vasiliy Pypkin", "vasia2000")
session.add(vasiaUser)
```

Этот объект будет находиться в ожидании сохранения, никакого SQL-запроса пока послано не будет. Сессия пошлет SQL-запрос, чтобы сохранить Васю, как только это понадобится, используя процесс сброса на диск(flush). Если мы запросим Васю из базы, то сначала вся ожидающая информация будет сброшена в базу, а запрос последует потом. Для примера, ниже мы создадим новый объект запроса (Query), который загружает User-объекты. Мы «отфильтровываем» по атрибуту «имя=Вася» и говорим, что нам нужен только первый результат из всего списка строк. Возвращается тот User, который равен добавленному:

```
ourUser = session.query(User).filter_by(name="vasia").first()
#<User('vasia','Vasiliy Pypkin', 'vasia2000')>
```

На самом деле сессия определила, что та запись из таблицы, что она вернула, та же самая, что она уже представляла во внутренней хеш-таблице объектов. Так что мы просто получили точно тот же самый объект, что добавили. Та концепция ORM, что работает здесь, известная как карта идентичности, обеспечивает, что все операции над конкретной записью внутри сессии оперируют одним и тем же набором данных. Как только объект с неким первичным ключом появится в сессии, все SQL-запросы на этой сессии вернут тот же самые объект Python для этого самого первичного ключа. Будет выдана ошибка в случае попытки поместить в эту сессию другой, уже сохранённый объект, с тем же первичным ключом. Мы можем добавить больше User-объектов, использовав add_all()

```
session.add_all([User("kolia", "Cool Kolian[S.A.]","kolia$$$"),
           User("zina", "Zina Korzina", "zk18")])   #добавить сразу пачку
записей
```

А вот тут Вася решил, что его старый пароль слишком простой, так что давайте его сменим:

```
vasiaUser.password = "-=VP2001=-"    #старый пароль был таки ненадежен
```

Сессия внимательно следит за нами. Она знает, что Вася был модифицирован:

```
session.dirty

#IdentitySet([<User('vasia','Vasiliy Pypkin', '-=VP2001=-')>])
```

И что ещё пара пользователей ожидают сброса в базу:

```
session.dirty

#IdentitySet([<User('vasia','Vasiliy Pypkin', '-=VP2001=-')>])
```

Сохранение в базу

А теперь мы скажем сессии, что мы хотим отправить все оставшиеся изменения в базу и применить их, зафиксировав транзакцию, которая до того была в процессе. Это делается с помощью commit():

```
session.commit()
```

commit() сбрасывает все оставшиеся изменеия в базу и фиксирует транзакции. Ресурсы подключений, что использовались в сессии, снова освобождаются и возвращаются в набор. Последовательные операции с сессией произойдут в новой транзакции, которая снова запросит себе ресурсов по первому требованию. Если посмотреть васин атрибут id, что раньше был None, то мы увидим, что ему присвоено значение:

```
vasiaUser.id
#1
```

После того, как сессия вставит новые записи в базу, все только что созданные идентификаторы будут доступны в объекте, немедленно или по первому требованию. В нашем случае, целая запись была перезагружена при доступе, так как после вызова commit() началась новая транзакция. SQLAlchemy стандартно обновляет данные от предыдущей транзакции при первом обращении с новой транзакцией, так что нам доступно самое последнее её состояние.

Практическое задание

1. Развернуть у себя на компьютере/виртуальной машине/хостинге MongoDB и реализовать функцию, записывающую собранные вакансии в созданную БД.
2. Написать функцию, которая производит поиск и выводит на экран вакансии с заработной платой больше введённой суммы.
3. Написать функцию, которая будет добавлять в вашу базу данных только новые вакансии с сайта.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Информация по MongoDB.
2. Сравнение MySQL и MongoDB.
3. Документация по PyMongo.
4. Документация по MongoDB.
5. Туториал с примерами по работе с SQLite в SQLAlchemy.
6. Документация SQLAlchemy.