NoSQL, MongoDB

NoSQL (Not only SQL — «не только SQL») — это семейство нереляционных систем управления базами данных, которые хранят данные в моделях, отличных от традиционных таблиц, например, в виде документов, пар «ключ-значение», колонок или графов. Такие системы появились как решение проблем масштабируемости и гибкости, присущих реляционным базам данных при работе с большими объемами неструктурированных или постоянно меняющихся данных.

Основные модели данных NoSQL

- **Документоориентированные:** Данные хранятся в виде документов (например, в формате JSON или BSON) и могут иметь различную структуру. Это позволяет хранить связанную информацию в одном месте, как это делается в базах данных MongoDB и Couchbase.
- **Ключ-значение:** Каждая запись имеет уникальный ключ и соответствующее значение. Эта модель подходит для быстрого доступа к данным, например, для кеширования.
- **Колоночные (столбцовые):** Данные организованы по столбцам, что обеспечивает высокую производительность при запросах, выбирающих только необходимые поля из больших наборов данных. Применяется в Apache Cassandra и Google Cloud Bigtable, Redis.
- **Графовые:** Используют узлы, ребра и свойства для моделирования связей между данными. Эта модель эффективна для анализа взаимосвязей, например, в социальных сетях или рекомендательных системах. Примеры — Neo4j и Amazon Neptune.

Ключевые особенности NoSQL

- **Гибкая схема:** В отличие от реляционных БД, NoSQL не требует заранее определенной структуры, что позволяет легко добавлять новые поля к данным.
- **Горизонтальное масштабирование:** Системы NoSQL разработаны для масштабирования путем добавления новых серверов, что делает их подходящими для работы с большими объемами данных (Big Data).
- **Разнообразие моделей:** Отсутствие единого стандарта SQL означает существование множества различных моделей и подходов к хранению данных, каждый из которых оптимизирован под определенные задачи.

Основное отличие концепций NoSQL и SQL баз данных

Этим отличием являются их модели - ACID и BASE.
ACID и BASE - то модели, определяющие правила, по которым разные сервера/ компьютеры одной системы достигают того, чтобы в итоге данные в базе этой системы, были одинаковые в один и тот же момент времени.

ACID	BASE
Atomicity (Атомарность) Операция делается до конца вся или не делается вовсе.	**B**asically **A**vailable (Базово доступна) Система всегда отвечает на запросы, даже если некоторые узлы недоступны.
Consistency (Согласованность) Данные становятся одинаковые на всех серверах системы немедленно.	**S**oft state (Мягкое состояние) Данные могут быть временно несогласованными.
Isolation (Изолированность) Операции не мешают друг другу. (Тут можно вспомнить про 4 уровня изоляции)	**E**eventually consistent (В конечном счете согласованная) Если не поступает новых обновлений, то со временем все узлы придут к одинаковому состоянию.
Durability (Долговечность) После завершения операции данные сохраняются и не потеряются.	

MongoDB

MongoDB - документноориентированная система управления базами данных, которая хранит данные в виде гибких документов, похожих на *JSON (в формате BSON).*

JSON - JavaScript Object Notation - это текстовый формат документа для хранения и передачи структурированных данных, который легко читается любым программистом. Используются пары "ключ: значения". У значений всего 6 типов данных.

BSON - Binary JSON - расширенный формат JSON(двоичный), в который добавляются множество типов данных. Нечитаемый человеком, но компьютер работает с ним быстрее, чем с JSON. Используется для хранения данных, создан специально для MongoDB.

Структура JSON

1. Стандартный

```
{
"ключ1": "значение1",
"ключ2": "значение2",
"ключ3": значение3
}
```

Ключ - это аналог столбца в реляционной базе данных,
а значение - ячейка таблицы.
Весь документ - это одна строка.

2. Вложенная структура

```
{
"ключ1": "значение1",
"ключ2": {
"ключ2.1": "значение2.1",
"ключ2.2": "значение2.2",
"ключ2.3": "значение2.3"
}
}
```

Ключу 2 соответствует значение 2

- всё что внутри фигурных скобок.
- Это тесно связанные данные, они обновляются, вставляются, удаляются вместе. "Документ в документе"

2. Массив объектов

```
{
"ключ1": [
{
"ключ1.1.1": "значение1.1.1",
"ключ1.1.2": "значение1.1.2",
"ключ1.1.3": "значение1.1.3"
},
{
"ключ1.2.1": "значение1.2.1",
"ключ1.2.2": "значение1.2.2",
"ключ1.2.3": "значение1.2.3"
}
]
}
```

Массивы используются, когда одному ключу соответствует
несколько значений
либо несколько вложенных документов. Заключаются в квадратные скобки.

В MongoDB всё что в фигурных скобках - это один документ, который может быть прочитан и обновлён независимо. И несколько таких документов сами по себе не образуют классическую таблицу, как в реляционной базе данных. Эти документы

можно объединять в коллекции. Для этого в JSON файле нужно объединить строки в контейнер, при помощи квадратных скобок в самом начале файла и в самом конце.

MongoDB используется преимущественно через языки программирования.

- Python
- JavaScript
- Java
- C#
- Go

pymongo — это официальная библиотека Python для работы с MongoDB. Она преобразует JSON файлы в BSON и наоборот.

Синтаксис pymongo

Подключение:

- `from pymongo import MongoClient` - *Импорт библиотеки*
- `client = MongoClient("mongodb://localhost:27017/")` - *Подключение к локальной MongoDB*
- `db = client["my_database"]` - *Выбор базы данных*
- `collection = db["user_events"]` - *Выбор коллекции*

Вставка:

- `collection.insert_one({"ключ" : "значение".....})` - *вставка одного документа в коллекцию*
- `collection.insert_many([{"ключ" : "значение".....}, {"ключ" : "значение".....}])` - *вставка нескольких документов в коллекцию*

Чтение, поиск:

- `n = collection.find_one({"ключ": "значение"})` - *найти один такой документ*
- `n = collection.find({"ключ": "значение"})` - *найти все такие документы*
- Есть ещё поиск с условием, который работает через collection.find. Операторы будут ниже.
- Т.к. поиск должен вернуть значения, а не просто выполнить операцию, назначается переменная.

Обновление:

- Обновить один документ
  `collection.update_one(`
  `{"ключ": "значение"},`
  `{"$set": {"ключ": "значение"}}`
  `)`
  Вторя строка - это фильтр, как WHERE в PostgreSQL. Где обновить?. Третья - что обновить.
- Обновить несколько документов
  `collection.update_many(`
  `{"ключ": "значение"},`
  `{"$set": {"ключ": "значение"}}`
  `)`

Удаление:

- `collection.delete_one({"ключ": "значение"})` - *удаление одного документа*
- `collection.delete_many({"ключ": "значение"})` - *удаление нескольких документов*

Операторы:

- `"$gt"` - *больше чем*
- `"$lt"` - *меньше чем*
- `"$gte"` - *больше или равно*
- `"$lte"` - *меньше или равно*
- `"$and"` - *и*
- `"$or"` - *или*
  Первые четыре используются в значении. Например , `{"ключ":{"$gt":"значение"}}` - "ключ со значением больше, чем это значение".
  Последние два пишутся перед массивом документов. Например, `{"$or": [{"ключ": {"$lt": "значение"}}, {"ключ": {"$gt": "значение"}}]}` - "ключ со значением меньше чем это значение ИЛИ больше чем это значение".

Отдельно нужно разобрать агрегацию:
В ней выделяют несколько "стадий". Они содержат параметры, по которым проходит выборка. Стадии могут быть в том порядке, в каком задумал сам пользователь для корректной работы. Тут нет строгого порядка, как в планировщике SQL.

Общая структура команды:

```
`n = [`
    `"$match":{}`
```

Тут прописывает "где", можно использовать операторы, для уточнения условий. WHERE.

```
    `"$group":{}`
```

Тут группировка, как GROUP BY. Так же должна быть агрегатная функция.

```
`"$sort"{"ключ": -1}`
```

Это ORDER BY. Указываем поле сортировки. Если пишем -1 - значит будет в порядке убывания, просто 1 - в порядке возрастания

```
`"$project":{}`
```

Выбор полей. Select

```
`]`
`results = collection.aggregate(n)`
```

Назначаем переменную результату агрегации.

Стадия $group основная для всей операции, поэтому имеет более подробное описание параметров, чем остальные:

`"$group": {`
`"_id": "$название поля",` - по чему будем группировать
`"название нового поля1": { "аккумулятор1": выражение },`
`"название нового поля2": { "аккумулятор2": выражение }`
}

Группировать можно:

- По одному полю, как в примере
- По нескольким полям
- По одному полю, но с фильтрацией при помощи операторов
- Указать одну группу для всех, написав null в названии поля

Аккумуляторы - это функции, которые накапливают одно значение на группу и работают только внутри стадии $group. Они преобразуют сырые данные в готовые.
Существуют:

- `$sum` — суммирование
- `$avg` — среднее значение

- `$min` / `$max` — экстремумы
- `$first` / `$last` — первое/последнее значение
- `$push` / `$addToSet` — добавляет ВСЕ значения в массив (с дубликатами)/добавляет УНИКАЛЬНЫЕ значения в массив
- `$mergeObjects` — объединение объектов
- `$stdDevPop` / `$stdDevSamp` — стандартное отклонение

Т.к. аккумулятор-функция, то далее ему назначается параметр - это выражение.
Выражения могут быть использованы, как и операторы, на других стадиях.
Параметры есть специальные, которых большое множество, тут некоторые:

- `$$ROOT` — весь исходный документ
- `$$CURRENT` — текущий документ
- `$$REMOVE` — исключить документ из результатов
- `$reduce` — сворачивание массива
- `$map` — преобразование массива
- `$cond` — условный оператор
- `$switch` — множественные условия
- `$dateToString` — форматирование дат
- `$dateFromParts` — создание даты
- `$concat` - конкатенация
- `$indexOfArray` - поиск в массиве
- `$slice` — часть массива, срез.

А есть простой параметр - пути к полю(Field Path Expression), при котором возвращается значение из поля.

```
{
"$group": {
"_id": "$user_id",
"events": {"$push": "$название поля"}
}
}
```

Работа с датами

JSON не поддерживает специальные типы данных для дат. Даты в нём хранятся в виде строк. Для корректной работы БД необходима библиотека **datetime**. Она позволяет получать текущие дату и время, форматировать их в различные строки, а также выполнять арифметические операции с временными промежутками, используя класс `timedelta`.

Базовый синтаксис библиотеки:

- *Подключение:*
  `from datetime import datetime`
- *Текущая дата и время:*
  `now = datetime.now() # 2024-01-15 14:30:25.123456`
- *Конкретная дата:*
  `specific_date = datetime(2024, 1, 15) # 2024-01-15 00:00:00`
  `specific_datetime = datetime(2024, 1, 15, 10, 30, 0) # 2024-01-15 10:30:00`
- *Получение компонентов даты:*
  `now = datetime.now()`
  `print(now.year) # 2024`
  `print(now.month) # 1`
  `print(now.day) # 15`
  `print(now.hour) # 14`
  `print(now.minute) # 30`
- *Работа с промежутками времени(timedelta):*
  `thirty_days = timedelta(days=30)`
  `fourteen_days = timedelta(days=14)`
  `two_weeks = timedelta(weeks=2)`
  `three_hours = timedelta(hours=3)`
- *Арифметика с датами:*
  `today = datetime.now()`

`thirty_days_ago = today - timedelta(days=30)`
`two_weeks_ago = today - timedelta(weeks=2)`

`next_week = today + timedelta(days=7)`
`in_three_hours = today + timedelta(hours=3)`

- *Форматирование дат в строки:*
  `today = datetime.now()`

`iso_date = today.strftime("%Y-%m-%d") # "2024-01-15"`
`human_date = today.strftime("%d.%m.%Y") # "15.01.2024"`
`full_date = today.strftime("%Y-%m-%d %H:%M:%S") # "2024-01-15 14:30:25"`
`time_only = today.strftime("%H:%M") # "14:30"`

```
  Популярные спецификаторы:
   -  `%Y` - год (4 цифры) `2024`
   - `%m` - месяц (01-12) `01`
   - `%d` - день (01-31) `15`
   - `%H` - час (00-23) `14`
   - `%M` - минуты (00-59) `30`
   - `%S` - секунды (00-59) `25`
```

- *Парсинг строк в даты:*
  `date_str = "2024-01-15"`
  `parsed_date = datetime.strptime(date_str, "%Y-%m-%d")`
  `time_str = "15.01.2024 14:30"`
  `parsed_datetime = datetime.strptime(time_str, "%d.%m.%Y %H:%M")`

```
  `strptime()` создаёт объект `datetime`:
     datetime.datetime(2024, 1, 15,    0,     0)
                  |    |    |   |      |      |
                  |   год  мес день   час   минута
                  |
            класс "дата-время"

      Можно обращаться к компонентам:
          `print(parsed.year)        # 2024`
          `print(parsed.month)       # 1`
          `print(parsed.day)         # 15`
          `print(parsed.hour)        # 0 (по умолчанию)`
         `print(parsed.minute)      # 0 (по умолчанию)`
```

Работа с JSON в Python.

Аналогично с датами, для для преобразования данных между Python-объектами и JSON-форматом необходима библиотека json. Она позволяет сохранять Python-структуры (словари, списки) в JSON-файлы и читать их обратно, обеспечивая совместимость с другими системами и удобное хранение данных.

Базовый синтаксис:

- *Подключение:*
  `import json`
- *Запись в файл:*
  `data = {"ключ": "значение", "число": 123}`
  `with open("файл.json", "w", encoding="utf-8") as f: # Python-объект → JSON-файл json.dump(data, f, indent=2, ensure_ascii=False)`
- *Чтение из файла:
  `with open("файл.json", "r", encoding="utf-8") as f: data = json.load(f)` # JSON-файл → Python-объект
- *Преобразование в строку:*
  `json_string = json.dumps(data, indent=2)`

  `data = json.loads(json_string)`

Библиотека `os`:

Предназначена для взаимодействия с операционной системой, предоставляя функции для работы с файлами и каталогами, управления процессами, доступа к переменным окружения и выполнения команд оболочки. Она нужна для того, чтобы делать код **переносимым** между разными ОС (например, Windows, macOS, Linux), обеспечивая единообразный способ выполнения системных задач, таких как создание или удаление папок, переименование файлов или получение списка файлов в директории.
Если прям коротко - она нужна для работы с файлами и путями.

- *Подключение:*
  `import os`
- *Основные функции:*
  - `os.path.exists("путь")` — проверяет существование файла/папки
  - `os.makedirs("папка")` — создаёт папку (если нет)
  - `os.path.join("папка", "файл.json")` — создаёт корректный путь
  - `os.getcwd()` — возвращает текущую директорию