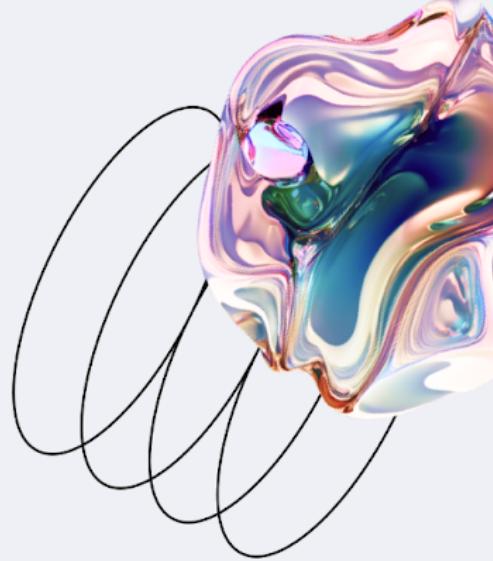


Системы управления базами данных MongoDB и ClickHouse в Python

Сбор и разметка данных



Оглавление

Словарь терминов	3
Введение	3
Введение в MongoDB	3
Подключение к MongoDB	7
Локальное подключение	7
Подключение в облаке	11
Создание баз данных и коллекций MongoDB в Python	17
Выполнение операций CRUD в MongoDB	22
Создание запросов в MongoDB	23
Операторы MongoDB	24
Обновление данных в коллекции MongoDB	28
Основы работы в ClickHouse	30
Работа с ClickHouse в Python	40
Заключение	48
Что можно почитать еще?	50

Словарь терминов

Ассоциативный массив — абстрактный тип данных (интерфейс к хранилищу данных), который позволяет хранить пары вида «ключ-значение» и поддерживает операции добавления пары, а также поиска и удаления пары по ключу.

BSON (Binary JavaScript Object Notation) — формат электронного обмена цифровыми данными, бинарная форма представления простых структур данных и ассоциативных массивов (которые в контексте обмена называют объектами или документами).

Введение

В течение предыдущих двух уроков мы познакомились с двумя принципиально разными способами получения данных — с помощью API и путем скрейпинга HTML. Сегодня перейдем к работе с базами данных: поговорим как о хранении, так и об извлечении данных. Ранее вы уже знакомились с системами управления реляционными базами данных и языком структурированных запросов SQL, однако у того, о чём речь пойдет сегодня, есть существенные отличия. Мы будем говорить о системах управления базами данных (СУБД) MongoDB и ClickHouse.

MongoDB — одна из самых популярных, интересных и простых в изучении баз данных. Мы начнем с небольшого введения в MongoDB, затем рассмотрим способы подключения к базе данных локально или в облаке, узнаем, как получить доступ к базе данных в Python, и разберем самые важные функции.

ClickHouse — столбцовая СУБД для онлайн-обработки аналитических запросов (OLAP).

Введение в MongoDB

MongoDB широко используется в больших данных. Это ведущая NoSQL (Not only SQL) система управления базами данных.

MongoDB — документоориентированная база данных (document based). Под определением «документоориентированная» мы не подразумеваем базу данных для хранения документов PDF или Microsoft Word. Под документами мы понимаем **ассоциативный массив**. Это тот же тип структуры, который вы видите в объекте JSON или в словаре Python. MongoDB позволяет хранить эти типы иерархических

структур данных непосредственно в базе данных в виде отдельных элементов или документов.

Документы, хранящиеся в MongoDB, используют JSON-подобный синтаксис.

Посмотрим на примере. Это информационное окно игры «Ведьмак 3» в Steam:

Вы — Геральт из Ривии, наемный убийца чудовищ. Вы путешествуете по миру, в котором бушует война и на каждом шагу подстерегают чудовища. Вам предстоит выполнить заказ и найти Цири — Дитя Предназначения, живое оружие, способное изменить облик этого мира.

ВСЕ ОБЗОРЫ: Крайне положительные (545,650) *

ДАТА ВЫХОДА: 18 мая. 2015

РАЗРАБОТЧИК: CD PROJEKT RED
ИЗДАТЕЛЬ: CD PROJEKT RED

Популярные метки для этого продукта:

Открытый мир Ролевая игра Глубокий сюжет +

В информационном блоке есть описание игры, количество обзоров, дата выхода, разработчик и так далее. Обратите внимание, что некоторые поля содержат несколько разных значений, например, «Популярные метки» или таблицы с поддерживаемыми языками.

Языки:			
	Интерфейс	Озвучка	Субтитры
русский	✓	✓	✓
английский	✓	✓	✓
французский	✓	✓	✓
итальянский	✓		✓
немецкий	✓	✓	✓

[Просмотреть все поддерживаемые языки \(16\)](#)

Разберемся, как мы можем смоделировать эти данные в MongoDB в виде документов, используя синтаксис JSON.

Начнем с полей с числовыми значениями. У нас есть числовые значения в поле id приложения (в строке HTTP-запроса), количество положительных и отрицательных отзывов. Документ очень похож на JSON-файл. Поскольку вы уже знакомы с форматом JSON, MongoDB должен быть очень простым для вас.

```
{  
    "appid": 292030,  
    "positive": 632627,  
    "negative": 25245,  
}
```

Затем текстовые поля — название, жанр, разработчик, паблишер, дата выхода.

```
{  
    "appid": 292030,  
    "positive": 632627,  
    "negative": 25245,  
  
    "name" : "The Witcher 3: Wild Hunt",  
    "developer" : "CD PROJEKT RED",  
    "publisher" : "CD PROJEKT RED",  
    "genre" : "RPG",  
    "release_date" : "2015/05/18",  
}
```

В информационном окне игры есть теги продукта. Через Steam API можно получить количество пользователей, отметивших теги. Добавим эту информацию в наш документ.

```
{  
    "appid": 292030,  
    "positive": 632627,  
    "negative": 25245,  
  
    "name" : "The Witcher 3: Wild Hunt",  
    "developer" : "CD PROJEKT RED",  
    "publisher" : "CD PROJEKT RED",  
    "genre" : "RPG",  
    "release_date" : "2015/05/18",  
}
```

```
    "tags" : {  
        "Open World" : 11677,  
        "RPG" : 10024,  
        "Story Rich" : 9219,  
        "Atmospheric" : 6478,  
        "Mature" : 6234,  
        "Fantasy" : 6057  
    }  
}
```

В документах MongoDB важно то, что можно сделать вложенный документ, то есть массив внутри массива. Из-за этого MongoDB действительно гибкий: мы можем поместить массив внутрь всего документа, нам не нужно думать о том, как это структурировать, или заботиться о типе данных, в отличие от реляционных баз данных, где информацию нужно разместить в таблицу.

На самом деле, данные не хранятся в формате JSON, они хранятся в формате BSON. Он двоичный, поэтому работает очень быстро. Что действительно здорово — мы также можем запросить эти вложенные данные. Разберем это позже.

Почему MongoDB так популярна? Одна из причин — ее гибкая схема. Это позволяет легче обрабатывать данные в иерархических форматах, где отдельные поля состоят из нескольких элементов. Также MongoDB ориентирована на программистов: JSON-документы, хранящиеся в MongoDB, содержат типы данных, встречающиеся в большинстве популярных языков программирования. Например, массивы, словари Python и так далее.

Также MongoDB имеет клиентские библиотеки (иначе они называются драйверами) для большинства популярных языков программирования. Эти драйверы выполняют работу по переводу данных в разные типы данных и обратно в рамках конкретного языка в соответствии с требованиями приложения, а также поддерживают ряд других функций, которые упрощают разработку.

Еще одна особенность, которая делает MongoDB отличным инструментом для специалистов по анализу данных, — возможность развертывания разными способами. Всего за пару минут можно загрузить, установить, запустить MongoDB на ноутбуке и быстро начать разработку приложения. Также можно развернуть MongoDB на нескольких серверах для работы с большими данными.

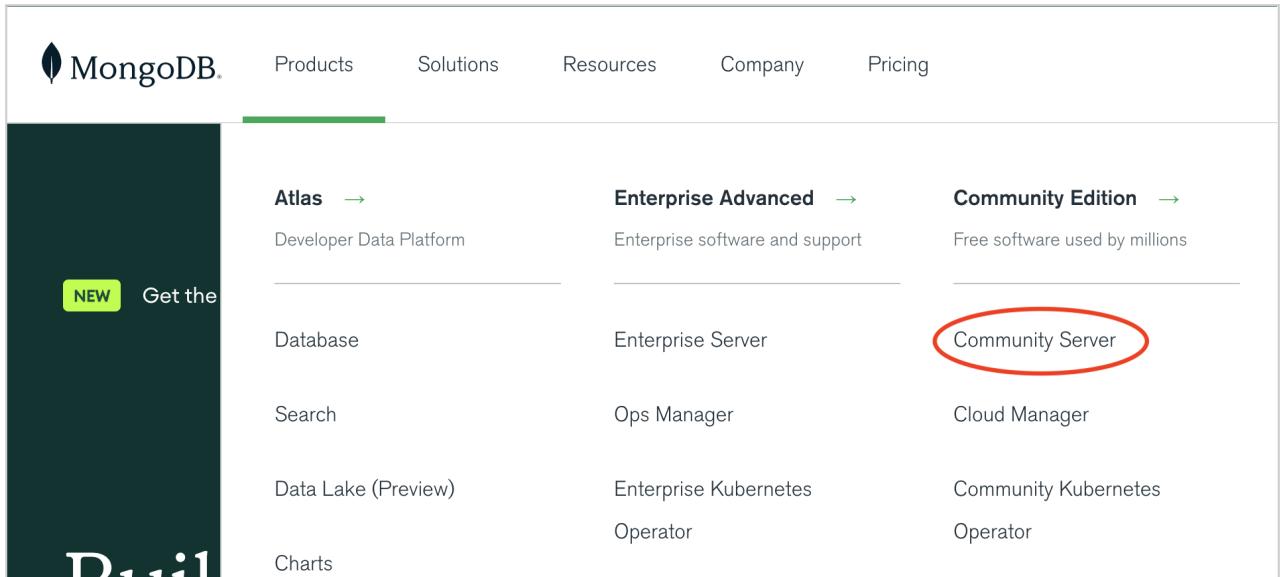
MongoDB разработана для больших данных. Она хорошо масштабируется по горизонтали, на аппаратном обеспечении, и включает поддержку Map Reduce, а также позволяет создавать эффективные аналитические приложения.

Подключение к MongoDB

Давайте посмотрим, как подключаться к базе данных MongoDB двумя способами: локально и в облаке.

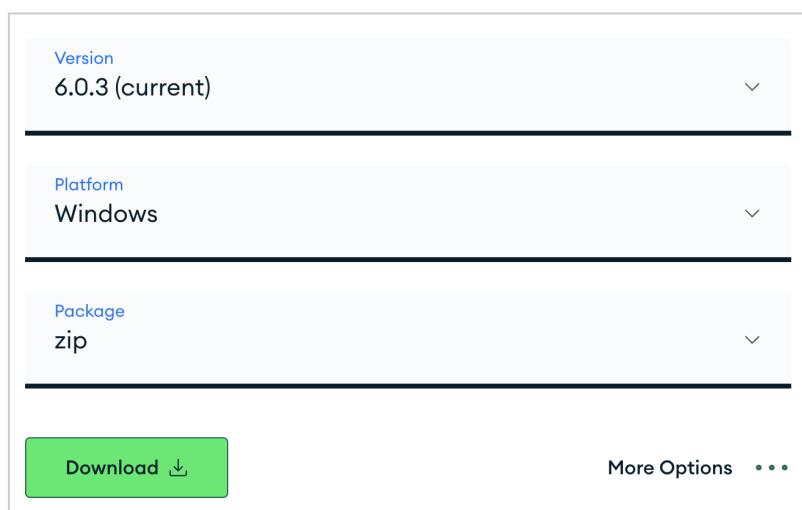
Локальное подключение

Переходим на официальный сайт mongodb.com, в меню выбираем Products → Community Server.



The screenshot shows the MongoDB homepage with a dark sidebar on the left containing a 'NEW' button and the text 'Get the Data'. The main menu includes 'Products', 'Solutions', 'Resources', 'Company', and 'Pricing'. Below the menu, there are three main categories: 'Atlas' (Developer Data Platform), 'Enterprise Advanced' (Enterprise software and support), and 'Community Edition' (Free software used by millions). The 'Community Edition' section is circled in red. It contains sub-options for 'Database', 'Search', 'Data Lake (Preview)', and 'Charts', each with corresponding links to 'Enterprise Server', 'Ops Manager', 'Enterprise Kubernetes', and 'Operator' respectively.

В открывшемся разделе сайта выбираем версию и нужную платформу: например, Windows. Жмем Download.

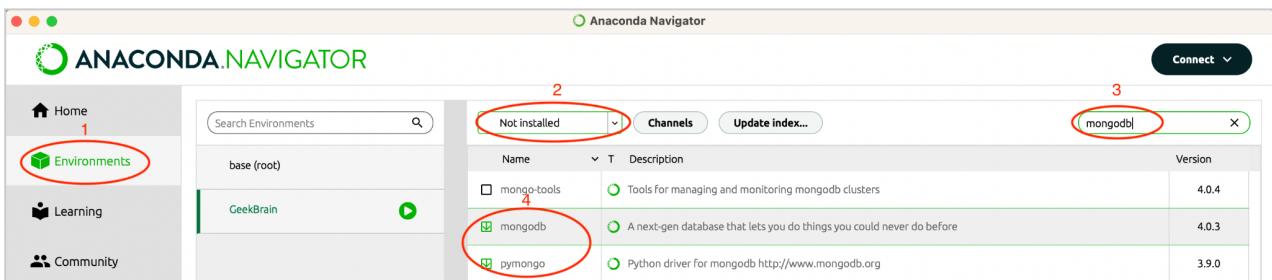


The screenshot shows a dropdown menu for selecting a MongoDB version. The current selection is '6.0.3 (current)'. Below it, another dropdown menu is set to 'Windows'. A third dropdown menu is set to 'zip'. At the bottom of the interface are two buttons: a green 'Download' button with a downward arrow icon, and a 'More Options' button with three dots.

Далее следуем инструкции по установке в зависимости от вашей системы: [Install MongoDB Community Edition](#).

Другая опция – установка в Anaconda Navigator:

1. Заходим в Environments.
2. Выбираем в меню Not installed.
3. В поисковой строке пишем «mongodb».
4. Выбираем mongodb и pymongo.
5. Нажимаем Apply в правом нижнем углу окна.



Вы можете запустить MongoDB в терминале как службу macOS с помощью brew или запустить MongoDB вручную как фоновый процесс:

```
brew services start mongodb-community@6.0
```

```
petrrubin — -zsh — 114x24
(GeekBrain) petrrubin@MacBook-Pro-Petr ~ % brew services start mongodb-community@6.0
==> Successfully started `mongodb-community` (label: homebrew.mxcl.mongodb-community)
(GeekBrain) petrrubin@MacBook-Pro-Petr ~ %
```

Запустить MongoDB можно командой **mongo**. Теперь можно взаимодействовать с базой данных.

```
petrrubin — -zsh — 114x24
(GeekBrain) petrrubin@MacBook-Pro-Petr ~ % mongo
```

Кроме прочего, можно использовать и графический интерфейс. Для этого нужно установить программу Compass, которую вы найдете на домашней странице в разделе Products → Compass.

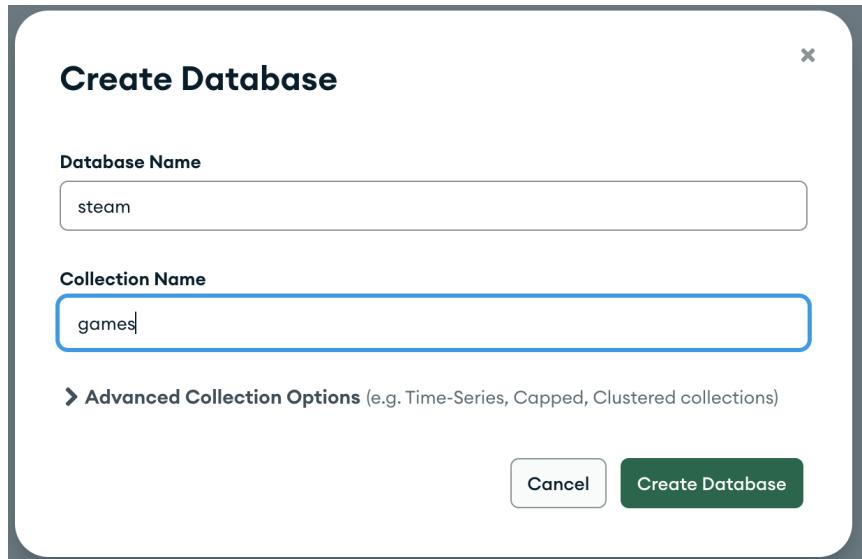
The screenshot shows the official MongoDB website. At the top, there's a navigation bar with links for Products, Solutions, Resources, Company, Pricing, a search icon, and Sign In. Below the navigation, there's a dark sidebar on the left with a 'NEW' badge and the text 'Get the'. The main content area is divided into four columns: 'Atlas → Developer Data Platform', 'Enterprise Advanced → Enterprise software and support', 'Community Edition → Free software used by millions', and 'Tools → Build faster'. A red circle highlights the 'Compass →' link under the Tools section.

После установки в графическом интерфейсе мы можем нажать Connect. Здесь по умолчанию есть три базы данных: admin, config и local.

The screenshot shows the MongoDB Compass application running on localhost:27017. The interface has tabs for My Queries, Databases (which is selected), and Performance. Under the Databases tab, there are three database entries: 'admin', 'config', and 'local'. Each entry provides storage statistics: admin (20.48 kB), config (20.48 kB), and local (36.86 kB). Each entry also shows the number of collections (1 for admin and config, 1 for local) and indexes (1 for admin and config, 1 for local).

Database	Storage size:	Collections:	Indexes:
admin	20.48 kB	1	1
config	20.48 kB	1	2
local	36.86 kB	1	1

Здесь же можно создать новую базу данных, нажав Create database. Далее введем название базы данных (например, steam) и название коллекции (например, games).



Нажмем на Create Database. После этого база данных появится в списке. Теперь можем войти в нее, просто кликнув на названии. Увидим только что созданную коллекцию games.

MongoDB Compass - localhost:27017

Collections

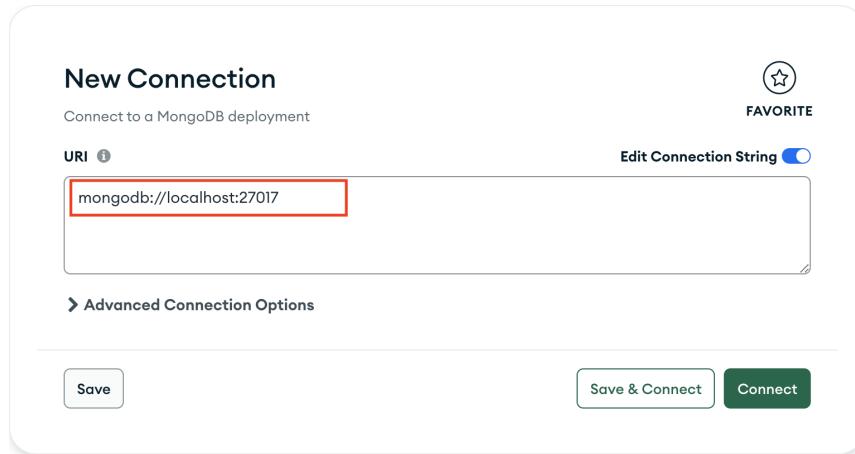
Create collection View

games

Storage size: 4.10 kB Documents: 0

Если зайдем в коллекцию games, сможем добавлять данные. Сделаем это чуть позже.

Чтобы подключиться к базе данных, в верхнем меню выбираем Connect → New Connection. Отсюда нужно сохранить строку подключения (Connection String), она нам понадобится в коде. Нажимаем Connect.



Подключение в облаке

Перейдем к другому варианту подключения к MongoDB — в облаке. Заходим на сайт MongoDB, затем Products → Atlas.

Atlas — это облачная система управления базами данных, где вся инфраструктура автоматизирована. Есть разные варианты использования, в том числе бесплатный.

Для начала нужно зарегистрироваться. Заполняем форму регистрации или заходим с помощью Google-аккаунта.

Далее нужно выбрать тип базы данных, которую мы хотим создать.

Welcome to MongoDB Atlas! We've recommended some of our most popular options, but feel free to customize your cluster to your needs. For more information, check our [documentation](#).

Serverless Dedicated **FREE Shared**

Мы можем выбрать между Serverless, Dedicated и Shared. Serverless и Dedicated предназначены для производственных приложений с расширенными потребностями в конфигурации. Мы выбираем бесплатную версию — Shared. Она дает хранилище на 512 МБ и предназначена для обучения MongoDB.

Далее мы можем выбрать облачного провайдера и регион, в котором хотим развернуть наш кластер.

Cloud Provider & Region

AWS, Bahrain (me-south-1)

★ Recommended region ⓘ ⓘ Dedicated tier region ⓘ

NORTH AMERICA

- 🇺🇸 N. Virginia (us-east-1) ★
- 🇺🇸 Oregon (us-west-2) ★
- 🇺🇸 Ohio (us-east-2) ⓘ
- 🇺🇸 N. California (us-west-1) ⓘ

EUROPE

- 🇸🇪 Stockholm (eu-north-1) ★
- 🇩🇪 Frankfurt (eu-central-1) ★
- 🇮🇪 Ireland (eu-west-1) ★
- 🇫🇷 Paris (eu-west-3) ★

AUSTRALIA

- 🇦🇺 Sydney (ap-southeast-2) ★

ASIA

- 🇨🇳 Hong Kong (ap-east-1) ★
- 🇸🇬 Singapore (ap-southeast-1) ★

Мы также можем включить резервное копирование, если перейдем на кластер не ниже M2 (эта опция платная). И, наконец, можем выбрать имя нашего кластера. Просто сохраним все значения по умолчанию и нажмем кнопку Create Cluster.

Теперь, когда кластер создан, можно зайти в него

Database Deployments

Find a database deployment...

Load sample datasets to Cluster0.
Atlas provides sample data you can load into your Atlas clusters. You can also import your own data in MongoDB.

Cluster0 Connect View Monitoring Browse Collections ...

Enhance Your Experience
For production throughput and richer metrics, upgrade to a dedicated cluster now!

Upgrade

R	0
Last 6 hours	100.0/s

W	0
Last 6 hours	100.0/s

И добавить образец датасета — Load Sample Dataset (загрузка может занять несколько минут).

The screenshot shows the MongoDB Atlas Cluster0 Overview page. At the top, it displays the version (5.0.14), region (AWS Bahrain (me-south-1)), and cluster tier (M0 Sandbox (General)). Below this, there are tabs for Overview, Real Time, Metrics, Collections, Search, Profiler, Performance Advisor, Online Archive, and Cmd Line Tools. Under the Overview tab, there are buttons for SANDBOX, NODES, and REPLICA SET. On the right side, there are buttons for CONNECT, CONFIGURATION, and three dots, with 'Load Sample Dataset' circled in red. Other buttons include 'Terminate' and 'Operations R: 0 W: 0 100.0/s'. A note says 'This is a Shared Tier Cluster' with a link to upgrade.

Когда данные загружаются, сделаем некоторые настройки, чтобы получить доступ к данным извне. Заходим в Database Access, а затем кликаем Add New Database User в центре экрана.

The screenshot shows the MongoDB Atlas Project 0 Data Services page. The left sidebar has sections for Deployment, Database (selected), Data Services (Triggers, Data API, Data Federation), Security (Database Access circled in red), Network Access, and Advanced. The main area shows the database 'Cluster0' with its Overview, Real Time, Metrics, and SANDBOX/NODES/REPLICA SET tabs. It lists nodes: ac...shard-00-00.s... (SECONDARY), ac...shard-00-01.s... (PRIMARY), and ac...shard-00-02.s... (SECONDARY). The URL is PETR'S ORG - 2022-12-10 > PROJECT 0 > DATABASES.

На странице добавления нового пользователя базы данные вводим имя и пароль.

Add New Database User

Create a database user to grant an application or user, access to your project. Granular access control can be configured with default privileges at the project or organization using the corresponding Access Manager.

Authentication Method

Password

Certificate

MongoDB uses [SCRAM](#) as its default authentication method.

Password Authentication

e.g. new-user_31

Enter password

Autogenerate Secure Password

Copy

В разделе Built-in Role выбираем Read and write to any database. Здесь также можно ограничить права пользователя на доступ к некоторым кластерам или ограничить время пользования базой данных, но сейчас мы этого делать не будем.

Просто добавляем пользователя — Add User.

Built-in Role

Select one [built-in role](#) for this user.

Select Role

Atlas admin

Read and write to any database

Only read any database

Specific Privileges

Возвращаемся к нашему кластеру. Теперь нам нужно подключиться к нему — нажимаем Connect.

Database Deployments

Find a database deployment...

Cluster0 **Connect** View Monitoring Browse Collections ...

Enhance Your Experience
For production throughput and richer metrics, upgrade to a dedicated cluster now!

R 0 W 0 Last 6 hours 100.0/s Connections Last 6 hours 8.0

Upgrade

VERSION	REGION	CLUSTER TIER	TYPE
5.0.14	AWS / Bahrain (me-south-1)	M0 Sandbox (General)	Replica Set - 3 nodes

В следующем окне выбираем Connect your application, потому что собираемся использовать Python для взаимодействия с базой данных.

В следующем окне нужно выбрать DRIVER. В нашем случае – Python. VERSION – это не версия Python, а версия драйвера Mongo. После этого появляется строка соединения (connection string).

Connect to Cluster0

✓ Setup connection security ✓ Choose a connection method Connect

1 Select your driver and version

DRIVER	VERSION
Python	3.12 or later

2 Add your connection string into your application code

Include full driver code example

```
mongodb+srv://petr:<password>@cluster0.sqjshf7.mongodb.net/?retryWrites=true&w=majority
```

Replace **<password>** with the password for the **petr** user. Ensure any option params are [URL encoded](#).

Эта строка нужна для подключения из Python и служит той же цели, что и строка подключения, которую мы сохраняли в локальной версии MongoDB.

Здесь, как и в локальной версии, можно добавлять коллекции. Перейдем на вкладку Collections.

The screenshot shows the 'ClusterO' overview page. At the top, there are tabs: 'Overview' (highlighted in green), 'Real Time', 'Metrics', 'Collections' (circled in red), 'Search', and 'Profiler'. Below the tabs, there are three buttons: 'SANDBOX', 'NODES', and 'REPLICA SET'. A section labeled 'REGION' shows 'Bahrain (me-south-1)' with three nodes listed: 'ac...shard-00-00.s...' (SECONDARY), 'ac...shard-00-01.s...' (PRIMARY), and 'ac...shard-00-02.s...' (SECONDARY). To the right, a callout box states 'This is a Shared Tier Cluster' with a note: 'If you need a database that's better for high-performance production applications, upgrade to dedicated cluster.' A 'Upgrade' button is located at the bottom right of this box.

Слева вы видите примеры баз данных, которые мы загружали чуть ранее. Теперь добавим собственную. Нажимаем Create Database, вводим Database name и Collection name. Нажимаем Create.

The dialog box has a title 'Create Database' with a close button 'x'. It contains two input fields: 'Database name' with 'test_airbnb' and 'Collection name' with 'listings'. Below these are 'Additional Preferences' with two checkboxes: 'Capped Collection' and 'Time Series Collection', both of which are unchecked. At the bottom are 'Cancel' and 'Create' buttons.

Итак, когда база данных создана, мы можем приступить к взаимодействию с ней из Python-кода. Мы будем работать с локальной версией.

Создание баз данных и коллекций MongoDB в Python

Для работы с MongoDB из Python нам понадобится **PyMongo** – один из драйверов или клиентских библиотек MongoDB. Основная задача драйвера – поддерживать соединение с базой данных и позволить вам работать на выбранном языке программирования. Подробно обо всех поддерживаемых драйверах можно прочитать на сайте [MongoDB API Documentation](#).

Установка PyMongo: [PyMongo – MongoDB Drivers](#).

```
pip install pymongo
```

Продолжим использовать пример с игрой «Ведьмак 3». Перейдем к коду.

Из библиотеки pymongo импортируем класс MongoClient. Далее создаем экземпляр класса. Обратите внимание, что здесь мы указываем строку подключения. В этой строке мы можем подключиться к любой базе данных MongoDB, к которой у нас есть доступ, например, в облаке или локально.

```
from pymongo import MongoClient  
client = MongoClient('mongodb://localhost:27017')
```

Далее возьмем словарь Python с данными об игре. Сейчас попробуем записать эти данные в коллекцию, которую мы создали в Compass. Помним, что коллекция games находится в базе данных steam.

```
witcher = {  
    "appid": 292030,  
    "positive": 632627,  
    "negative": 25245,  
  
    "name" : "The Witcher 3: Wild Hunt",  
    "developer" : "CD PROJEKT RED",  
    "publisher" : "CD PROJEKT RED",  
    "genre" : "RPG",  
    "release_date" : "2015/05/18",  
  
    "tags" : {  
        "Open World" : 11677,  
        "RPG" : 10024,
```

```

        "Story Rich" : 9219,
        "Atmospheric" : 6478,
        "Mature" : 6234,
        "Fantasy" : 6057
    }
}

db = client.steam
db.games.insert_one(witcher)

```

Мы указываем, что хотим использовать базу данных steam, и используем метод `insert_one` для передачи данных словаря `witcher` в коллекцию.

Наконец, мы сделаем запрос `find`, который просто выведет все документы коллекции `games` (в нашем случае один — тот, который мы только что записали в нее).

```

for a in db.games.find():
    print (a)

```

Итак, давайте запустим код:

```

# импорт модуля
from pymongo import MongoClient

# создание нового объекта MongoClient для подключения к
локальному серверу MongoDB
client = MongoClient('mongodb://localhost:27017')

# создание словаря Python с информацией об игре
witcher = {
    "appid": 292030,
    "positive": 632627,
    "negative": 25245,

    "name" : "The Witcher 3: Wild Hunt",
    "developer" : "CD PROJEKT RED",
    "publisher" : "CD PROJEKT RED",
    "genre" : "RPG",
    "release_date" : "2015/05/18",

    "tags" : {
        "Open World" : 11677,
        "RPG" : 10024,
    }
}

```

```

        "Story Rich" : 9219,
        "Atmospheric" : 6478,
        "Mature" : 6234,
        "Fantasy" : 6057
    }
}

# подключение к базе данных steam на сервере MongoDB
db = client.steam
# добавление данных в коллекцию games в базе данных steam
db.games.insert_one(witcher)

# найти все документы в коллекции games и вывести их на консоль
for a in db.games.find():
    print (a)

```

Мы видим: то, что мы получили обратно, — единственный документ, который мы записали в эту коллекцию. Обратим внимание на дополнительную запись, которая появилась в документе, — '_id'. Это уникальный идентификатор документа, который MongoDB добавляет автоматически. Таким образом, MongoDB гарантирует, что любой документ, который мы добавляем, может быть однозначно идентифицирован по полю id.

```
In [10]: runfile('/Users/petrrubin/Downloads/mongoDB_lesson_03.py', wdir='/Users/petrrubin/Downloads')
{'_id': ObjectId('6399e1d464fa47d291852adb'), 'appid': 292030, 'positive': 632627, 'negative': 25245,
 'name': 'The Witcher 3: Wild Hunt', 'developer': 'CD PROJEKT RED', 'publisher': 'CD PROJEKT RED',
 'genre': 'RPG', 'release_date': '2015/05/18', 'tags': {'Open World': 11677, 'RPG': 10024, 'Story Rich': 9219, 'Atmospheric': 6478, 'Mature': 6234, 'Fantasy': 6057}}
```

Примечание: как часто бывает с данными, у некоторых записей или документов будут поля, которых нет у других. В любом проекте модель данных обычно проходит через несколько редакций. MongoDB была разработана для решения обеих этих проблем. Она представляет гибкую схему, которая хорошо справляется как с отдельными документами, различающимися по полям, так и со схемой для всей коллекции, которую нужно изменить.

Для работы мы будем использовать датасет, включающий информацию об играх из Steam. JSON-файл с данными прилагается.

Для начала удалим нашу базу данных из MongoDB, а затем создадим новую и загрузим в нее JSON-датасет с информацией об играх.

Все операции можно выполнить из консоли или в графическом интерфейсе.

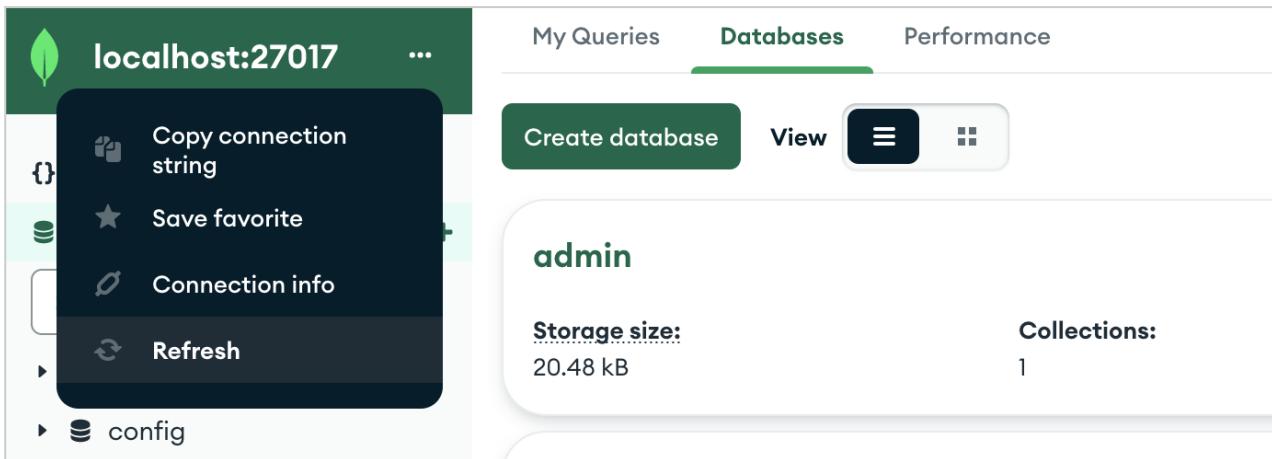
Чтобы посмотреть список всех имеющихся баз данных, выполняем команду:

```
database_list = client.list_database_names()  
print(database_list)
```

Команда для удаления базы данных:

```
client.drop_database('steam')
```

Можно проверить в Compass, что все получилось. Для этого обновим список баз данных: в Compass нажимаем три точки в верхнем правом углу и выбираем Refresh.



Теперь для создания базы данных и загрузки JSON-файла из Python можно запустить скрипт или сделать то же самое в графическом интерфейсе пользователя:

```
import json  
from pymongo import MongoClient  
client = MongoClient('mongodb://localhost:27017/')  
db = client['steam']  
  
collection = db['games']  
with open('steam_games.json') as file:  
    file_data = json.load(file)  
  
collection.insert_one(file_data)
```

Разберем код.

- `import json` – строка импортирует встроенный модуль Python `json`.

- `from pymongo import MongoClient` — импортируем класс `MongoClient` из библиотеки `pymongo`.
- `client = MongoClient('mongodb://localhost:27017/')` — создаем новый объект `MongoClient` и подключаемся к серверу MongoDB, запущенному на локальной машине на порту по умолчанию 27017. Вы можете заменить `localhost` на имя хоста или IP-адрес удаленного сервера MongoDB, чтобы подключиться к нему вместо этого.
- `db = client["steam"]` — создает новую базу данных под названием `steam` и присваивает ее переменной `db`. Если база данных уже существует, это приведет к подключению к ней.
- `collection = db["games"]` — создает новую коллекцию под названием `games` в базе данных `steam` и присваивает ее переменной `collection`. Если коллекция уже существует, это приведет к подключению к ней.
- `with open('steam_games.json') as file:` — открываем файл JSON с именем `steam_games.json` в текущем каталоге в режиме чтения. Оператор `with` используется здесь для автоматического закрытия файла после завершения блока кода.
- `file_data = json.load(file)` — считываем содержимое файла и загружаем его в словарь Python под названием `file_data` с помощью метода `json.load()`.
- `collection.insert_one(file_data)` — вставляем данные из `file_data` как новый документ в коллекцию `games` с помощью метода `insert_one()`. Метод `insert_one()` используется здесь, потому что мы вставляем только один документ. Если бы было несколько документов для вставки, мы могли бы использовать метод `insert_many()`.

Можно записать данные в базу данных и из командной строки следующим образом:

```
mongoimport -d dbname -c collectionname --file input-file.json
```

- `mongoimport` — название инструмента командной строки для импорта данных в базу данных MongoDB.
- `-d dbname` — здесь указывается имя базы данных, в которую будут импортированы данные.
- `-c collectionname` — указывает имя коллекции, в которую будут импортированы данные.

- `--file input-file.json` — указывает входной файл в формате JSON, содержащий данные для импорта.

Мы познакомились с тем, как сделать запись в базу данных, теперь перейдем к созданию запросов.

Выполнение операций CRUD в MongoDB

Аббревиатура CRUD означает четыре операции для манипулирования данными в коллекциях: Create, Read, Update и Delete (создание, чтение, модификация, удаление). Давайте подробнее рассмотрим каждую из них.

Create служит для создания нового документа в коллекции. Вы можете использовать метод `insert_one()` или метод `insert_many()`, если хотите вставить несколько документов одновременно.

Например:

```
db.mycollection.insert_one({"name": "John", "age": 30})
```

Read — операция для чтения данных из коллекции. Вы можете использовать метод `find()`, который возвращает объект курсор, по которому вы можете выполнять итерации для доступа к документам в коллекции.

Например:

```
cursor = db.mycollection.find({"name": "John"})
for document in cursor:
    print(document)
```

Update обновляет документ в коллекции. Вы можете использовать метод `update_one()` или `update_many()`, если нужно обновить сразу несколько документов.

Например:

```
db.mycollection.update_one({"name": "John"}, {"$set": {"age": 40}})
```

Delete — удаление документа из коллекции. Вы можете использовать метод `delete_one()` или метод `delete_many()`, если хотите удалить сразу несколько документов.

Например:

```
db.mycollection.delete_one({ "name": "John" })
```

Создание запросов в MongoDB

Запросы в MongoDB основаны на модели документов. Чтобы найти интересующие нас элементы, мы должны указать документ, который ищем, и что именно ищем. Рассмотрим пример с нашим датасетом.

Если мы хотим найти все игры, разработчик которых — Valve, создаем запрос следующим образом:

```
db.games.find({ "developer" : "Valve" })
```

Общий принцип такой: у нас есть поле developer и значение Valve, которые мы хотели бы найти в датасете.

Запустим следующий скрипт и посмотрим на результат.

```
from pymongo import MongoClient

client = MongoClient('mongodb://localhost:27017')
db = client.steam

def find():
    query = { "developer" : "Valve" }

    games = db.games.find(query)
    for a in games:
        print(a)

if __name__ == '__main__':
    find()
```

Мы видим, что во всех документах, которые мы получили, есть значение производителя Valve.

Это был простой запрос в MongoDB. Если мы хотим получить более точную информацию, можем просто добавить дополнительные поля в запрос.

Например, чтобы найти все игры разработчика Valve в жанре Action:

```
query = {"developer": "Valve",  
         "genre": "Action"}
```

Последнее, что рассмотрим, прежде чем перейдем к более продвинутым темам запросов — возможность указать проекцию в дополнение к нашей строке запроса. Проекция означает указание на выбор документов, отвечающих определенным критериям.

Допустим, вместо того чтобы получить все документы полностью, мы хотим получить только названия игр. Для этого можем указать проекцию в качестве второго параметра find — и вместо полного содержимого документов получим только названия игр (по умолчанию мы также получим ID, поэтому уставим `_id` в 0, чтобы он не выводился).

```
query = {"developer": "Valve",  
         "genre": "Action"}  
  
projection = {"_id": 0, "name": 1}  
games = db.games.find(query, projection)
```

Запустим код и увидим, что документы содержат только поле имени.

Операторы MongoDB

Во многих ситуациях нам нужно сделать выборку по неточным критериям. Например, все люди старше определенного возраста, или все города с населением, превышающим некоторое число. MongoDB предоставляет множество операторов для решения подобных задач. Идея с операторами MongoDB похожа на то, что мы видим в языках программирования, где есть операторы сравнения, неравенства и так далее.

Операторы MongoDB используют тот же синтаксис, что мы использовали для создания запросов, но операторы выделяются с помощью знака доллара. Давайте рассмотрим условные операторы.

Условные операторы задают условие, которому должно соответствовать значение ключа документа:

- **\$eq** — равно,
- **\$ne** — не равно,
- **\$gt** — больше чем,
- **\$lt** — меньше чем,
- **\$gte** — больше или равно,
- **\$lte** — меньше или равно,
- **\$in** определяет массив значений, одно из которых должно иметь поле документа,
- **\$nin** определяет массив значений, которые не должно иметь поле документа.

Рассмотрим несколько запросов к нашей коллекции игр. Представьте, что мы хотим запросить все игры с числом положительных отзывов более 500 000. Используем следующий синтаксис:

```
query = {"positive" : {"$gt" : 500000}}
```

Если мы хотим запросить все игры с числом положительных отзывов от 500 000 до 600 000, то запрос будет таким:

```
query = {"positive" : {"$gt" : 500000, "$lte" : 600000}}
```

Немного перепишем скрипт, чтобы отображать только количество получаемых документов:

```
from pymongo import MongoClient

client = MongoClient('mongodb://localhost:27017')

db = client.steam

def find():
    query = {"positive" : {"$gt" : 500000, "$lte" : 600000}}
    games = db.games.find(query)

    num_games = 0
    for i in games:
        print(i)
        num_games += 1
```

```
print('Число игр: %d' % num_games)

for a in games:
    print(a)

if __name__ == '__main__':
    find()
```

Этот тип запроса на основе диапазона работает с различными типами данных. Мы можем сделать нечто подобное для строк. Давайте найдем количество игр, названия которых находятся в диапазоне от А до С (иными словами, все игры, название которых начинается на А и В).

```
query = {"name" : {"$gte" : "A", "$lt" : "C"}}
```

Аналогичные запросы можно делать и для других типов упорядоченных данных, например, для дат.

Наконец, мы также можем использовать оператор \$ne (not equals), чтобы, например, найти все игры, в которых разрешенный возраст не равен 0:

```
query = {"required_age" : {"$ne" : 0}}
```

Рассмотрим еще несколько операторов. Ранее мы рассмотрели запрос к полям со строковым значением. У вас может возникнуть потребность выполнить более сложный запрос по строковым значениям. Например, найти все строки, начинающиеся с определенной комбинации букв и так далее. MongoDB поддерживает подобные запросы с помощью оператора regex. Regex основан на библиотеке регулярных выражений. Дополнительную информацию можно почитать по ссылкам:

- Live RegEx tester at regexpal.com.
- MongoDB [\\$regex Manual](#).
- Official Python [Regular Expression HOWTO](#).

Рассмотрим несколько примеров. Запрос ниже выведет все игры, в названии которых есть сочетание букв Puzzle:

```
query = {"name" : {"$regex" : "Puzzle"}}
```



```
>>> 469
```

Важно понимать, что Puzzle здесь — это не строка, а регулярное выражение. Мы знаем, что в некоторых названиях слово puzzle может начинаться не только с заглавной, но и со строчной буквы. Внесем простое изменение и получим все названия с сочетанием букв puzzle, начинающихся с заглавной или строчной буквы:

```
db.games.count_documents({ "name" : { "$regex" : "[Pp]uzzle" } })  
>>> 505
```

Мы получили 505 документов, а было 469. Расширим запрос и включим еще одно слово.

```
query = { "name" : { "$regex" : "[Pp]uzzle | [Gg]ame" } }  
>>> 1031
```

Это регулярное выражение будет идентифицировать все документы, содержащие либо слово puzzle, либо слово game. Любое из этих слов может быть написано с заглавной или строчной буквы.

Одна из самых мощных особенностей MongoDB — возможность выполнять запросы к полям, которые не являются просто единичными значениями, такими как строки или целые числа, но и к структурированным данным, таким как массивы. Мы можем выполнять поиск в массивах несколькими способами.

Возьмем для примера поле categories. Сделаем запрос на вывод игр со значением категории Со-оп.

```
query = { "categories" : "Со-оп" }
```

Несмотря на то, что значение для ключа categories — массив, MongoDB ищет внутри массива то, что мы запросили.

Что, если мы хотим найти несколько значений внутри одной категории? Например, все игры с категорией Со-оп и Online PvP. Для этого нам понадобится ввести оператор \$in.

```
query = { "categories" : { "$in" : [ "Со-оп", "Remote Play on  
Tablet" ] } }  
>>> 8599
```

`$in` позволяет указать массив значений. В этом запросе мы получим все документы, для которых поле `categories` содержит любое из значений этого массива. При этом можно использовать `$in` с любыми полями, не только с массивами.

```
query = {"categories" : {"$in" : ["Co-op", "Remote Play on Tablet", "Steam Achievements"]}}
```

>>> Число игр: 29805

И, наконец, последний способ для работы с помощью массивов — оператор `$all`. По сути, он делает работу, противоположную `$in`. При использовании `$all` мы указываем все более узкую цель. Чтобы найти игры, категории которых включены в массив:

```
query = {"categories" : {"$all" : ["Co-op", "Remote Play on Tablet", "Steam Achievements"]}}
```

>>> Число игр: 231

Мы можем еще больше сузить запрос:

```
query = {"categories" : {"$all" : ["Steam Trading Cards", "Co-op", "Remote Play on Tablet", "Steam Achievements"]}}
```

>>> Число игр: 145

Обновление данных в коллекции MongoDB

Итак, мы рассмотрели несколько способов взаимодействия с коллекциями MongoDB. Мы узнали, как добавлять документы в коллекцию, и разобрали несколько способов запросов для поиска интересующих нас документов. Теперь перейдем к обновлению, то есть к внесению изменений в существующие документы в коллекции. В MongoDB есть несколько способов сделать это.

Мы рассмотрим использование метода `update_one`.

Метод `update_one` обновляет первый совпадший документ в коллекции на основе заданного запроса. При обновлении документа значение поля `_id` остается неизменным. Метод обновляет один документ за раз, а также может добавлять новые поля в заданный документ. `Update_one` ожидает документ запроса в качестве первого параметра, а затем в качестве второго параметра ожидает ключ и значение, которое нужно обновить.

Оператор `set` работает следующим образом: если документ еще не содержит указанные поля, поле с данным значением должно добавиться. Если документ уже содержит указанное поле, оно обновится до указанного значения.

Давайте изменим документ какой-нибудь игры, например, Quake. Сделаем запрос к коллекции игр, чтобы найти Quake:

```
game = db.games.find_one({"name" : "Quake"})
```

Мы получим один документ.

Причина, по которой мы гарантированно получим один документ, в том, что вместо команды `find` мы используем команду `find_one` — она возвращает первый найденный документ.

Сейчас модифицируем документ игры и добавим поле с системными требованиями.

```
sysreq = ["Win 10 64-bit version",
          "Intel Core i5-3570 @3.4 GHz or AMD Ryzen 3 1300X @3.5 GHz",
          "NVIDIA GeForce GTX 650 TI (2GB) or AMD HD 7750 (1GB)",
          "8GB System RAM",
          "Minimum 2GB free space on hard drive (additional space required
for add-on downloads)",
          "High speed broadband connection required for online play"]

db.games.update_one ({ "name" : "Quake"}, { "$set" : { "sysreq" :
sysreq} } )
```

Давайте запустим это и проверим, что документ обновился должным образом.

Оператор `unset` — противоположность `set`. Если `set` добавляет новое поле, то `unset` удаляет существующее. Если в документе еще нет указанного поля, у вызова не будет эффекта.

Итак, мы рассмотрели команду `update_one`, которая позволяет изменять документы, и два оператора, которые используются в сочетании с ней. Последнее, о чём мы поговорим в связи с обновлением документов, — как обновлять несколько документов одновременно.

Метод `update_many` обновляет все документы в коллекциях MongoDB, которые соответствуют заданному запросу. При обновлении документа значение поля `_id` остается неизменным. Этот метод также может добавлять новые поля в документ. Можно обновить вообще все документы в коллекции — для этого нужно указать пустую строку `({})` в критериях отбора.

Теперь, когда мы рассмотрели основы сбора данных с помощью MongoDB, обратимся к другой технологии, которая хорошо подходит для сбора и анализа данных — ClickHouse.

В то время как MongoDB — популярный выбор для хранения данных на основе документов, ClickHouse — это база данных, ориентированная на столбцы, оптимизированная для высокоскоростной аналитики и отчетности по большим наборам данных.

И MongoDB, и ClickHouse становятся все более важными инструментами для современного сбора и анализа данных, поскольку предлагают мощные и гибкие способы хранения, управления и запроса больших объемов данных. Рассматривая эти две темы вместе, мы сможем получить более широкий взгляд на разные инструменты и технологии для сбора и анализа данных, а также понять, какие технологии могут лучше подходить для разных случаев использования.

Основы работы в ClickHouse

ClickHouse — это колоночная аналитическая СУБД, разработанная Яндексом.

Начнем с подключения к сервису. Чтобы создать бесплатное подключение к ClickHouse в ClickHouse Cloud, нужно зарегистрироваться или войти с помощью Google-аккаунта. После входа в систему ClickHouse Cloud запустит мастер настройки, который проведет вас через создание новой службы ClickHouse. Выберем желаемый регион для развертывания службы и дадим новой службе имя. Нажмем Create Service.

Email verified - Nice to meet you, Петр Рубин!

Let's create your first hosted ClickHouse database. This will just take a few moments.

Cloud provider

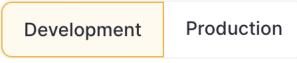


Region



Service name

Purpose



- ✓ Great for smaller workloads and proof of concepts
- ✓ Up to 1 TB storage and 16 GB total memory
- ✓ Competitive monthly pricing

AWS Private Link support, Uptime SLAs, and automatic scaling are available for production services only

Excellent, we're now creating your first service...

Where would you like to connect from?

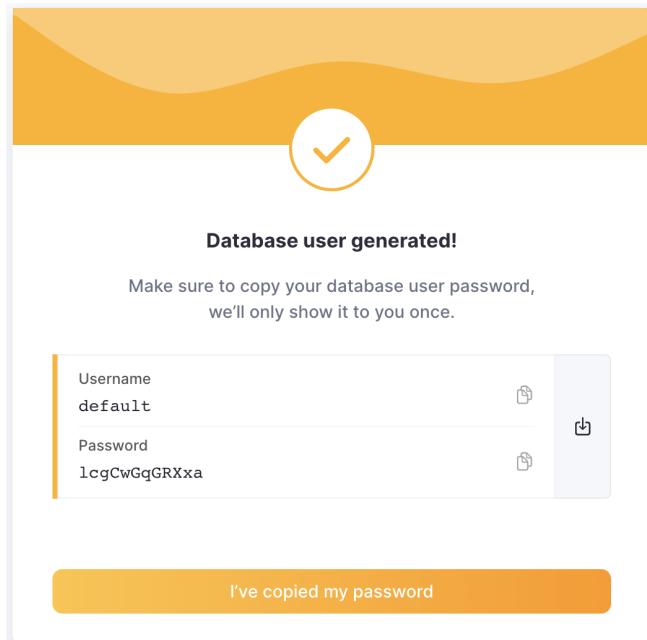
To ensure an extra layer of security for your service, we recommend only allowing traffic from IP addresses that you know and trust.

Specific locations
Control where your service can be accessed from

Anywhere
Allow service access to anyone with your credentials

Continue

ClickHouse Cloud генерирует пароль для пользователя по умолчанию — не забудьте сохранить свои учетные данные. Вы всегда сможете изменить их позже.



Новый сервис будет создан. Мы видим его на дашборде ClickHouse Cloud.

A screenshot of the ClickHouse Cloud service dashboard for a service named "test". The top navigation bar has tabs for "Actions" and "Connect". Below the tabs, there are three main sections: "Summary" (underlined), "Backups", and "Security". The "Summary" section contains a "Congratulations!" message, stating "Your service has been created - but ClickHouse is better with data! Please choose what you'd like to do next." It features two cards: "Load data" (with a cloud icon and "Get Started" button) and "Connect to SQL Console" (with a database icon and "Get Started" button). The "Load data" card includes a sub-note: "Start loading data into your service, we'll walk you through the process of setting up."

Давайте немного остановимся и поговорим об архитектуре ClickHouse. В отличие от других SQL СУБД, когда создается база данных в ClickHouse, каждая таблица в базе данных имеет **table engine**, который определяет, где и как хранятся данные.

В ClickHouse есть разные table engines, разделенные на 4 категории:

1. MergeTree Family
2. Log Family
3. Integrations
4. Special

Табличные движки семейства **MergeTree** – ядро возможностей по работе с данными в ClickHouse. Они предоставляют большинство функций для обеспечения высокопроизводительного поиска данных.

Семейство **Log** разработано для сценариев, когда нужно быстро создать множество небольших таблиц (примерно до 1 миллиона строк).

Integrations предоставляет разные средства для интеграции с внешними системами, включая табличные движки. С точки зрения пользователя, настроенная интеграция выглядит как обычная таблица, но запросы к ней передаются внешней системе. ClickHouse поддерживает интеграцию со множеством СУБД: MySQL, MongoDB, PostgreSQL, SQLite и другими.

Остальные движки уникальны по своему назначению и не сгруппированы в семейства, поэтому они помещены в категорию **Special**. Например, движки, позволяющие отправлять HTTP-запросы к серверу, обращаться к файлам, расположенным в файле, на сайте и так далее.

Каким же движком вы должны пользоваться? Чаще всего – MergeTree (первая категория). Они самые универсальные.

Когда мы создаем таблицу для базы данных в ClickHouse, нужно определить primary key. Это отличается от того, что вы изучали ранее в SQL СУБД.

В ClickHouse primary key – это уникальный идентификатор таблицы, который используется для обеспечения целостности данных, хранящихся в таблице. Первичный ключ определяется по одному или нескольким столбцам таблицы.

Когда первичный ключ определен для таблицы, ClickHouse автоматически накладывает несколько ограничений на данные в таблице. Например, он гарантирует, что ни у каких двух строк в таблице нет одинаковых значений для столбцов первичного ключа, и что столбцы первичного ключа не могут быть обновлены до значения, которое уже существует в таблице.

В ClickHouse первичный ключ можно определить либо явно, используя синтаксис PRIMARY KEY в операторе CREATE TABLE, либо неявно, указав уникальное ограничение на один или несколько столбцов таблицы. Например, если мы не определим primary key, то можем использовать сортированный список.

The ORDER BY tuple becomes the PRIMARY KEY

```
CREATE TABLE my_table
(
    column1    FixedString(1),
    column2    UInt32,
    column3    String
)
ENGINE = MergeTree()
ORDER BY (column1, column2)
```

Источник: learn.clickhouse.com

В приведенном примере у нас есть таблица. Тип первого столбца — FixedString, второго — UInt32. Мы можем использовать эти столбцы в качестве primary key, если предполагаем, что поиск по базе данных будет производиться в первую очередь по этим столбцам.

Давайте посмотрим, как MergeTree хранит данные.

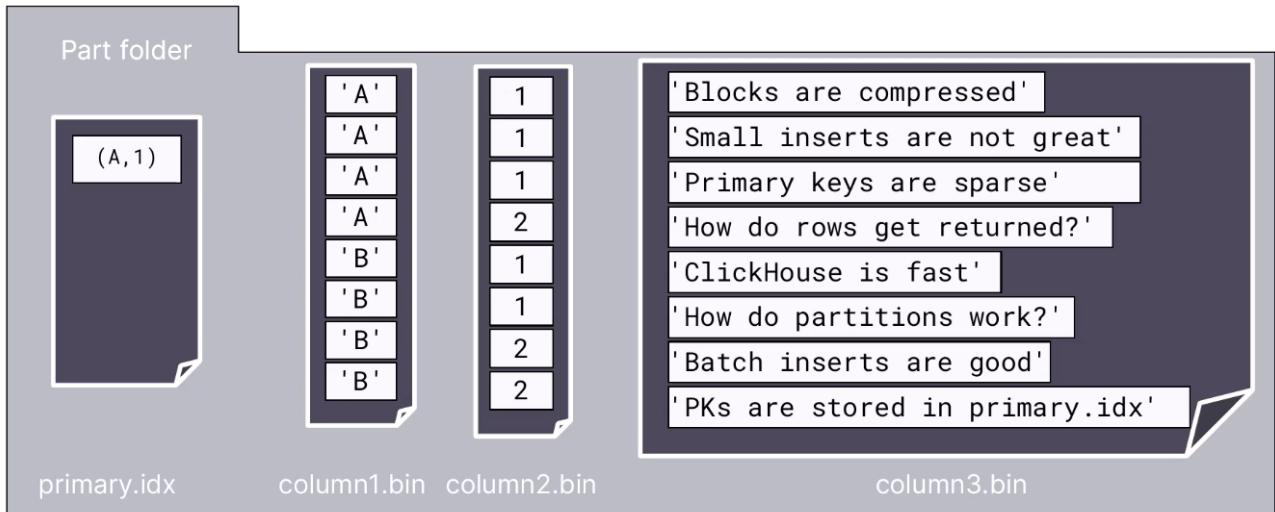
В ClickHouse мы, как правило, не вводим данные по одной строке. Мы выполняем массовую загрузку — загружаем настолько много данных, насколько можем: например, миллион строк в секунду.

Почему важно иметь в виду каждую загрузку данных, будь то одна строка или миллион? Потому что при каждой загрузке данных создается так называемая часть (part), которая хранится в отдельной папке.

```
INSERT INTO my_table (column1, column2, column3) VALUES
('B', 1, 'ClickHouse is fast'),
('A', 1, 'Blocks are compressed'),
('B', 2, 'Batch inserts are good'),
('A', 1, 'Small inserts are not great'),
('B', 1, 'How do partitions work?'),
('B', 2, 'PKs are stored in primary.idx'),
('A', 2, 'How do rows get returned?'),
('A', 1, 'Primary keys are sparse')
```

Источник: learn.clickhouse.com

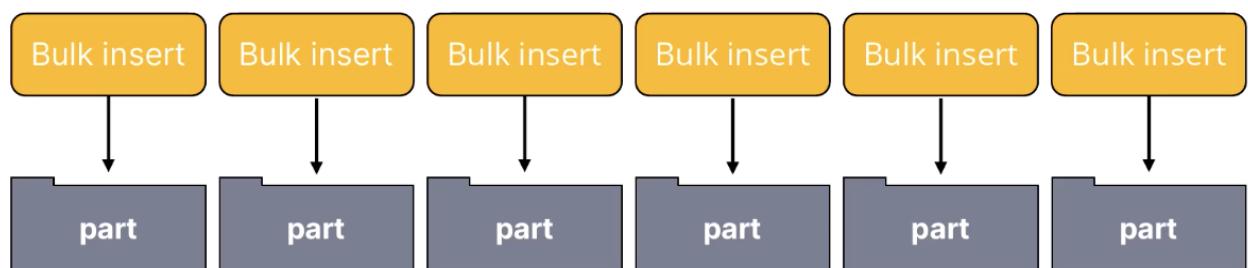
В этом случае мы загружаем 8 строк, и одна эта загрузка создает отдельный блок — part. Как вы помните, ClickHouse — колоночная СУБД, поэтому после загрузки данные будут организованы в таблицу, каждая из колонок которой будет представлена отдельным файлом, как показано на рисунке:



Источник: learn.clickhouse.com

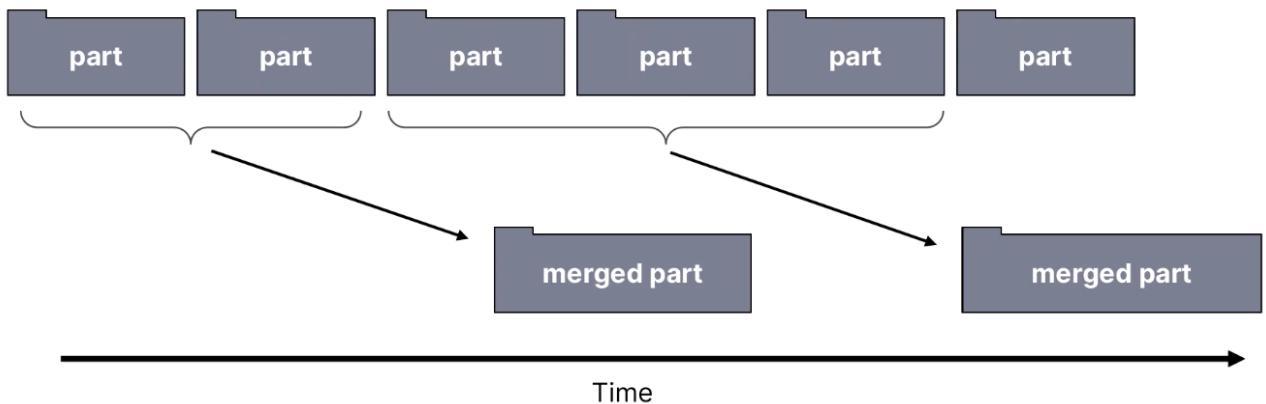
Обратите внимание, что произошло: записи в колонке 1 и 2 отсортированы, потому что мы указали на это командой ORDER BY, и это служит первичным ключом в файле primary.idx.

Представьте, что вы постоянно загружаете порции информации, миллион записей в секунду, и каждая эта порция записывается в отдельную директорию.



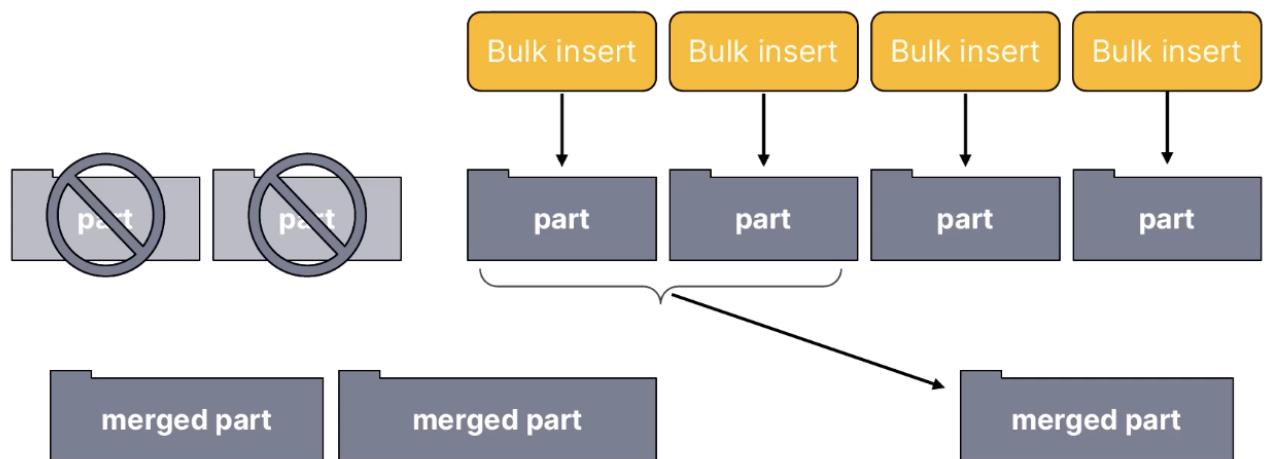
Источник: learn.clickhouse.com

Очевидно, что через некоторое время у вас будет огромное количество директорий в вашей файловой системе. Со временем эти части будут объединяться. ClickHouse выполняет эту работу.



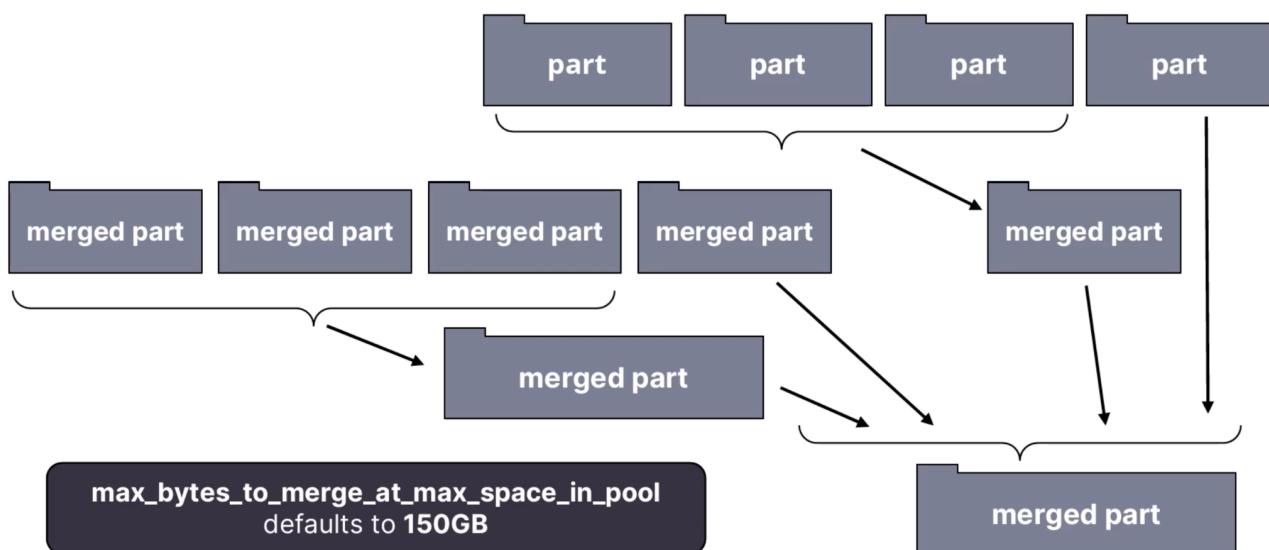
Источник: learn.clickhouse.com

По мере того как части объединяются, а новые добавляются, старые части будут удаляться.



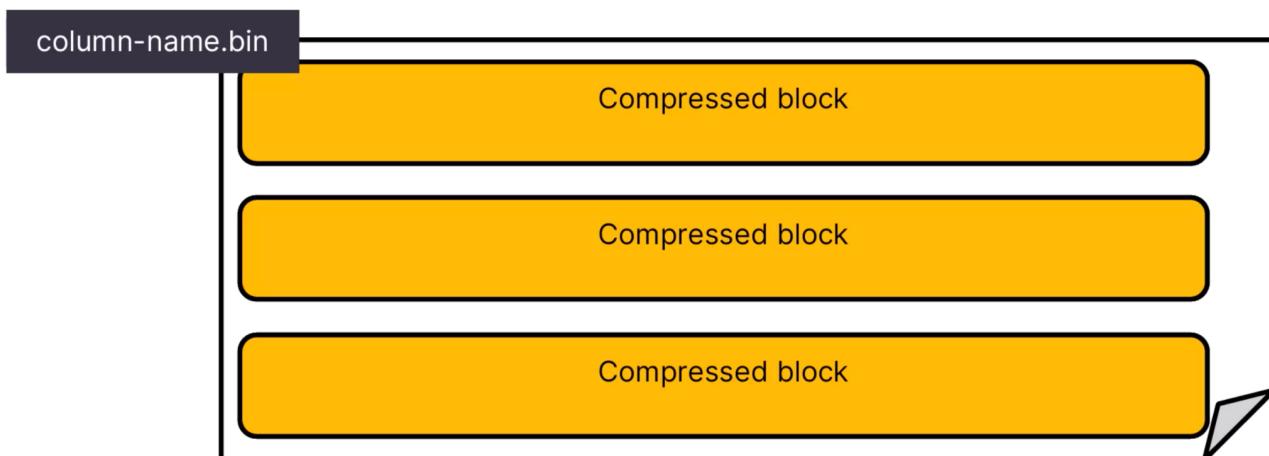
Источник: learn.clickhouse.com

Далее уже объединенные части также будут объединяться. Но этот процесс не будет продолжаться бесконечно, и максимальный размер файла ограничен 150 GB. По мере того как части объединяются, продолжается и сортировка. Все это ClickHouse выполняет по определенным алгоритмам.



Источник: learn.clickhouse.com

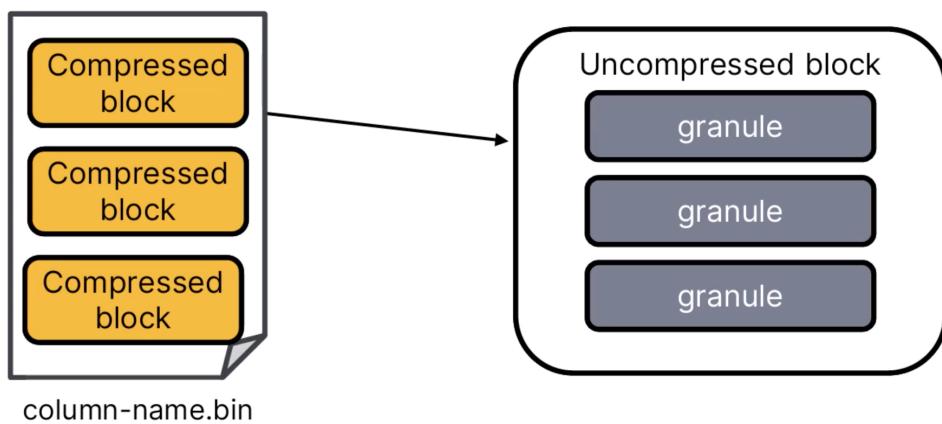
Файлы колонок также подвергается сжатию. При этом компрессия происходит не в один большой файл, точнее, формируется один файл, но он разбит на части, которые называются блоками (block).



Источник: learn.clickhouse.com

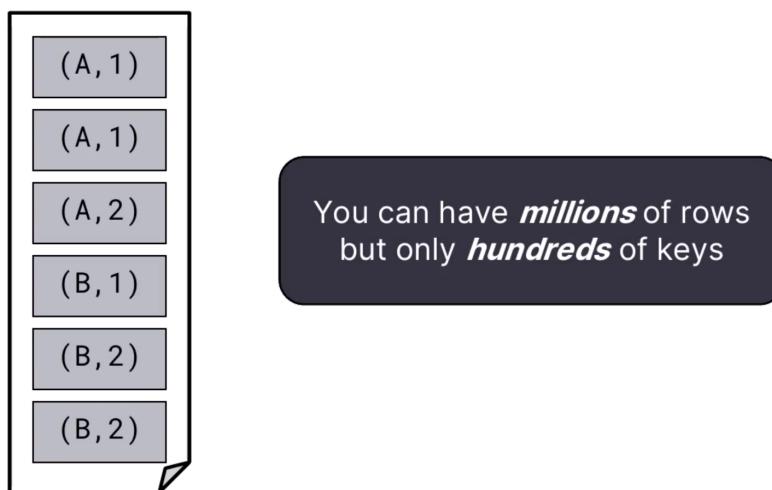
Причина, по которой это сделано: когда мы делаем запрос, ClickHouse не разархивирует один большой файл, а находит один или два блока, в котором находятся нужные строки, и разархивирует их.

Как ClickHouse находит эти блоки внутри сжатого файла? Для этого используется концепция, которая называется «гранулы» (granule). Файл первичных ключей указывает на первую строку каждой гранулы, что соответствует 8192 строкам или 10 МВ (значение по умолчанию, которое можно менять). Все это логическая структура, а не то, как на самом деле физически файлы хранятся на диске.



Источник: learn.clickhouse.com

Гранулы позволяют найти нужные нам строки максимально быстро. Предположим, у нас шесть гранул, как на картинке ниже. У нашего файла с primary index порядка 8192 x 6 строк. Первый индекс (A, 1) указывает на первые 8192 строки, следующий индекс указывает на следующие 8192 строки, что соответствует второй грануле, далее индекс (A, 2) указывает на 16385 строку и так далее.



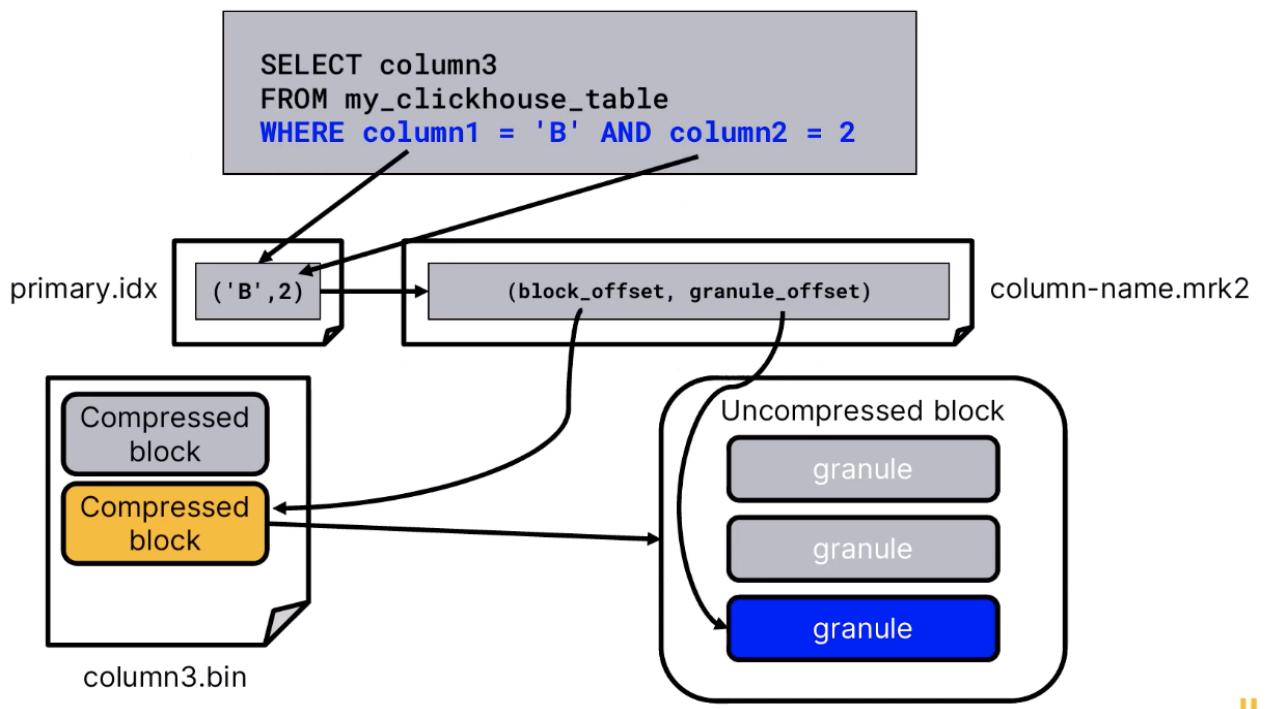
Источник: learn.clickhouse.com

Такая распределенная структура позволяет хранить файл первичных ключей в памяти. Даже если у нас миллионы строк, если их количество разделить на 8192, то станет ясно, что количество гранул и первичных ключей будет измеряться сотнями на миллионы строк, что обеспечивает быстродействие.

Итак, у нас есть гранулы внутри файла первичных ключей, есть блоки в файле столбца. Возникает вопрос: как мы соотносим строку в первичном ключе с блоком в файле столбца? Для этого есть дополнительные файлы — marks. Если мы посмотрим директорию, в которой ClickHouse хранит данные, то увидим файлы .mrk2. А файл меток определяет то, как сопоставляется первичный ключ с фактическим блоком

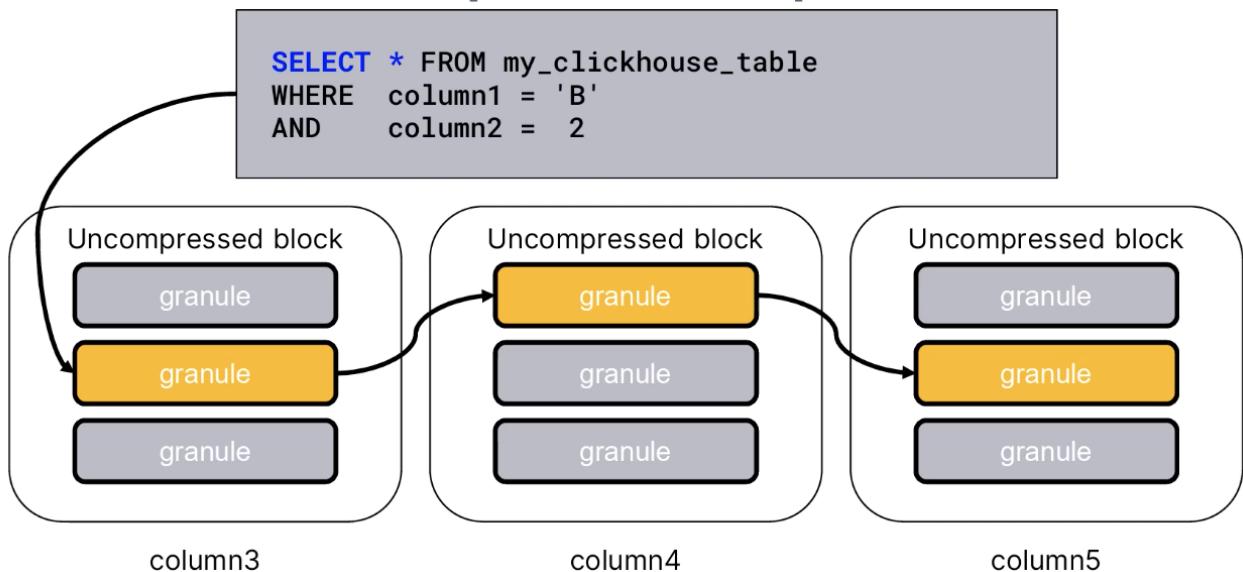
внутри файла колонок. Таким образом, файл меток знает, какой блок использовать, а затем внутри каждого блока он знает, какую гранулу обрабатывать.

Например, у нас есть запрос, в котором мы выбираем столбец 3 и передаем значения для столбцов 1 и 2. Столбцы 1 и 2 – это то, что находится в файле первичного ключа. Файл первичного ключа сразу определит, где находятся первые гранулы для этого конкретного запроса, и будет использовать mark-файл, чтобы определить, какие соответствующие блоки, находящиеся в файле column3.bin, должны быть распакованы. И затем, когда эти блоки распаковываются, mark-файл также знает, какую гранулу внутри этих блоков обрабатывать. В примере нужно обработать только одну гранулу.



Источник: learn.clickhouse.com

По сути, получается, что когда мы записывали данные, то мы конструировали блоки, а теперь, когда кто-то запрашивает данные, они восстанавливаются. Вот что происходит здесь. Эти гранулы связаны вместе в нечто, что мы называем stripe (полоса). На следующем рисунке мы берем гранулу из столбца 3, гранулу из столбца 4 и гранулу из столбца 5, и они объединяются вместе.



Источник: learn.clickhouse.com

Именно этот stripe отправляется или обрабатывается потоком для обработки. Поток просто находится в пуле потоков, ожидая, пока stripes будут обработаны, и все эти stripes обрабатываются одновременно. Если у нас 32 ядра, то в любой момент будет обрабатываться 32 stripes. И если один из этих потоков работает медленнее, чем остальные, то этот медленный поток может быть передан другому потоку. Так что эти задачи могут быть перераспределены. Это также одна из причин, почему ClickHouse работает так быстро.

Зачем мы говорим здесь обо всех этих подробностях? Важно понимать терминологию. Вам нужно понимать эти детали, чтобы читать документацию. Если кто-то упомянет гранулу при обсуждении ClickHouse, вы поймете, о чем речь.

Работа с ClickHouse в Python

Как и при работе с MongoDB, для работы с ClickHouse в Python нам нужен драйвер. Соответствующий модуль называется `clickhouse-driver`. Он предлагает простой интерфейс, позволяющий клиентам Python подключаться к ClickHouse, выполнять команды и обрабатывать результаты.

Исходный код опубликован на GitHub под лицензией MIT: [ClickHouse Python Driver with native interface support](https://github.com/ClickHouse/ClickHouse-Python-Driver).

Установка обычная — из PyPI с помощью `pip` либо через Anaconda аналогично тому, как мы делали с установкой драйвера MongoDB:

```
pip install clickhouse-driver
```

Начнем с добавления данных. Мы будем использовать часть датасета с информацией о поездках желтого такси Нью-Йорка, около 2 записей (полный датасет включает более 3 миллиардов поездок).

[New York Taxi Data | ClickHouse Docs](#)

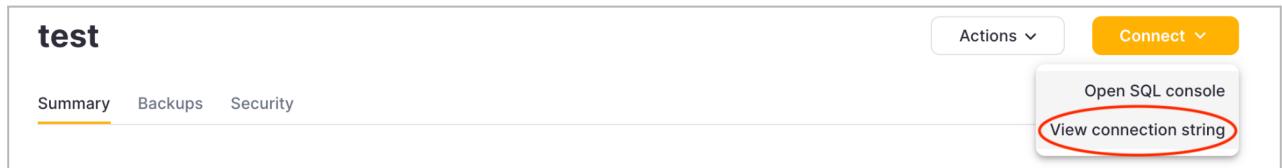
Данные о такси Нью-Йорка содержат информацию о времени и месте посадки и высадки, стоимости, сумме чаевых, дорожных сборах, типе оплаты и так далее. Создадим таблицу для хранения этих данных.

Данные возьмем из сервиса Amazon S3 (Simple Storage Service) — это служба хранения данных, предоставляемая Amazon Web Services (AWS).

Работать будем в Jupyter-ноутбуке. Clickhouse-driver прост в использовании. Основным интерфейсом является класс Client, который большинство программ импортируют напрямую.

```
from clickhouse_driver import Client
```

Для установки соединения сначала нужно скопировать в терминале ClickHouse строку подключения:



В ноутбуке создаем экземпляр класса со строкой подключения, куда вводим имя пользователя и пароль, которые вы получили при регистрации:

```
client = Client(host =
    'p7868955p1.eu-central-1.aws.clickhouse.cloud',
    user='default',
    secure = True,
    port = 9440,
    password = '*****')
```

Для выполнения запросов служит простая команда client.execute(). В скобках мы указываем SQL-запрос. Сначала создадим таблицу trips в базе данных default:

```
client.execute (
    """
CREATE TABLE trips
    (
```

```

`trip_id` UInt32,
`vendor_id` Enum8('1' = 1, '2' = 2, '3' = 3, '4' = 4, 'CMT' =
5, 'VTS' = 6, 'DDS' = 7, 'B02512' = 10, 'B02598' = 11, 'B02617' =
12, 'B02682' = 13, 'B02764' = 14, '' = 15),
`pickup_date` Date,
`pickup_datetime` DateTime,
`dropoff_date` Date,
`dropoff_datetime` DateTime,
`store_and_fwd_flag` UInt8,
`rate_code_id` UInt8,
`pickup_longitude` Float64,
`pickup_latitude` Float64,
`dropoff_longitude` Float64,
`dropoff_latitude` Float64,
`passenger_count` UInt8,
`trip_distance` Float64,
`fare_amount` Float32,
`extra` Float32,
`mta_tax` Float32,
`tip_amount` Float32,
`tolls_amount` Float32,
`ehail_fee` Float32,
`improvement_surcharge` Float32,
`total_amount` Float32,
`payment_type` Enum8('UNK' = 0, 'CSH' = 1, 'CRE' = 2, 'NOC' =
3, 'DIS' = 4),
`trip_type` UInt8,
`pickup` FixedString(25),
`dropoff` FixedString(25),
`cab_type` Enum8('yellow' = 1, 'green' = 2, 'uber' = 3),
`pickup_nyct2010_gid` Int8,
`pickup_ctlabel` Float32,
`pickup_borocode` Int8,
`pickup_ct2010` String,
`pickup_boroct2010` String,
`pickup_cdeligibil` String,
`pickup_ntacode` FixedString(4),
`pickup_ntaname` String,
`pickup_puma` UInt16,
`dropoff_nyct2010_gid` UInt8,
`dropoff_ctlabel` Float32,
`dropoff_borocode` UInt8,
`dropoff_ct2010` String,
`dropoff_boroct2010` String,

```

```

`dropoff_cdeligibil` String,
`dropoff_ntacode` FixedString(4),
`dropoff_ntaname` String,
`dropoff_puma` UInt16
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(pickup_date)
ORDER BY pickup_datetime;
'''
)

```

Код создает таблицу под названием trips в базе данных ClickHouse с помощью метода client.execute() в Python. Таблица определяется с опцией движка ENGINE = MergeTree, которая указывает, что таблица хранится в механизме MergeTree.

Таблица разбивается на разделы по столбцу pickup_date и сортируется по столбцу pickup_datetime внутри каждого раздела, что может улучшить производительность запросов при поиске или агрегировании данных в таблице.

При создании таблицы в ClickHouse важно учитывать характеристики хранимых данных и типы запросов, которые будут выполняться к этим данным. Тщательно выбирая подходящие типы данных и варианты хранения для таблицы, вы можете оптимизировать ее производительность и облегчить работу с ней в рабочих процессах аналитики и отчетности.

Теперь, когда у нас создана таблица, добавим данные о такси Нью-Йорка. Они находятся в файлах CSV в AWS S3, вы можете загрузить данные оттуда.

Следующая команда вставляет ~2 000 000 строк в таблицу trips из двух разных файлов в S3: trips_1.tsv.gz и trips_2.tsv.gz:

```

client.execute(
"""
INSERT INTO trips
SELECT * FROM s3(
'https://datasets-documentation.s3.eu-west-3.amazonaws.com/nyc-taxi/trips_{1..2}.gz',
'TabSeparatedWithNames', "
`trip_id` UInt32,
`vendor_id` Enum8('1' = 1, '2' = 2, '3' = 3, '4' = 4, 'CMT' =
5, 'VTS' = 6, 'DDS' = 7, 'B02512' = 10, 'B02598' = 11, 'B02617' =
12, 'B02682' = 13, 'B02764' = 14, '' = 15),

```

```

`pickup_date` Date,
`pickup_datetime` DateTime,
`dropoff_date` Date,
`dropoff_datetime` DateTime,
`store_and_fwd_flag` UInt8,
`rate_code_id` UInt8,
`pickup_longitude` Float64,
`pickup_latitude` Float64,
`dropoff_longitude` Float64,
`dropoff_latitude` Float64,
`passenger_count` UInt8,
`trip_distance` Float64,
`fare_amount` Float32,
`extra` Float32,
`mta_tax` Float32,
`tip_amount` Float32,
`tolls_amount` Float32,
`ehail_fee` Float32,
`improvement_surcharge` Float32,
`total_amount` Float32,
`payment_type` Enum8('UNK' = 0, 'CSH' = 1, 'CRE' = 2, 'NOC' =
3, 'DIS' = 4),
`trip_type` UInt8,
`pickup` FixedString(25),
`dropoff` FixedString(25),
`cab_type` Enum8('yellow' = 1, 'green' = 2, 'uber' = 3),
`pickup_nyct2010_gid` Int8,
`pickup_ctlabel` Float32,
`pickup_borocode` Int8,
`pickup_ct2010` String,
`pickup_boroct2010` String,
`pickup_cdeligibil` String,
`pickup_ntacode` FixedString(4),
`pickup_ntaname` String,
`pickup_puma` UInt16,
`dropoff_nyct2010_gid` UInt8,
`dropoff_ctlabel` Float32,
`dropoff_borocode` UInt8,
`dropoff_ct2010` String,
`dropoff_boroct2010` String,
`dropoff_cdeligibil` String,
`dropoff_ntacode` FixedString(4),
`dropoff_ntaname` String,
`dropoff_puma` UInt16

```

```
    ") SETTINGS input_format_try_infer_datetimes = 0  
    '''  
)
```

Код вставляет данные в таблицу ClickHouse под названием trips из двух сжатых файлов данных, хранящихся в AWS S3.

Метод client.execute() в Python используется для выполнения SQL-запроса, который использует оператор ClickHouse INSERT INTO для вставки данных в таблицу trips. Данные извлекаются из хранилища S3 с помощью функции ClickHouse s3(), которая позволяет читать данные из файла в S3 по протоколу HTTP(S).

Функция s3() принимает несколько аргументов. Первый — это URL файла или файлов для чтения, который в данном случае представляет собой диапазон файлов в S3 с именами, включающими цифры 1 или 2. Второй аргумент определяет формат считываемых данных, который в данном случае представляет собой имена столбцов, разделенные табуляцией. Третий аргумент определяет схему считываемых данных.

Пункт SETTINGS в конце запроса устанавливает опцию input_format_try_infer_datetimes в 0, что отключает автоматическое определение типов даты и времени при вводе данных. Эта опция может быть использована для повышения производительности при обработке больших объемов данных.

Когда вставка будет завершена, проверим, что она сработала:

```
client.execute("SELECT count() FROM trips")  
  
>>> [(1999657,)]
```

Загрузили около 2 миллионов строк.

Как вы видите, запрос состоит из команды SELECT, которая выполняет получение данных, за ней следует функция count(), которая подсчитывает количество строк или not-NUL значенияй, FROM trips — из таблицы trips.

Выполним простой анализ и парсинг данных. Например, вычислим среднее чаевых:

```
tip_amount = client.execute("SELECT round(avg(tip_amount), 2)  
FROM trips")  
print(tip_amount)  
  
>>> [(1.68,)]
```

Функция `avg()` вычисляет среднее арифметическое, функция `round()` округляет значение до заданного количества десятичных знаков, которое указывается в качестве второго аргумента. В этом случае мы округляем до двух знаков после запятой. Если мы проверим тип переменной `tip_amount`, то увидим, что ответ базы данных мы получаем типа список, в который вложен кортеж. Это имеет значение при выполнении вычислений в Python, если нам нужно получить число, то нужно использовать соответствующую индексацию, то есть:

```
tip_amount[0][0]
```

Теперь немного усложним запрос и рассчитаем среднюю стоимость поездки в зависимости от количества пассажиров:

```
avg_price = client.execute(
    """
SELECT
    passenger_count,
    ceil(avg(total_amount),2) AS average_total_amount
FROM trips
GROUP BY passenger_count
    """
)
```

Здесь мы запрашиваем число пассажиров — `passenger_count`, рассчитываем среднюю стоимость поездки — `avg(total_amount)`, функция `ceil()` аналогична функции `floor`, но возвращает наименьшее округленное число, которое больше или равно x. `GROUP BY` осуществляет группировку по числу пассажиров.

И снова мы получаем список кортежей, который при необходимости можно трансформировать в pandas-датафрейм для дальнейшего анализа.

```
df = pd.DataFrame(avg_price, columns=("passenger_count",
    "average_total_amount"))
```

Посмотрим ежедневное количество заказов в каждом районе Нью-Йорка:

```
pickup_num = client.execute(
    """
SELECT
    pickup_date,
    pickup_ntaname,
    SUM(1) AS number_of_trips
    """
```

```
FROM trips
GROUP BY pickup_date, pickup_ntaname
ORDER BY pickup_date ASC
'''
)
```

Запрашиваем данные о дате заказа — pickup_date, названия районов — pickup_ntaname, функция sum() считает сумму строк. ORDER BY выполняет сортировку, в нашем случае по дате в возрастающем порядке — ASC, если нужно отсортировать в нисходящем порядке, используем DESC.

Следующий запрос рассчитывает длительность поездки и группирует результаты по этому значению:

```
trip_minutes = client.execute(
'''
SELECT
    avg(tip_amount) AS avg_tip,
    avg(fare_amount) AS avg_fare,
    avg(passenger_count) AS avg_passenger,
    count() AS count,
    truncate(date_diff('second', pickup_datetime,
dropoff_datetime)/3600) as trip_minutes
FROM trips
WHERE trip_minutes > 0
GROUP BY trip_minutes
ORDER BY trip_minutes DESC
'''
)
```

Функция date_diff() возвращает разницу между двумя датами или датами со значениями времени. Разница вычисляется с использованием относительных единиц, например, разница между 2022-01-01 и 2021-12-29 составляет 3 дня для единицы измерения «день». Первый аргумент функции — тип интервала: например, ‘second’ — секунды, ‘minute’ — минуты и так далее. Таким образом, trip_minutes — это разница между временем высадки и посадки, деленная на 3600, чтобы получить время в минутах.

Функция truncate(x[, N]) возвращает округленное число с наибольшим абсолютным значением, абсолютное значение которого меньше или равно x. Во всех остальных отношениях она аналогична функции 'floor'.

Условие WHERE позволяет отфильтровать данные, поступающие из пункта FROM предложения SELECT.

Далее данные сгруппированы и отсортированы по trip_minutes.

Итак, как видно из приведенных примеров, язык запросов очень похож на SQL. Однако у него есть дополнительные функции. Подробное описание всех команд есть в справке ClickHouse: [SQL Reference | ClickHouse Docs](#).

Мы рассмотрели архитектуру ClickHouse, один из способов загрузки данных, а также базовые запросы. При правильном использовании и в подходящих сценариях ClickHouse — мощное, масштабируемое и быстрое решение.

Подводя итог, перечислим основные условия применения ClickHouse:

- работа с огромными объемами данных (измеряемыми в терабайтах), которые постоянно записываются ичитываются;
- использование таблицы с огромным количеством столбцов, но значения столбцов достаточно короткие;
- данные хорошо структурированы, но еще не агрегированы;
- данные вставляются большими партиями в тысячи-миллионы строк;
- подавляющее большинство операций — это чтение с агрегацией;
- при чтении обрабатывается большое количество строк, но довольно малое количество столбцов;
- данные позже не потребуют изменения;
- нет потребности извлекать отдельные строки.

Заключение

Итак, на этой лекции мы рассмотрели две ключевые темы: MongoDB и ClickHouse.

MongoDB — популярная база данных NoSQL, которая широко используется для хранения и управления неструктуризованными и полуструктурными данными. Мы рассмотрели основы MongoDB, включая создание баз данных и коллекций, выполнение операций CRUD над данными, а также запросы к данным.

ClickHouse — высокопроизводительная, ориентированная на столбцы база данных, предназначенная для аналитических и отчетных рабочих процессов. Мы

рассмотрели архитектуру, ключевые особенности и возможности ClickHouse. Познакомились с тем, как создавать таблицы в ClickHouse, как загружать данные, как запрашивать и анализировать данные с помощью SQL.

Лекция охватила основные концепции и методы сбора данных с использованием MongoDB и ClickHouse. Это основа для дальнейшего изучения и применения этих мощных инструментов в реальных проектах по работе с данными.

Что делать, если вы хотите продолжить изучение и эксперименты с MongoDB и ClickHouse?

1. Изучите более продвинутые функции и сценарии использования. И у MongoDB, и у ClickHouse большой спектр расширенных возможностей и примеров использования.

- Для MongoDB это может быть сегментирование данных, конвейеры агрегации и моделирование данных для конкретных случаев использования.
- Для ClickHouse — материализованные представления, распределенная обработка запросов и сжатие данных.

Изучите эти возможности и поэкспериментируйте с вариантами использования, чтобы глубже понять, на что способны эти инструменты.

2. Работайте с реальными данными. Чтобы лучше понять, как можно использовать MongoDB и ClickHouse в реальных проектах, поработайте с общедоступными наборами данных или соберите собственные.

3. Подключайтесь к другим инструментам и сервисам. И MongoDB, и ClickHouse легко интегрируются с другими инструментами и сервисами — хранилищами данных, инструментами ETL и платформами BI. Рассмотрите возможность этих интеграций и экспериментов с разными рабочими процессами, чтобы увидеть, как эти инструменты можно использовать вместе для создания комплексного решения по работе с данными.

4. Посмотрите семинары и конференции. Есть множество семинаров и конференций, посвященных MongoDB и ClickHouse, а также другим темам управления данными и аналитики. В интернете можно найти множество записей подобных мероприятий, чтобы получить знания от экспертов в области.

Что можно почитать еще?

- [Официальный сайт MongoDB](#)
- [NoSQL – Википедия](#)
- [Формат данных JSON](#)
- [ClickHouse GitHub](#)
- [YouTube-канал ClickHouse](#)