ProjectPro

Python and MongoDB Project for Beginners with Source Code

CookBook

Table of Contents:

Introduction *2*

Key Concepts *4*

CAP Theorem 4

MongoDB Architecture 5

Mongod and Mongos 6

Project -> Cluster -> Database -> Collection -> Document 6

MongoDB document limitations 7

JSON vs BSON 8

Sharding 10

MongoDB Atlas vs On-Premises Setup 11

Setting up MongoDB Atlas Cluster 12

Standard Connection vs DNS Seed List 13

MongoDB Compass 14

Use-case depicted in this project *15*

Dataset Description 15

Project flow 16

PyMongo Installation 17

Primary vs Secondary Nodes 19

Admin and Local DB 19

Oplog in MongoDB 20

CRUD operations in PyMongo 20

COLLSCAN and IXSCAN 26

Indexes in MongoDB 27

ACID Transactions 28

Aggregation Framework 28

Summary *37*

Introduction

MongoDB is a popular, open-source NoSQL database management system. It uses a document-oriented data model, which means that data is stored in documents instead of tables and rows as in traditional relational databases. The documents are stored in a BSON (Binary JSON) format, which allows for a flexible data structure and easy integration with modern web and mobile applications. MongoDB also offers features such as automatic scaling, high performance, and easy administration, making it a popular choice for modern web and mobile applications that require fast, dynamic access to large amounts of data.

MongoDB is a popular choice for big data due to several advantages it offers:

- **Scalability:** MongoDB's horizontal scalability and **sharding** capabilities make it well-suited for handling large amounts of data and high traffic levels.
- **High performance:** MongoDB uses an **in-memory storage** engine and **indexing** techniques, making it one of the fastest NoSQL databases.
- **Document-oriented:** MongoDB stores data in documents, making it easy to represent complex hierarchical relationships and handle variable schema.
- **Real-time processing:** MongoDB's support for real-time data processing and aggregation, along with rich query language, makes it ideal for use cases where you need to analyze and act on incoming data quickly.
- **Geospatial data management**: MongoDB's built-in **geospatial indexing** and querying capabilities make it well-suited for handling geospatial data, such as location data or map data.
- **Strong community:** MongoDB has a strong community of users and contributors, providing support to help you get the most out of the database.
- **Flexible data model:** MongoDB allows for flexible and **dynamic data modeling**, enabling you to store and manage data in various formats without worrying about pre-defined schemas.
- **Wide adoption:** MongoDB is widely adopted in many industries and has a proven track record of success in various use cases.
- **Ease of use**: MongoDB's intuitive and easy-to-use query language and APIs make it simple to work with, even for those **without prior database experience**.

Overall, MongoDB's scalability, performance, flexibility, ease of use, and real-time processing capabilities make it a popular choice for big data projects.

This training will help you to understand the MongoDB database, its basic architecture, the data modeling used behind MongoDB and the Aggregation Framework, along with various other Services under MongoDB Atlas for Data Engineering.

Did you know -SQL vs NoSQL?

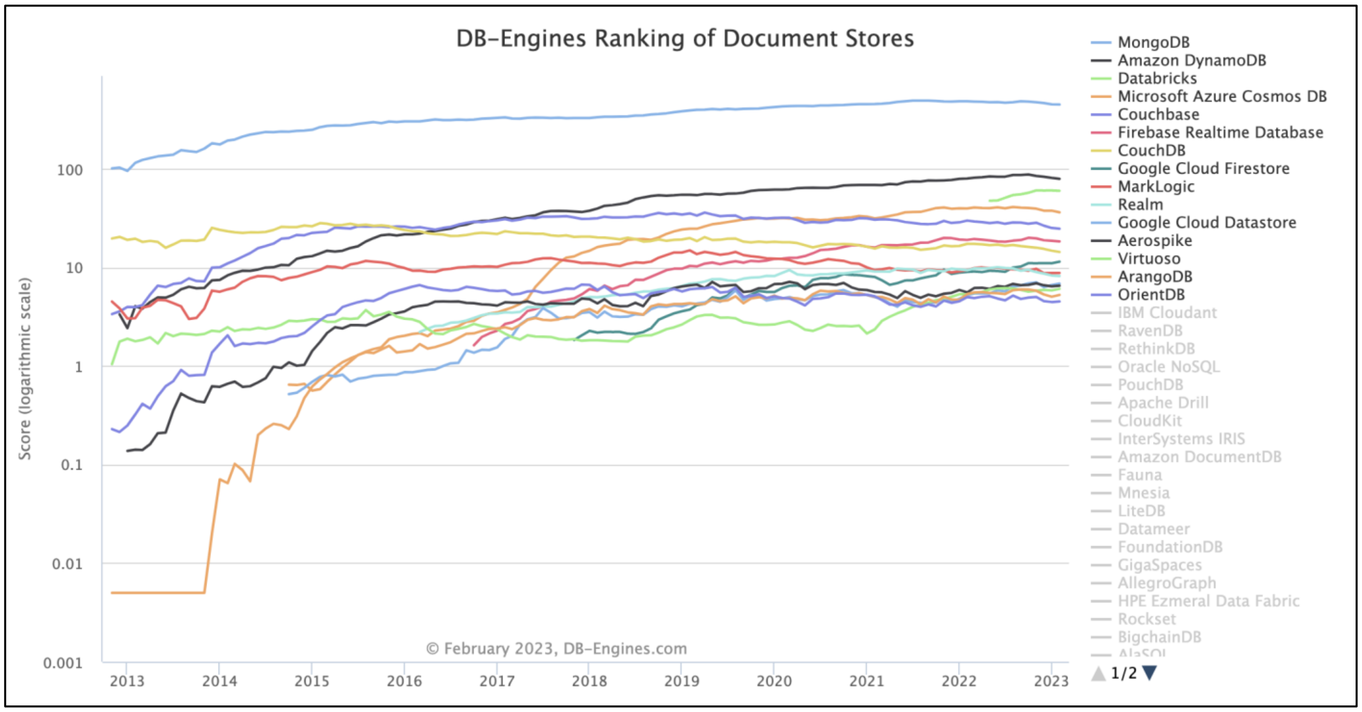
SQL (Structured Query Language) and NoSQL (Not only SQL) are two different approaches to data storage and retrieval.

SQL databases are based on the relational model and use a fixed schema, meaning the structure of the data must be defined before it is stored. They are optimized for structured data, transactions, and complex queries, making them well suited for applications that require strong consistency and reliability.

NoSQL databases, on the other hand, use a variety of data models and do not have a fixed schema. They are designed to handle large amounts of unstructured or semi-structured data, and are optimized for horizontal scalability, distributed systems, and big data processing. NoSQL databases are often used in high-traffic web and mobile applications, where the ability to scale and process large amounts of data quickly is critical.

When choosing between SQL and NoSQL, it's important to consider the specific requirements of the project, such as the type of data being stored, the volume of data, the expected read and write traffic, and the desired level of consistency and reliability. Both SQL and NoSQL have their strengths and weaknesses, and the choice will depend on the use case.

Popularity of MongoDB (VS all Document stores)



Key Concepts

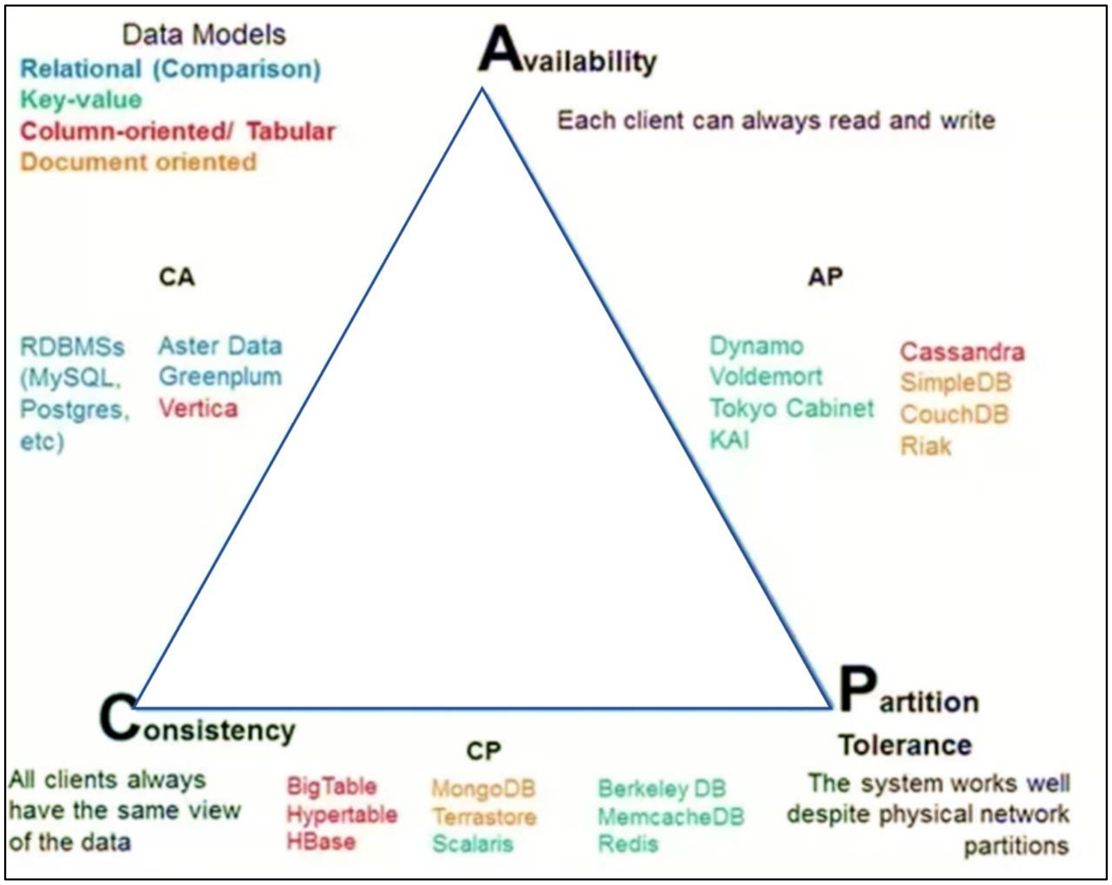
CAP Theorem

The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees: Consistency, Availability, and Partition tolerance.

In terms of the CAP theorem, MongoDB is classified as a CP (Consistent and Partition tolerant) system. This means that in the event of a network partition, MongoDB prioritizes consistency over availability. MongoDB achieves consistency through its built-in replication and automatic failover capabilities. If a primary node becomes unavailable, MongoDB will automatically failover to a secondary node to maintain data availability.

While MongoDB prioritizes consistency over availability, it still provides high levels of availability through its built-in replication and failover capabilities. The trade-off is that in the event of a network partition, there may be a brief period where the system is not available.

It is important to keep the CAP theorem in mind when choosing a database for a big data project, as it will help determine the best fit based on the specific requirements and priorities of the project.

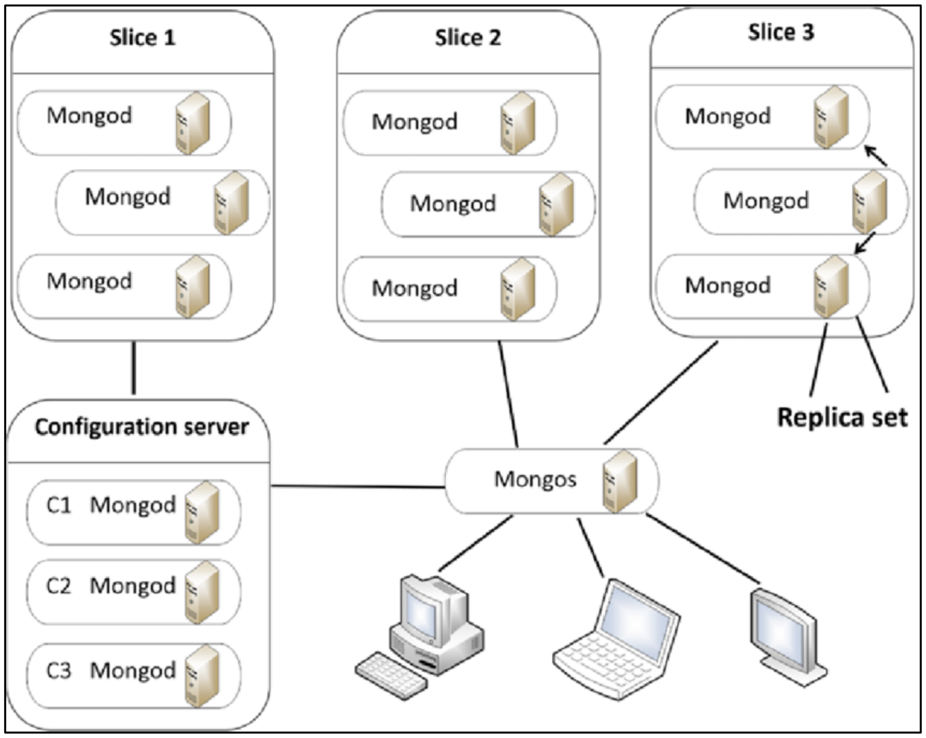


MongoDB Architecture

The architecture of MongoDB is based on a collection of independent nodes that work together to form a distributed system.
The key components of MongoDB architecture are:

- **Shard**: A shard is a self-contained database that contains a portion of the data in a MongoDB cluster. Shards are used to distribute data across multiple nodes for horizontal scaling.
- **Replica Set**: A replica set is a group of MongoDB nodes that maintain identical copies of data, allowing for high availability and fault tolerance. In a replica set, one node is designated as the primary node and the others are secondary nodes.
- **Router**: The router component is responsible for routing incoming requests to the appropriate shard or replica set.
- **Configuration Server**: The configuration server component is responsible for storing metadata about the cluster, such as the location of data within the cluster and the status of each node.
- **Client**: The client component communicates with the router component to submit requests and receive responses.

In addition to these components, MongoDB also includes several other features and tools for working with big data, such as auto-sharding, indexing, and aggregation. MongoDB's architecture provides a scalable and highly available solution for working with big data, enabling organizations to easily manage and analyze large amounts of data.



Mongod and Mongos

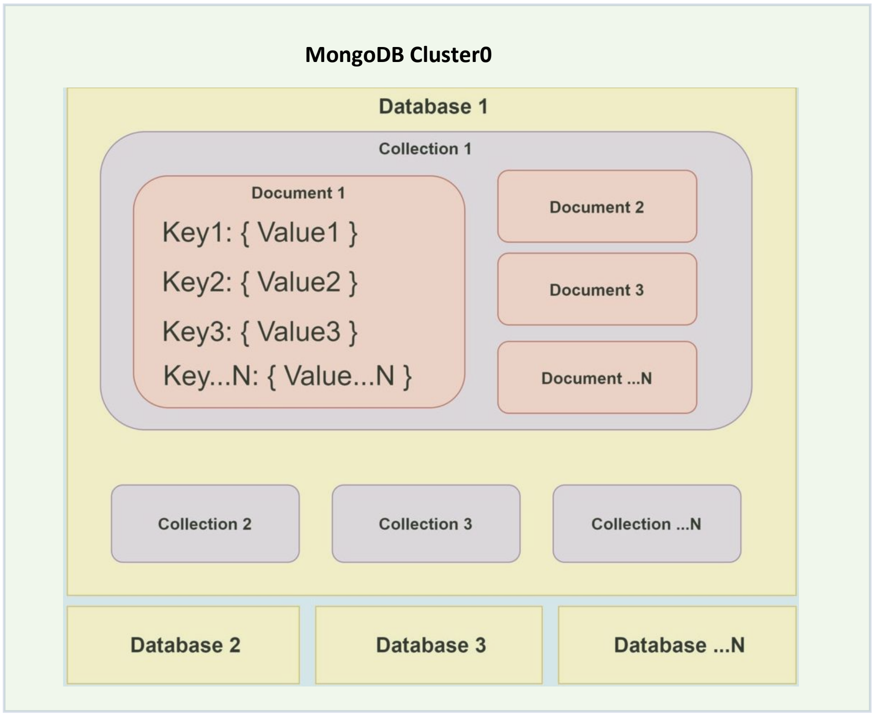
Mongod: Mongod is the component that runs on each node in a MongoDB cluster. It is responsible for managing the data and providing access to the data. Each mongod instance handles one or more databases and runs as a standalone process on a server.

Mongos: Mongos is the component that acts as a router for a MongoDB cluster. It is responsible for routing client requests to the appropriate shard or replica set in the cluster. The mongos process runs as a standalone process on a server, and multiple instances can be run for redundancy and load balancing.

In summary, Mongod manages the data and provides access to it, while Mongos acts as a router to distribute client requests to the appropriate shard or replica set in the cluster. These two components work together to provide a scalable and flexible solution for managing large amounts of data in a MongoDB cluster.

Project -> Cluster -> Database -> Collection -> Document

- A MongoDB **project** refers to a container for managing databases, users, roles, and other resources.
- A MongoDB **cluster** refers to a group of MongoDB servers that work together to provide a single, logical database. The cluster provides high availability, horizontal scaling, and geographic distribution of data.
- A MongoDB **database** is a container for collections and is made up of one or more collections.
- A MongoDB **collection** is a group of MongoDB documents. It is a way of grouping together related information and organizing it within a database.
- A MongoDB **document** is a single unit of data within a MongoDB collection, which is made up of key-value pairs. This is visually similar to a Python Dictionary OR a JSON Object but is stored as a BSON (Binary JSON).



MongoDB document limitations

Each MongoDB document has the following limitations:

Document size: The maximum size of a MongoDB document is 16 MB, which is the maximum size of a BSON (Binary JSON) document.

- Document structure: MongoDB documents are unstructured, which means that each document in a collection can have a different structure. However, this can lead to data consistency issues if proper validation is not implemented.
- Nested documents: MongoDB supports nested documents, but there is a limit to the number of levels of nesting that can be used. Additionally, deep nesting can lead to performance issues, so it's important to carefully consider the data structure before storing large amounts of nested data.
- Field names: Field names in MongoDB are limited to a length of 120 bytes, and cannot contain the '.' character, which is reserved for use in dot notation.
- Data types: MongoDB supports a wide range of data types, but not all data types are equally efficient. For example, arrays and objects can be slow to query compared to scalar data types.

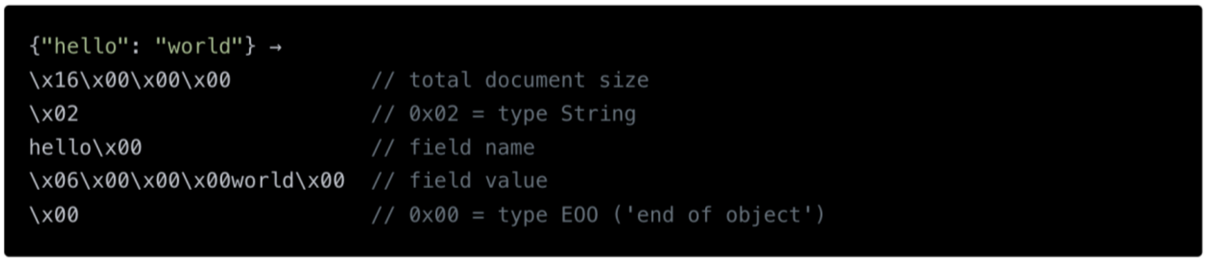
While MongoDB documents offer a flexible data model, it's important to carefully consider the data structure and data types when designing a MongoDB solution. Proper validation and indexing can help mitigate many of these limitations, and allow for efficient and effective use of MongoDB for big data projects.

JSON vs BSON

JSON (JavaScript Object Notation) and BSON (Binary JSON) are both data interchange formats used to store and exchange data. However, there are some key differences between them:

- Binary format: BSON is a binary encoded serialization of JSON data. This means that BSON data is stored in a binary format, while JSON data is stored in a text-based format.
- Size: BSON is typically smaller in size than JSON, since it uses a binary encoding that is more efficient than the text-based encoding used by JSON.
- Data types: BSON supports a wider range of data types than JSON, including binary data, date/time values, and 64-bit integers.
- Performance: BSON is faster to serialize and deserialize than JSON, since it uses a binary encoding that is more efficient than the text-based encoding used by JSON.
- Interoperability: JSON is more widely used and supported than BSON, making it easier to interoperate with other systems.

Both JSON and BSON have their strengths and weaknesses, and the choice between them will depend on the specific requirements of a project. MongoDB uses BSON as its native data format, making it a good choice for working with big data stored in MongoDB.



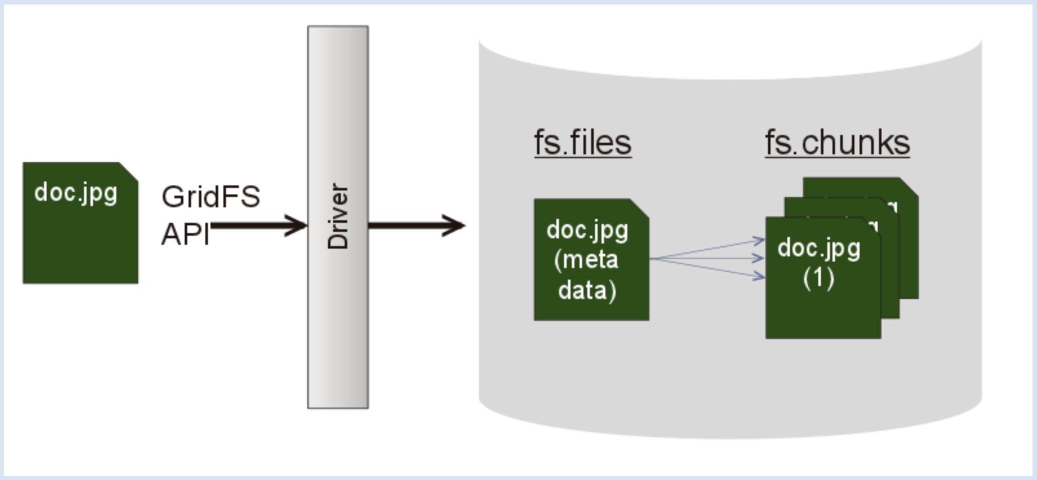
```
{"hello": "world"} →
\x16\x00\x00\x00            // total document size
\x02                        // 0x02 = type String
hello\x00                   // field name
\x06\x00\x00\x00world\x00   // field value
\x00                        // 0x00 = type EOO ('end of object')
```

Did you know?

GridFS is a specification in MongoDB for storing and retrieving large files (files that exceed the BSON-document size limit of 16 MB) and binary data. It works by dividing a large file into smaller chunks and storing each chunk as a separate document in a MongoDB collection. The chunks are then reassembled into the original file when retrieved.

The benefits of using GridFS in MongoDB include:

- *Large file support: GridFS enables MongoDB to store and retrieve large files, such as images, videos, and audio, that might not fit in a single MongoDB document.*
- *Scalability: GridFS enables data to be distributed across multiple nodes in a MongoDB cluster, providing the ability to scale horizontally as storage needs grow.*
- *Efficient retrieval: GridFS supports efficient partial retrieval of files, which is useful when only a portion of a large file is needed, such as when streaming video or audio.*
- *Atomic operations: GridFS supports atomic operations, ensuring that either all or none of the chunks of a file are stored in the database, providing a consistent and reliable method for storing and retrieving large files.*



Sharding

Sharding is a method of horizontally partitioning data across multiple servers in a MongoDB cluster. The goal of sharding is to scale out the database to handle increasing amounts of data and traffic.

In MongoDB, sharding is implemented at the collection level, which means that each shard contains a subset of the documents in a single collection. When a collection becomes too large to fit on a single server, it can be split across multiple shards.

The process of manually sharding a collection involves the following steps:

- Choose the shard key: The shard key is the field in the documents that will be used to distribute the data across the shards. The shard key should have a well-defined distribution of values, so that the data can be evenly distributed across the shards.
- Enable sharding: To enable sharding, you must run the sh.enableSharding() method, which enables sharding for the database.
- Shard the collection: Next, you shard the collection by running the sh.shardCollection() method, which splits the collection into multiple shards based on the shard key.
- Configure chunk distribution: MongoDB uses chunk distribution to manage the placement of documents in the shards. The chunk distribution algorithm ensures that each shard has a similar number of chunks, which helps to ensure that the data is evenly distributed across the shards.

With sharding, MongoDB provides a scalable and flexible way to manage large amounts of data and to handle increasing amounts of traffic. By distributing the data across multiple servers, sharding helps to ensure that the database can scale horizontally to meet the growing needs of your application.
In MongoDB Atlas, sharding is managed automatically and transparently, making it easier for you to scale your database as your data grows.

MongoDB Atlas vs On-Premises Setup

MongoDB Atlas and an on-premise setup are two different ways of deploying MongoDB.

MongoDB Atlas is a fully managed cloud service offered by MongoDB, Inc. It provides users with the ability to run MongoDB databases in the cloud without having to manage any underlying infrastructure. Some benefits of using MongoDB Atlas include:

- Scalability: MongoDB Atlas allows users to easily scale their databases up or down as needed, without having to manage any underlying infrastructure.
- High availability: MongoDB Atlas provides built-in high availability, ensuring that your data is always accessible and your applications stay online.
- Security: MongoDB Atlas includes a number of security features, such as encryption-at-rest and network isolation, to help keep your data secure.
- Easy management: MongoDB Atlas provides a user-friendly web interface that makes it easy to manage and monitor your databases, without requiring any operational expertise.

On the other hand, an on-premise setup refers to installing and running MongoDB on your own servers or infrastructure. Some benefits of an on-premise setup include:

- Control: An on-premise setup gives you complete control over your data and infrastructure, allowing you to make custom configurations and fine-tune your setup as needed.
- Cost savings: Running MongoDB on-premise can be less expensive than using a fully managed cloud service, especially for large-scale deployments.
- Integration: An on-premise setup can provide tighter integration with other systems and applications that you have running on your own infrastructure.

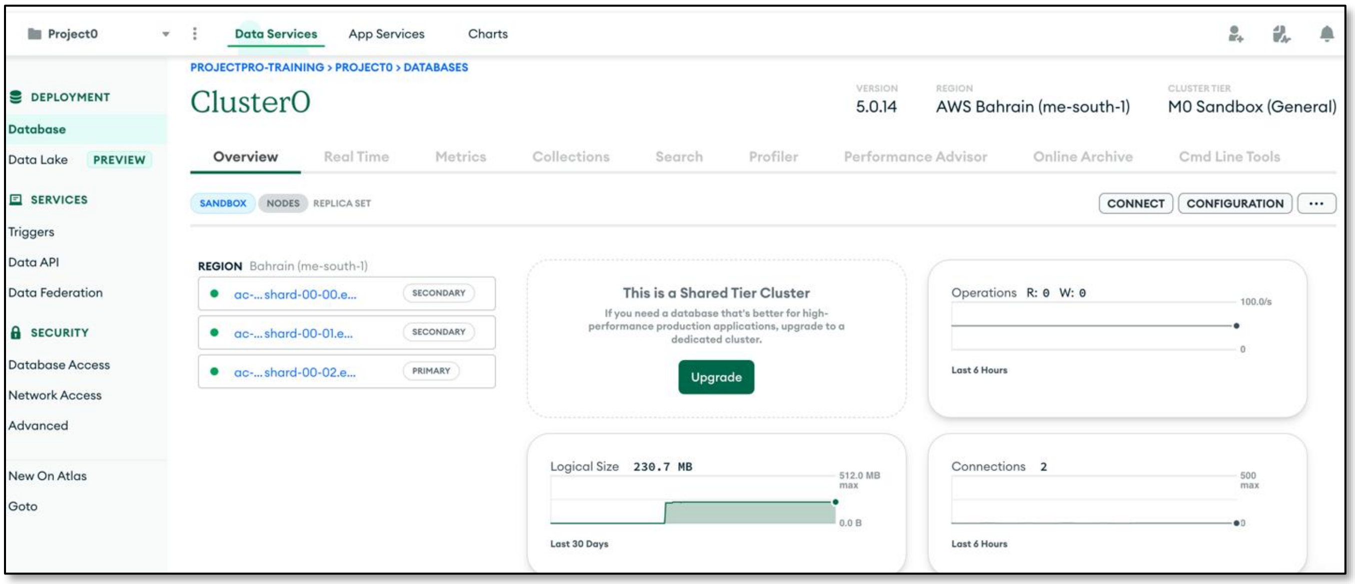
The choice between MongoDB Atlas and an on-premise setup will depend on your specific needs and requirements. For example, if you need the ability to scale quickly, need the security provided by a managed cloud service, or lack operational expertise, then MongoDB Atlas might be the better choice. However, if you require more control over your infrastructure, are looking to save on costs, or need tight integration with other systems, then an on-premise setup might be a better option.

Setting up MongoDB Atlas Cluster

Setting up a MongoDB Atlas cluster involves the following steps:

- Sign up: Start by signing up for MongoDB Atlas, which is free and only requires an email address.
- Create a project: Next, create a project in MongoDB Atlas, which is a logical container for your databases and resources.
- Define cluster: Once you have a project, you can create a new cluster. Choose the region and data center where you want to host your cluster, and select the appropriate cluster tier based on your performance and storage requirements.
- Configure security: In the next step, you'll need to configure security for your cluster. You can choose to enable IP whitelisting, create a MongoDB user with appropriate access privileges, and enable network peering.
- Launch cluster: Once you've completed these steps, you can launch your cluster. MongoDB Atlas will then create and configure the necessary infrastructure to host your cluster.
- Connect to cluster: Finally, you can connect to your cluster by following the connection instructions provided by MongoDB Atlas. You can connect using the MongoDB shell, a MongoDB driver, or a MongoDB client, such as Studio 3T.
- Load data: Once you have connected to your cluster, you can start loading data into your MongoDB databases. You can do this by inserting documents directly into collections or by using the MongoDB Connector for BI to integrate with existing data sources.

With these steps, you should be able to quickly set up a MongoDB Atlas cluster and start using it to store and manage your data.



PROJECTPRO-TRAINING > PROJECT0 > DATABASES

Cluster0

VERSION 5.0.14

REGION AWS Bahrain (me-south-1)

CLUSTER TIER M0 Sandbox (General)

REGION Bahrain (me-south-1)

ac-...shard-00-00.e... SECONDARY

ac-...shard-00-01.e... SECONDARY

ac-...shard-00-02.e... PRIMARY

This is a Shared Tier Cluster

If you need a database that's better for high-performance production applications, upgrade to a dedicated cluster.

Operations R: 0 W: 0 — 100.0/s, 0

Logical Size 230.7 MB — 512.0 MB max, 0.0 B

Connections 2 — 500 max, 0

Standard Connection vs DNS Seed List

A standard connection string is used to connect directly to a MongoDB server or replica set. The connection string specifies the hostname or IP address of the server and the port number, along with optional parameters such as the database name, username, and password. This type of connection is used when you have a specific MongoDB instance or replica set that you want to connect to and you want to control the connection details.

An example of a standard MongoDB connection string for a single MongoDB server is:

mongodb://hostname:port/database_name

A **DNS seed list** is a list of hostnames that are used to discover the members of a MongoDB replica set or sharded cluster. The DNS seed list is used to provide high availability and automatic failover in the event that a MongoDB server goes down.

In a MongoDB replica set, each member in the set is assigned a unique hostname. The hostnames are registered in a DNS server and are used to create the DNS seed list. The client uses the DNS seed list to connect to the replica set and to discover the current primary node. The primary node is the node that accepts writes, and the secondary nodes are used for reading and for failover in the event that the primary node goes down.

In a MongoDB sharded cluster, each shard is assigned a unique hostname, and the hostnames are registered in a DNS server to create the DNS seed list. The client uses the DNS seed list to connect to the sharded cluster and to discover the location of the data. This type of connection is useful when you want to abstract the details of the replica set members from the client and allow the replica set to manage the connection details.

The format for connecting to a MongoDB replica set using a DNS seed list is as follows:

mongodb+srv://<hostname>/<database>

The <hostname> is the hostname of the seed node and <database> is the name of the database you want to connect to. The mongodb+srv:// prefix indicates that the connection uses the SRV protocol, which is used to discover the replica set members through DNS.

Here is an example of a connection string using a DNS seed list:

mongodb+srv://cluster0.mongodb.net/test

In this example, cluster0.mongodb.net is the hostname of the seed node and test is the name of the database. When connecting to the seed node, MongoDB will use the SRV records associated with the hostname to discover the members of the replica set.

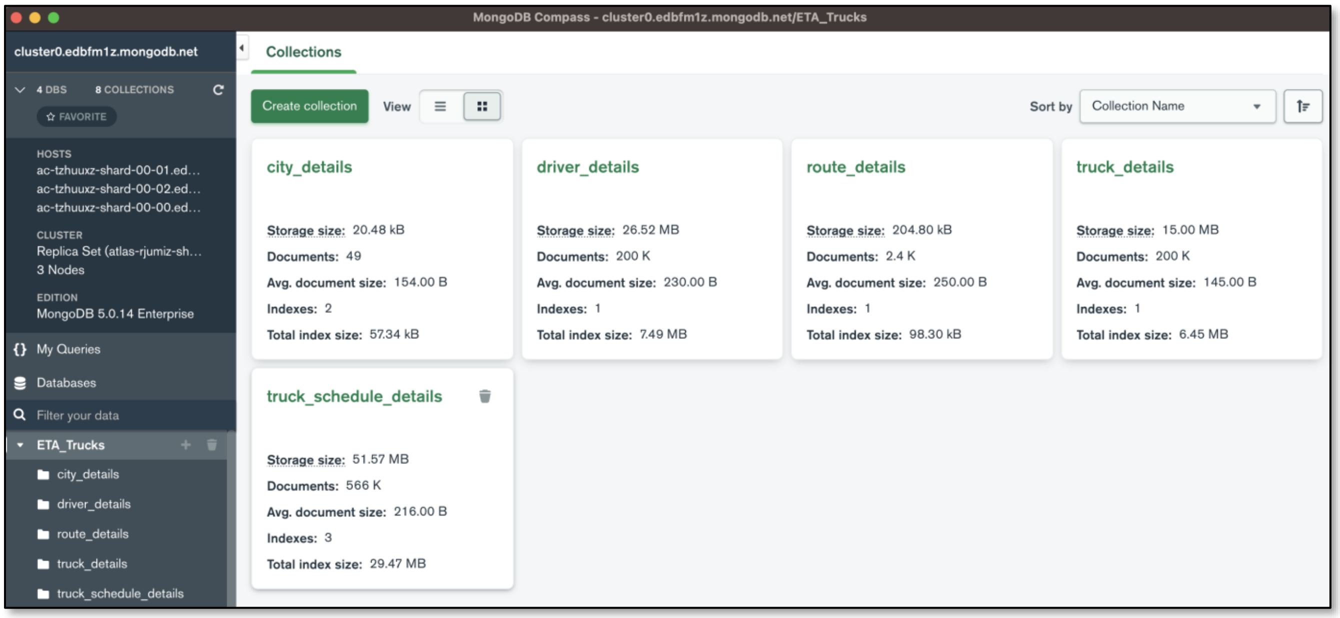
MongoDB Compass

MongoDB Compass is a graphical user interface (GUI) tool that allows you to interact with and visualize MongoDB databases. It provides a way to manage data, run queries, visualize data structures, and more.

To load data into a MongoDB database using MongoDB Compass, you can use the following steps:

1. Connect to the database: Open MongoDB Compass and connect to the database you want to load data into. You can do this by entering the connection string or selecting a previously used connection from the list.
2. Create a new collection: Once connected, create a new collection in the database where you want to load the data. You can do this by clicking the "Create Collection" button in MongoDB Compass.
3. Import data: To import data, you can use the "Import Data" option in MongoDB Compass. This opens a dialog that allows you to select the data file and choose the import method (e.g., CSV, JSON, BSON, etc.).
4. Preview and adjust data: After importing the data, you can preview the data in MongoDB Compass to ensure it was imported correctly. You can also adjust the data as needed using the MongoDB Compass UI.
5. Save data: Finally, click the "Save" button to save the imported data to the database.

This process provides a simple and straightforward way to load data into a MongoDB database using MongoDB Compass, without the need for writing any code or using the command-line interface.



HOSTS
ac-tzhuuxz-shard-00-01.ed...
ac-tzhuuxz-shard-00-02.ed...
ac-tzhuuxz-shard-00-00.ed...

CLUSTER
Replica Set (atlas-rjumiz-sh...
3 Nodes

EDITION
MongoDB 5.0.14 Enterprise

	city_details	driver_details	route_details	truck_details	truck_schedule_details
Storage size:	20.48 kB	26.52 MB	204.80 kB	15.00 MB	51.57 MB
Documents:	49	200 K	2.4 K	200 K	566 K
Avg. document size:	154.00 B	230.00 B	250.00 B	145.00 B	216.00 B
Indexes:	2	1	1	1	3
Total index size:	57.34 kB	7.49 MB	98.30 kB	6.45 MB	29.47 MB

Use-case depicted in this project

In this project we will simulate a logistical company use case and analyze the historical truck scheduled trips data to infer critical insights and accordingly prepare actionable items for the business. We will use the PyMongo library provided by MongoDB to perform aggregations on the various datasets.

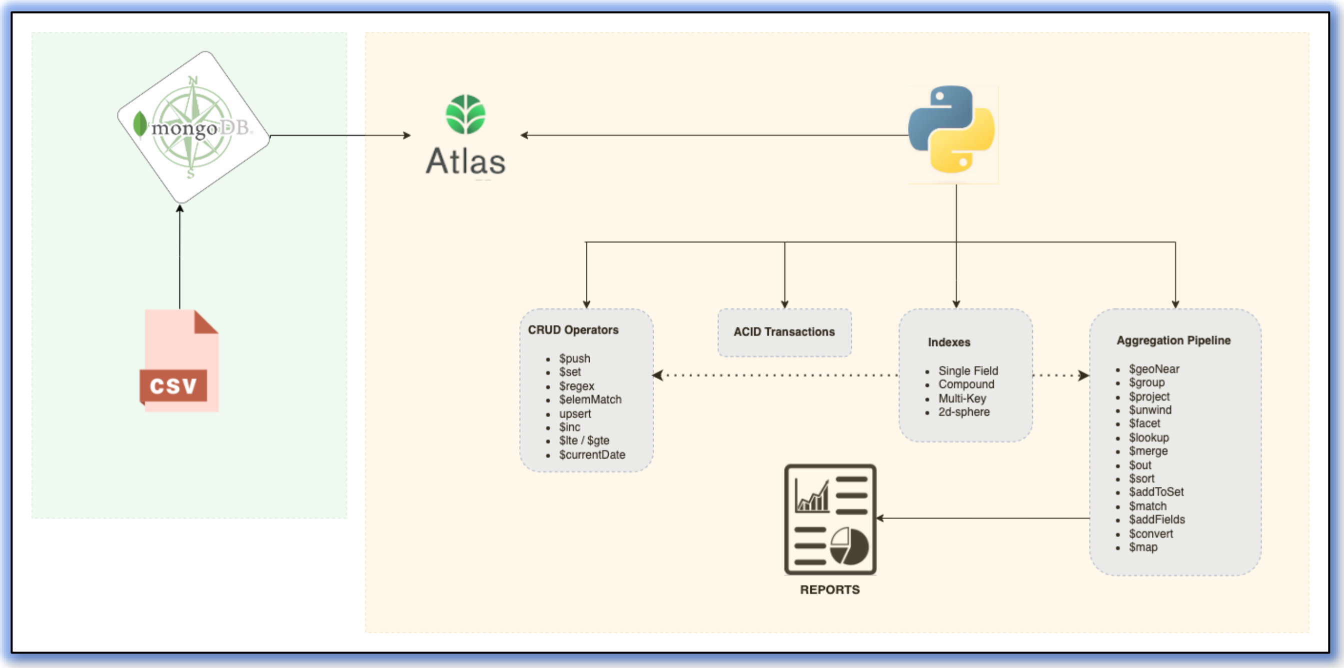
Dataset Description

We will use the transportation dataset, which contains information about the driver details, route details, city details, truck details, etc. The dataset has various tables in the manner given below. While importing the data using Compass, change the data types of the fields as mentioned alongside the field names:

1. **driver_details:** The driver details table contains information about 1300 truck drivers involved in the study across nine fields.
   a. 'driver_id' (String): unique identification for each driver
   b. 'name' (String): name of the truck driver
   c. 'gender' (String): gender of the truck driver
   d. 'age' (Int32/Number): age of the truck driver
   e. 'experience' (Int32/Number): experience of the truck driver in years
   f. 'driving_style' (String): driving style of the truck driver, conservative or proactive
   g. 'ratings' (double): average rating of the truck driver on a scale of 1 to 10
   h. 'vehicle_no' (String): the number of the driver's truck
2. **truck_details:** The truck details table contains information about 1300 trucks in the study across five fields.
   a. 'truck_id' (String): the unique identification number of the truck
   b. 'truck_age' (Int32/Number): age of the truck in years
   c. 'load_capacity_pounds' (Int32/Number): loading capacity of the truck in years
   d. 'mileage_mpg' (Int32/Number): mileage of the truck in miles per gallon
   e. 'fuel_type' (String): fuel type of the truck
3. **city_details:** The city details table contains the information for 49 cities.
   a. 'city_id' (Int32/Number): the unique identification number of the city
   b. 'city_name' (String): name of the city
   c. 'lat' (double): latitude
   d. 'lon' (double): longitude
4. **route_details:** The route details table contains information for 2352 different routes followed by the trucks
   a. 'route_id' (String): the unique identifier of the routes
   b. 'origin_city_id' (String): the city identification number for the origin city
   c. 'destination_city_id' (String): the city identification number for the destination
   d. 'distance(Miles)' (double): Distance between the origin and destination cities
   e. 'average_hours' (double): average time needed to travel from the origin to the destination in hours
   f. 'destination_lat'/'destination_lon' (double): Destination Latitude and Longitude
   g. origin_lat'/origin_lon' (double): Origin Latitude and Longitude

5. **truck_schedule_data:** The truck schedule data contains historical information of the trucks scheduled in the period of two months 2019-01-01 to 2019-02-28 and if arrival was delayed. It contains 16600 rows
   a. 'truck_id' (String): the unique identifier of the truck
   b. 'route_id' (String): the unique identifier of the route
   c. 'date' (Date): departure DateTime of the truck
   d. 'planned_eta' (Date): estimated arrival DateTime of the truck
   e. 'delay' (String): binary variable if the truck's arrival was delayed, 0 for on-time arrival and 1 for delayed arrival

Project flow



Here are some of the key learning points of this project.

- **Understand CRUD Operators in MongoDB**
- **Perform ACID Transactions using PyMongo**
- **Understanding the importance of Indexes in MongoDB**
- **Understanding the various Geospatial queries in MongoDB**
- **Generate Analytics and Reports using Aggregation Pipeline**

PyMongo Installation

PyMongo is a Python driver for MongoDB that provides a convenient way to interact with MongoDB databases from within Python. It allows developers to write Python code to query and modify data stored in MongoDB, and to map MongoDB data to Python objects for easier manipulation.

PyMongo provides a high-level API for working with MongoDB that abstracts away many of the underlying details of the database. With PyMongo, you can connect to a MongoDB instance, create and manage collections and documents, perform CRUD operations (Create, Read, Update, Delete), and perform complex queries and aggregations.

Additionally, PyMongo supports MongoDB's built-in features such as sharding and replica sets, allowing you to work with a highly available and scalable database setup.

PyMongo is widely used by Python developers due to its ease of use, comprehensive documentation, and active development community. If you're looking to integrate MongoDB with your Python application, PyMongo is an excellent choice.

```
#install libraries
!pip install python-dotenv
!pip install pymongo[srv]

#import necessary modules
from dotenv import load_dotenv
from datetime import datetime
from pprint import pprint
import os
import bson
import pymongo

load_dotenv()

#connect to mongoDB
#get the CONN_STRING env variable from .env file
client = pymongo.MongoClient(os.environ.get("CONN_STRING"))

#connect to Database
db=client['ETA_Trucks']
```

Did you know?

A .env file is a simple text file that contains environment variables. Environment variables are key-value pairs that can be used to store configuration data for an application.

The .env file is commonly used in development and test environments to store sensitive information such as database credentials, API keys, and other configuration data that should not be stored in the source code repository.

When an application is started, it can read the .env file and use the values stored in the environment variables to configure itself. This makes it easy to switch between different configurations for different environments (e.g. development, test, and production).

Here's an example of a .env file:

```
CONN_STRING= <mongo_connection_string>
```

To use the environment variables in an application, you can use a library like dotenv to read the .env file and set the environment variables in your application's process.

```
import os

from dotenv import load_dotenv

load_dotenv()

client = pymongo.MongoClient(os.environ.get("CONN_STRING"))
```

Both os.getenv and os.environ.get in Python are used to retrieve the value of an environment variable.os.getenv returns None if the specified environment variable does not exist. if you just need to retrieve the value of an environment variable or check if it exists, os.getenv may be more convenient. If you need to retrieve the value of an environment variable and provide a default value in case it does not exist, os.environ.get is more convenient.

Primary vs Secondary Nodes

By default, MongoDB reads data from the primary node in a replica set. The primary node is responsible for all writes to the database and maintaining consistency across the replica set. Secondary nodes are used for data redundancy and to provide additional read capacity, but reads are typically directed to the primary node.

However, you can configure your MongoDB application to read from secondary nodes if desired, using a feature called "read preferences". This can improve read performance by distributing read workload across multiple nodes and can also provide automatic failover to a secondary node in the event that the primary node becomes unavailable.

It is important to understand the trade-offs of reading from secondary nodes, including potential consistency issues and a higher latency due to the need to replicate data from the primary node to the secondary nodes.

MongoDB writes are only accepted by the primary node in a replica set. The primary node then replicates the writes to all secondary nodes, ensuring consistency across the entire replica set.

Writing directly to secondary nodes is not allowed as it would compromise data consistency, as secondary nodes may not have the most recent data and may have conflicting data.

By having all writes go through the primary node, MongoDB can guarantee data consistency and atomicity across the entire replica set.

Admin and Local DB

In MongoDB, there are two special databases named "admin" and "local".

"admin" database: This is a built-in database in MongoDB that provides administrative features such as authentication and authorization. It's used to manage users, roles, and permissions for other databases within the same MongoDB instance.

"local" database: This is another built-in database that is used for storing metadata and configuration information specific to a particular MongoDB instance. The "local" database is used to store information such as the state of replication and the operation log for the current replica set. This database is not shared across replica sets and is only accessible from the instance where it resides.

Both "admin" and "local" databases are essential for the proper functioning of a MongoDB instance, and should not be modified or deleted without proper understanding and caution.

Oplog in MongoDB

The "oplog" (operation log) in MongoDB is a special capped collection in the "local" database that acts as a record of all operations performed on the database. It's used to maintain consistency in a MongoDB replica set and support replication.

The oplog contains a record of all write operations, such as insert, update, and delete, along with the associated timestamp. Secondary nodes in a replica set use the oplog to apply the same write operations in the same order as the primary node, thus ensuring that they have an up-to-date copy of the data.

The oplog is also used by the MongoDB tail-based change stream feature, which allows you to watch for real-time changes to data within a MongoDB instance.

It's important to note that the oplog has size limitations, and must be properly sized based on your database's write load. In addition, the oplog can use a significant amount of disk space, so it's important to monitor its usage and periodically clean out old entries.

CRUD operations in PyMongo

These are the queries for basic read operations

```
#find one
filter={'name':{"$eq":"Alabama"}}
db.city_details.find_one(filter)

#find using _id
filter={"_id":bson.ObjectId('63cf28c750cce1d3ca60f9fe')}
db.city_details.find_one(filter)
```

```
#find - implicit And - find most efficient diesel trucks with age greate
than 12
filter={"fuel_type":'diesel',
        "truck_age":{"$gt":12}}

projection={"_id":0,"fuel_type":0}

sort_seq=[("truck_age",pymongo.ASCENDING),
          ("load_capacity_pounds",pymongo.DESCENDING),
          ("mileage_mpg",pymongo.DESCENDING)]

result=db.truck_details.find(filter, projection).sort(sort_seq).limit(5)

for i in result:
  pprint(i)
  print("\n")
```

$and is used to match all the conditions in an array. A document is returned only if all the conditions are true. **$or** is used to match any of the conditions in an array. A document is returned if any of the conditions are true.

$gte (greater than or equal to) is a comparison operator used in a query document to match documents with fields greater than or equal to the specified value.

$lte (less than or equal to) is a comparison operator used in a query document to match documents with fields less than or equal to the specified value.

In PyMongo, filter and projection are used to retrieve a subset of documents based on certain criteria.

Filter is used to specify which documents should be retrieved from the collection. It is specified as a query document (dictionary) that matches the fields in the documents.

Projection is used to specify which fields should be included or excluded from the retrieved documents. It is specified as a query document that specifies the inclusion or exclusion of fields.

```
#count no.of documents - young proactive male drivers who are reliable for
long distances

filter={
    "$and":[
        {"gender":"male"},
        {"experience":{"$gt":10}},
        {"driving_style":{"$in": ["proactive"]}}
    ],
    "$or":[
        {"$and":[
            {"age":{"$gte":35}},
            {"age":{"$lte":45}}
        ]},
        {"ratings":{"$gt":4}}

    ]
}

print(db.driver_details.count_documents(filter))

sort_seq=[("ratings",-1), #-1 is for descending order
          ("age",1)]      # 1 is for ascending order
result=db.driver_details.find(filter).sort(sort_seq).limit(5)

for i in result:
  pprint(i)
  print("\n")
```

$regex is a MongoDB operator that performs regular expression matching on a specified field in a document. It can be used to search for a specific pattern in string data stored in your MongoDB collections. The operator returns all documents that have a match for the specified pattern in the specified field. To use $regex, you need to specify the field that you want to match and the regular expression pattern as a string value. The regular expression pattern can include special characters and symbols that define the pattern to match. For example, you can use the symbol "^" to match the start of a string, "$" to match the end of a string, and "." to match any character. The $regex operator performs a case-sensitive match by default. You can also use the $options operator to modify the behavior of the regular expression match.

For example, you can use $options: "i" to make the match case-insensitive. $regex is a very powerful operator that allows you to perform complex string matching in MongoDB. It can be used in combination with other query conditions to filter documents based on complex matching criteria.

```
#regex- first name 'Tyler'

filter={"name":{"$regex":"^Tyler .*", "$options":'ms'}}
result=db.driver_details.find(filter).limit(2)

# 'm' for caret(^)-startWith and $- endWith
# 's' for period(.) - all characters

for i in result:
  pprint(i)

#regex- last name 'Dean'

filter={"name":{"$regex":".* Dean$", "$options":'ms'}}
result=db.driver_details.find(filter).limit(2)

for i in result:
  pprint(i)
```

Update Operators-

$set is used to update a specific field in a document. It creates a new document if the field does not exist.

```
#update one
filter={"truck_id":"2f0705c938e545a0aa03c505ec97751d"}

update={
    "$set":
    {"fines":[
        {"date":datetime.utcnow(),
         "amount":500,
         "reason":"Speed Limit",
         "driver_id":"c3173876aeac4604863da23d0a90f6f3"}
         ]}}

db.truck_details.update_one(filter,update)
```

$push is used to add an item to an array field in a document.

```
#update to list
filter={"truck_id":"2f0705c938e545a0aa03c505ec97751d"}

update={
    "$push":
    {"fines":
     {"date":datetime.utcnow(),
      "amount":5000,
      "reason":"Red Light Crossed",
      "driver_id":"c3173876aeac4604863da23d0a90f6f3"}}}

db.truck_details.update_one(filter,update)
```

$inc is used to increment a numerical field by a specified value.

```
#update many - increment truck age for all trucks
filter={}
update={"$inc":{"truck_age":1}}
db.truck_details.update_many(filter,update)

#update many- decrement truck mileage
filter={"fuel_type":"gas"}
update={"$inc":{"mileage_mpg":-2}}
db.truck_details.update_many(filter,update)
```

Insert (Create) new records in a DB in a new collection

```
db2=client['Ecom']

db2.Quant.insert_many([
    {"item":"abc","quantity":41,"lastModified":datetime.strptime("2023-01-09 01:01:01", "%Y-%m-%d %H:%M:%S")},
    {"item":"def","quantity":56,"lastModified":datetime.strptime("2023-01-09 01:01:01", "%Y-%m-%d %H:%M:%S")},
    {"item":"xyz","quantity":78,"lastModified":datetime.strptime("2022-02-09 01:01:01", "%Y-%m-%d %H:%M:%S")},
    {"item":"jkl","quantity":67,"lastModified":datetime.strptime("2023-01-09 01:01:01", "%Y-%m-%d %H:%M:%S")}

]
)

db2.sales.insert_many([

{"_id":1,"item":"abc","price":8,"quantity":4,"date":datetime.strptime("2022-01-01 01:01:01", "%Y-%m-%d %H:%M:%S")},

{"_id":2,"item":"xyz","price":8,"quantity":4,"date":datetime.strptime("2022-02-01 01:01:01", "%Y-%m-%d %H:%M:%S")},

{"_id":3,"item":"jkl","price":10,"quantity":5,"date":datetime.strptime("2023-01-01 01:01:01", "%Y-%m-%d %H:%M:%S")},

{"_id":4,"item":"abc","price":12,"quantity":6,"date":datetime.strptime("2023-01-01 01:01:01", "%Y-%m-%d %H:%M:%S")},

{"_id":5,"item":"def","price":16,"quantity":8,"date":datetime.strptime("2023-01-01 01:01:01", "%Y-%m-%d %H:%M:%S")},

{"_id":6,"item":"jkl","price":10,"quantity":5,"date":datetime.strptime("2023-01-01 01:01:01", "%Y-%m-%d %H:%M:%S")},

{"_id":7,"item":"abc","price":10,"quantity":5,"date":datetime.strptime("2023-01-01 01:01:01", "%Y-%m-%d %H:%M:%S")}
])
```

Upsert is a flag in MongoDB that specifies whether to update a document or insert a new document if no match is found. By default, upsert is set to False, but can be set to True to insert a new document if no match is found.

```
#upsert
filter={"name":"Houston"}

update={"$set":{
    "name":"Houston",
    "city_id":50,
    "lat":29.74,
    "lon":-95.46}}

db.city_details.update_many(filter,update,upsert=True)
```

$elemMatch is used to match documents that contain an array field with at least one document that matches all the specified query criteria.

```
filter={
    "sales_list":{"$elemMatch":{"quantity":
                   {"$ne":4}, #not equal operator
                   "$and":[
                       {"date":{"$gte":datetime(2022,1,1)}},
                       {"date":{"$lte":datetime(2023,2,1)}}
                       ]}}}
result=db2.Inventory.find(filter)
for i in result:
  pprint(i)
  print('\n')
```

$currentDate is used to set a field to the current date and time.

```
filter={"item":"xyz"}
update={  "$inc":{
    "quantity": -7},
      "$push":{"sales_list":{
                 "item":"xyz",
                 "price":14,
                 "quantity":7,
                  "date":datetime.now()
                     }},
           "$currentDate":{"lastModified":True}}

db2.Inventory.update_many(filter,update)
```

COLLSCAN and IXSCAN

COLLSCAN and IXSCAN are query execution strategies used by MongoDB to determine the best way to perform a query. They are two of the possible stages in the query plan, and the choice of which strategy to use depends on the type of query, the size of the collection, and the availability of indexes.

COLLSCAN stands for Collection Scan and refers to a query execution strategy where MongoDB scans every document in the collection to find the matching documents. This is the most basic and simple query execution strategy and is used when no indexes are available for the query or when the query involves fields that are not indexed.

IXSCAN stands for Index Scan and refers to a query execution strategy where MongoDB uses one or more indexes to perform the query. This strategy is faster than a Collection Scan as it only needs to scan the indexed data, and it is used when there are one or more indexes available that can be used to satisfy the query.

In general, using indexes to perform a query is more efficient than a Collection Scan, so it's recommended to use the IXSCAN strategy whenever possible. To achieve this, you need to ensure that the appropriate indexes are available for your queries and that your queries are written in a way that allows MongoDB to use them.

Did you Know?

The explain() function in PyMongo is a method that provides detailed information about how MongoDB executed a query. It provides information about the query plan, such as the indexes used, the number of documents scanned, and the time it took to execute the query.

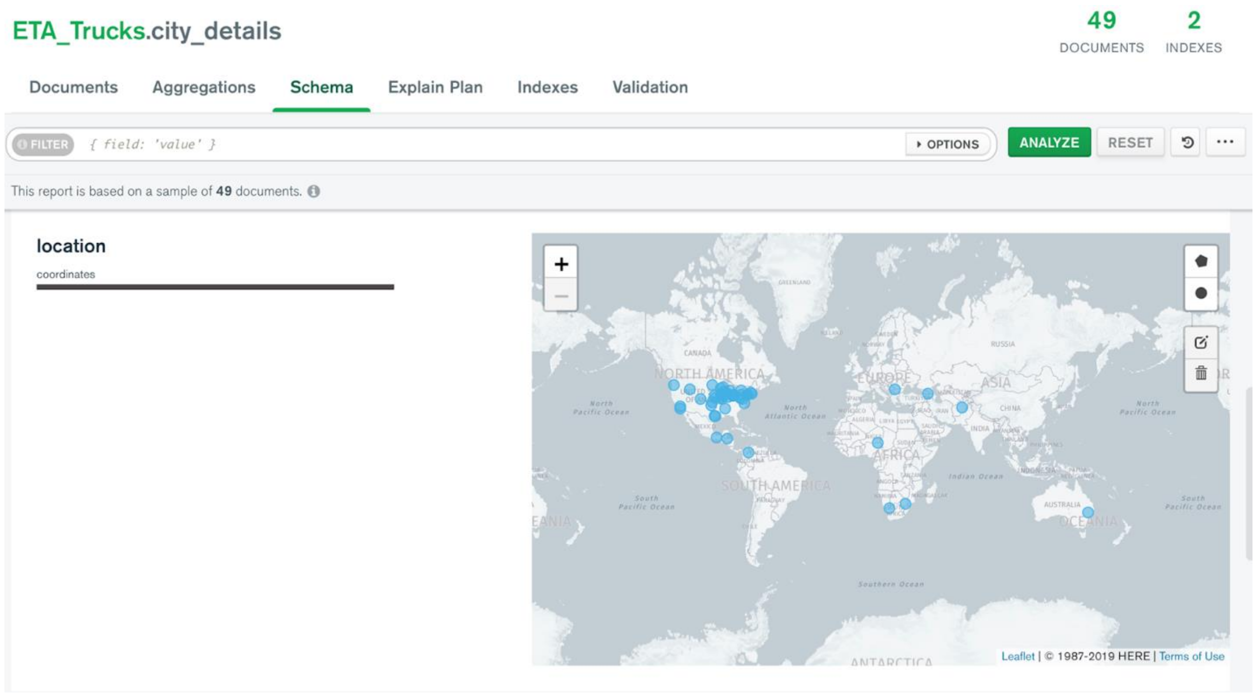
The explain() method returns a document that provides insight into how MongoDB executed the query. This information can be useful for optimizing query performance, as it allows you to see if the query is using the optimal indexes and if there are any performance bottlenecks.

It is important to note that explain() can only be used with find() operations, not with other methods such as update(), delete(), or aggregate(). The result of the explain() method can be quite complex, so it is recommended to only use it when necessary and to have a good understanding of how MongoDB works and what the different fields in the result document mean.

Indexes in MongoDB

Indexes in MongoDB are data structures that store a mapping of the values in a specific field or set of fields to the document they belong to. Indexes are used to improve the query performance by allowing MongoDB to quickly locate the relevant data based on the values specified in the query. The use of indexes allows MongoDB to avoid performing a full collection scan to retrieve the documents, which can be time-consuming and slow down the performance of the system. MongoDB supports several types of indexes including single-field, compound, and geospatial indexes, as well as text and hashed indexes. However, it's important to keep in mind that creating indexes comes at a cost in terms of disk space and write performance.

- Single-Field Index: Indexes a single field in the documents, useful for queries that match on a single field.
- Compound Index: Indexes multiple fields, allowing for queries that match on multiple fields.
- Text Index: Indexes the text content of the documents for full-text search.
- Geospatial Index: Indexes the geospatial data in the documents, allowing for location-based search and queries.



This report is based on a sample of 49 documents.

- Hashed Index: Hashes the values in a single field, allowing for efficient equality queries, but not range queries.
- Sparse Index: An index that only contains entries for documents that have the indexed fields.
- TTL (Time-To-Live) Index: An index that automatically removes documents based on the values in a specific field.

ACID Transactions

ACID stands for Atomicity, Consistency, Isolation, and Durability, and is a set of properties that define the behavior of a database transaction.

- Atomicity: A transaction is considered atomic if it is indivisible and irreducible, meaning either all the operations within the transaction are executed, or none of them.
- Consistency: The database should move from one consistent state to another after the transaction is executed, ensuring that the integrity of the data is maintained.
- Isolation: The execution of transactions should be isolated from each other, meaning that the effects of one transaction should not be visible to other transactions until it has been committed.
- Durability: The database should persist the changes made by a committed transaction, even in the event of a crash or power loss. This is usually achieved by writing the changes to a permanent storage, like disk or flash storage.

MongoDB supports ACID (Atomicity, Consistency, Isolation, Durability) transactions within a single replica set, but not across multiple replica sets or sharded clusters. In PyMongo, ACID transactions can be performed within a single session by using the with_transaction method. This method ensures that all the operations within the transaction are executed atomically, either all or none of them. To perform ACID transactions, the replica set must have a write concern of "majority". The use of ACID transactions allows for consistent and reliable data modification, even in the event of failures or crashes.

Aggregation Framework

The Aggregation framework in MongoDB is a way to perform operations on the data stored in a collection, to get the results in the form of aggregated values, such as total count, average, minimum, maximum, sum, etc. It provides a set of operations, such as grouping, filtering, transforming, and summarizing the data. The result of these operations is returned as an aggregation pipeline, which is a series of processing stages, each of which transforms the data. The stages can be used to filter, group, sort, limit and project the data. Aggregation framework provides more flexible and powerful data processing capabilities than the basic query and update operations.

The Aggregation framework in MongoDB provides a way to process and summarize data in a flexible, scalable manner. The following is a comprehensive explanation of each of the Aggregation operators:

$lookup: Joining documents in MongoDB is a bit different than in traditional relational databases. MongoDB is a NoSQL database and doesn't support joins as in relational databases. To perform a join-like operation, you must retrieve the documents you need and process the results on the client side. Alternatively, you can denormalize your data and store related documents within a single document. This operator joins two collections based on the specified conditions and returns documents that include both the joined data and the original data.

$merge: This operator merges the results of a pipeline with another collection.

```
stage_1={"$lookup":
     {
     "from":"sales",
     "localField":"item",
     "foreignField":"item",
     "as":"sales_list"}
    }

stage_2={
        "$unset":"sales_list._id"
            }

stage_3= {"$merge":{
        "into":{"db":"Ecom","coll":"Inventory"}
    }}

results=db2.Quant.aggregate([stage_1,stage_2,stage_3])

for i in results:
  pprint(i)
  print("\n")
```

$map: Applies an expression to each item in an array and returns an array with the applied results.

```
stage_1= {
      "$set": {
         "VAT": {
           "$map": {
             "input": "$sales_list",
             "as": "grade",
             "in": { "$multiply": ["$$grade.price", 0.05 ] }
           }
         }
      }
   }

result=db2.Inventory.aggregate([stage_1])

for i in result:
  pprint(i)
```

$group: This operator groups the documents by specified fields and applies various aggregate functions like sum, average, count etc. to calculate a summary value for each group.

$facet: This operator allows you to specify multiple sub-pipelines to process different sections of the data in parallel and return the results.

```
# find Avg age of all drivers

stage_1= {'$group': {'_id': None,
                     'avg_Age': {'$avg': '$age'}
                     }
          }

r=db.driver_details.aggregate([stage_1])

for i in r:
  pprint(i)

# find Avg age of all drivers by All documents, Gender, driving style in
one single pipeline

facet_stage={'$facet': {
            'AVG_AGE_ALL': [{'$group': {'_id': None,
                                        'avg_Age': {'$avg': '$age'},
                                        'total_count': {'$sum': 1}}}],

            'AVG_AGE_GENDER': [{'$group': {'_id': '$gender',
                                           'avg_Age': {'$avg': '$age'},
                                           'total_count': {'$sum': 1}}}],

            'AVG_AGE_STYLE': [{'$group': {'_id': '$driving_style',
                                          'avg_Age': {'$avg': '$age'},
                                          'total_count': {'$count': {}}}}]
              }}

r=db.driver_details.aggregate([facet_stage])

for i in r:
  pprint(i)
```

$match: This operator filters the documents to pass only those documents that match the specified condition.

$addFields: This operator adds new fields to the documents or modifies existing fields.

$project: This operator modifies the documents in the pipeline to return only the specified fields and computed values.

$unwind: This operator separates an array into individual documents for each element in the array.

$out: This operator writes the result of the aggregation pipeline to a new collection.

```
#join truck and driver details, add total_trucks and total_fines fields
and save to another collection

stage_1={"$lookup":
                {
                "from":"truck_details",
                "localField":"vehicle_no",
                "foreignField":"truck_id",
                "as":"trucks_assigned"}
              }

stage_2={"$unset":["vehicle_no","trucks_assigned._id"]}

stage_3={'$addFields': {'t_trucks': {'$size': '$trucks_assigned'}}}

stage_4={"$unwind":"$trucks_assigned"}

stage_5={'$addFields': {'t_fines': {'$cond': {'else': None,
                                   'if': {'$and': [{'$isArray':
'$trucks_assigned.fines'}]},
                                   'then': {'$size':
'$trucks_assigned.fines'}}}}}

stage_6={'$out': {'coll': 'driver_trucks', 'db': 'ETA_Trucks'}}

limit_stage={"$limit":5}

temp_stage={"$match":{"name":"Scott Hawkins"}}

result=db.driver_details.aggregate([stage_1, stage_2, stage_3, stage_4,
stage_5, limit_stage, stage_6])


#NOTE- $out and $merge are not optimal compute wise to store large number
of documents from an aggregation pipeline

for i in result:
  pprint(i)
  print("\n")
```

Did you Know?

The $merge operator in MongoDB is used in the aggregation pipeline to combine two collections into one, while retaining the original data in both collections. It allows you to merge documents from a source collection into a target collection and create a new collection or update documents in the target collection.

On the other hand, the $out operator in MongoDB writes the results of the aggregation pipeline to a new collection. It outputs the results of the aggregation to a new collection or replaces the data in an existing collection. The $out operator should always be the last stage in an aggregation pipeline.

In summary, the $merge operator is used to merge data from two collections, while the $out operator is used to write the results of an aggregation pipeline to a collection.

$geoNear: GeoJSON is a format for encoding geographic data structures, including points, lines, and polygonal shapes, using the JSON data format. In MongoDB, the format is used to store and query geographic information, such as coordinates and polygons, within a collection. The GeoJSON data can be indexed using the 2dsphere index, which enables efficient search and retrieval of data based on location and distance. Geospatial queries in MongoDB are used to search and analyze data based on location or geographic coordinates.

This operator is used to return documents sorted by their proximity to a specified GeoJSON point. Other query operators in MongoDB that support geospatial data with 2D and 3D indexing, which enhances the performance include:

- $geoWithin: Selects documents with geospatial data that are within a specified shape.
- $geoIntersects: Selects documents with geospatial data that intersect with a specified shape.
- $near: Selects documents with geospatial data that is closest to a specified point.
- $nearSphere: Selects documents with geospatial data that is closest to a specified point on a sphere.

```
#add a GeoJSON field
stage_1={'$addFields': {'location': {'coordinates': ['$lon', '$lat'],
'type': 'Point'}}}

stage_2={'$merge': {
            'into': 'city_details',
            'on': '_id',
            'whenMatched': 'replace'}
         }
result=db.city_details.aggregate([stage_1, stage_2])
for i in result:
  pprint(i)
  print("\n")
```

```
#create index for location
from pymongo import IndexModel, ASCENDING, DESCENDING

index1 = IndexModel([("location", "2dsphere")]) #for unique indexes, pass
unique=True

db.city_details.create_indexes([index1])
```

```
#find nearest centers for delivery using Spherical GeaSpatial Geometry
stage_1={'$geoNear': {
              'distanceField': 'dist.calculated',
              'distanceMultiplier': 0.001, #convert the output to kilometers
              'includeLocs': 'dist.location',
              'maxDistance': 800000, #outer radius to consider in meters
              'minDistance': 0, #inner radius to consider
              'near': {'coordinates': [-97.617134, 30.222296],
                       'type': 'Point'},
              'spherical': True}#specify to use spherical geometry
         }

stage_2={
    "$limit":10
}

result=db.city_details.aggregate([stage_1, stage_2])

for i in result:
  pprint(i)
  print("\n")
```

SAMPLE OUTPUT-

```
{'_id': ObjectId('63cf28c750cce1d3ca60fa11'),
 'city_id': 25,
 'dist': {'calculated': 211.57425184978044,
          'location': {'coordinates': [-95.538, 29.618], 'type': 'Point'}},
 'lat': 29.618,
 'location': {'coordinates': [-95.538, 29.618], 'type': 'Point'},
 'lon': -95.538,
 'name': 'Missouri City'}
```

$sort: This operator sorts the documents by specified fields in ascending(1) or descending(-1) order.

$addToSet: This operator returns an array of unique values for a specified field.

$convert: This operator converts data from one type to another, such as string to integer.

```
#total delayed journeys count

stage_1={'$group': {
            '_id': {
                '$toBool': {
                    '$convert': {
                        'input': '$delay',
                        'to': 'decimal'}}},
            'totalcount': {'$sum': 1}}
         }

stage_2={'$project': {
          '_id': 0,
          'is_delay': '$_id',
          'totalcount': '$totalcount'}
         }
result=db.truck_schedule_details.aggregate([stage_1, stage_2])

for i in result:
  pprint(i)
  print("\n")
```

OUTPUT-

```
{'is_delay': True, 'totalcount': 37113}
{'is_delay': False, 'totalcount': 528997}
```

```
#find the most congested routes

stage_1={'$match': {'delay': '1'}}

stage_2={'$group': {
            '_id': {
                'destination': '$destination',
                'origin': '$origin'},
            'route_id': {
                '$first': '$route_id'
                },
            'totalcount': {'$sum': 1}
            }
        }

stage_3={'$project': {
              '_id': 0,
              'route': '$_id',
              'route_id': '$route_id',
              'total_delays': '$totalcount'}
        }

stage_4={'$sort': {'total_delays': -1}}

limit_stage={"$limit":10}

result=db.truck_schedule_details.aggregate([stage_1, stage_2, stage_3,
stage_4, limit_stage])


for i in result:
  pprint(i)
  print("\n")
```

SAMPLE OUTPUT-

```
{'route': {'destination': 'Pennsylvania Furnace',
          'origin': 'South Dakota Park'},
 'route_id': 'South_Dakota_Park_Pennsylvania_Furnace_97381',
 'total_delays': 104}

{'route': {'destination': 'South Dakota Park',
          'origin': 'Pennsylvania Furnace'},
 'route_id': 'Pennsylvania_Furnace_South_Dakota_Park_1751',
 'total_delays': 104}
```

```
# find which months have the most delays and total routes affected each
month for past two years

stage_1={'$match': {'delay': '1'}}

stage_2={'$group': {
            '_id': {
                'month': {'$month': '$date'},
                'year': {'$year': '$date'}
                },
            'routes': {'$addToSet': '$route_id'},
            'total_delays': {'$sum': 1}
            }
         }

stage_3={'$project': {
              '_id': 0,
              'month_year': '$_id',
              'total_delays': '$total_delays',
              'unique_routes_affected': {'$size': '$routes'}
              }
         }

stage_4={'$sort': {'total_delays': -1}}

result=db.truck_schedule_details.aggregate([stage_1, stage_2, stage_3,
stage_4])


for i in result:
  pprint(i)
  print("\n")
```

SAMPLE OUTPUT-

```
{'month_year': {'month': 1, 'year': 2017},
 'total_delays': 3740,
 'unique_routes_affected': 1404}

{'month_year': {'month': 3, 'year': 2017},
 'total_delays': 2495,
 'unique_routes_affected': 1208}
```

Summary

- In this training we began with the notion of SQL vs NoSQL and discussed the advantages of a Document based Data Model.
- We understood the overall architecture of MongoDB along with where it stands under the CAP Theorem.
- Various components of data storage in MongoDB were discussed.
- We observed the key differences between On-Premise setup and Atlas service.
- The Database was accessed via Compass Client and PyMongo Drivers.
- Different CRUD operations were discussed with practical use cases.
- We understood the advantages of using Indexes in MongoDB.
- ACID Transactions were performed using PyMongo.
- Hands-on experience with Geospatial queries and location capabilities in MongoDB.
- Generated analytics and complex reports using Aggregation Pipeline.