MongoDB

Email ahammadmejbah@gmail.com | GitHub @BytesOfIntelligences | LinkedIn Mejbah Ahammad
Website Bytes of Intelligence | YouTube BytesofIntelligence | ResearchGate Mejbah Ahammad
Phone +8801874603631 | Hackerrank ahammadmejbah

    - Pipeline Stages and Common Operators
    - Practical Examples of Data Aggregation

7. **Security in MongoDB**

    - Basic Security Practices
    - Role-Based Access Control
    - Encrypting Data

8. **Scaling and Management**

    - Replication: Concepts and Configuration
    - Sharding: Strategies and Implementation
    - Monitoring and Maintenance

1. Introduction to MongoDB

- **What is MongoDB?**

  MongoDB is a document-oriented database classified as a NoSQL database. Here's what that means:

    - **Document-oriented:** Data in MongoDB is stored in flexible documents using a format similar to JSON called BSON (short for Binary JSON). Imagine these documents as enhanced versions of JSON objects where you can store arrays, other documents, and a variety of data types within.

    - **NoSQL:** NoSQL stands for "Not Only SQL." These databases deviate from the traditional rigid, table-based structure of relational databases (more on this below).

- **NoSQL vs. SQL Databases: Understanding the Differences**

Feature	SQL (Relational Databases)	NoSQL (Document Databases)
Data Structure	Rigid tables with predefined columns and rows	Flexible documents (JSON-like) where fields can vary between documents
Schema	Schema-on-write: Enforced structure at the time of data entry	Schema-on-read: Structure is interpreted when data is read, allowing for more flexibility
Scaling	Typically scales vertically (adding more powerful hardware)	Scales horizontally well (adding more servers to a cluster)
Use Cases	Ideal for structured datasets with pre-defined relationships	Suited for rapidly changing data, semi-structured data, and applications needing

Feature	SQL (Relational Databases)	NoSQL (Document Databases)
		high scalability

- **Flexible Schema:** Documents don't need to adhere to a fixed structure. This is fantastic for evolving data models or handling data with some variability.
- **Scalability:** MongoDB excels at horizontal scaling. You can easily distribute data across multiple machines, making it well-suited for large-scale applications.
- **Performance:** Optimized for reads and writes, particularly when paired with effective indexing.
- **Rich Query Language:** You can perform complex queries, aggregations, and manipulate data within MongoDB itself.
- **Developer Friendly:** The JSON-like document model often aligns well with how developers represent data in their applications, easing interactions.

2. Getting Started with MongoDB

Installation

There are two main paths depending on your goals:

- **MongoDB Atlas (Cloud-based):**
    - Ideal for quickly getting started and avoiding local setup.
    - Go to https://www.mongodb.com/cloud/atlas and create a free Atlas cluster. This handles installation and configuration behind the scenes.
- **MongoDB Community Edition (Local Installation):**
    - Great for development or learning on your own machine.
    - **Steps:**
        a. **Download:** Find the appropriate installer for your operating system on the MongoDB download page: https://www.mongodb.com/try/download/community
        b. **Run the installer:** Follow instructions for your OS.
        c. **Data Directory (Important):** MongoDB stores data in a default location (usually something like `/data/db`). Take note of this.

Basic Configuration

- **Starting the MongoDB server:**
    - **Atlas:** Managed for you, no action needed!
    - **Local Install:**
        - Open a terminal / command prompt.
        - Navigate to the MongoDB installation directory (where `mongod` is located).
        - Run the command `mongod` (you might need to add options to specify the data directory you noted earlier).

Understanding the MongoDB Environment

- **The Shell (mongosh):** This is your primary interaction point. Think of it as your command line for MongoDB.
    - **Atlas:** Provided within the web interface
    - **Local Install:** In a separate terminal, run the command `mongosh`.
- **Databases:** MongoDB organizes data into databases. Here's some common terminology:
    - `database`: A container for collections (similar to how a traditional database contains tables).
    - `collection`: A group of documents (analogous to a table in a relational database) .
    - `document`: A single record within a collection (like a row in a table).

Essential Shell Commands (try these out in `mongosh`)

- `show dbs`: List existing databases.
- `use <database_name>`: Switch to a database (`use mydatabase`). You can create one if it doesn't exist.
- `db.mycollection.insertOne({ field: "value" })`: Insert a simple document into a collection named 'mycollection'.
- `db.mycollection.find()`: Retrieve all documents from the collection.

Prerequisites

- **Have MongoDB set up:** Either a local installation or (easier for starting out) a MongoDB Atlas cluster.
- **Install PyMongo:** This is the official driver for interacting with MongoDB from Python. From your terminal run:

```
pip install pymongo
```

Connecting to MongoDB

```python
import pymongo

# For Atlas: Replace with your connection string
client = pymongo.MongoClient("mongodb+srv://<username>:<password>@<your-atlas-cluster>.mongodb

# Select database
db = client["mydatabase"]

# Select collection
collection = db["mycollection"]
```

Advanced MongoDB Shell Commands in Python

- **Insert:**

```
result = collection.insert_one({ "name": "David", "interests": ["reading", "sports"] })
print(result.inserted_id)  # Get the ID of the inserted document
```

- **Find (Queries):**

```
cursor = collection.find({"interests": "coding"})  # Find documents with 'coding' interest
for document in cursor:
    print(document)
```

- **Update:**

```
collection.update_one({"name": "David"}, {"$set": {"city": "New York"}})
```

- **Delete:**

```
collection.delete_many({"interests": {"$in": ["travel"]}})
```

- **Aggregation:**

```
pipeline = [
    {"$match": {"interests": "hiking"}},
    {"$group": {"_id": "$city", "totalHikers": {"$sum": 1}}}
]
results = collection.aggregate(pipeline)
for result in results:
    print(result)
```

3. Fundamentals of MongoDB in Python

Prerequisites (Recap)

- **MongoDB:** Running locally or on Atlas.
- **PyMongo:** Installed (`pip install pymongo`).

Databases and Collections

```
import pymongo

client = pymongo.MongoClient("...")  # Your connection string
```

```python
db = client["mydatabase"]

# Collections are implicitly created if they don't exist
mycollection = db["customers"]
```

- **Analogy:**
  - Database: A large container for related data.
  - Collection: A grouping of similar documents, similar to a table in SQL databases.

Documents: The Basics

```python
customer_document = {
    "name": "Emily",
    "address": {
        "street": "234 Oak Lane",
        "city": "Los Angeles"
    },
    "orders": [12345, 67890]
}
```

- **Key Points:**
  - Documents are like JSON objects (Python dictionaries).
  - Fields can have various data types (strings, numbers, booleans, arrays, subdocuments).
  - **_id field:** A unique identifier automatically generated for each document.

CRUD Operations: Create, Read, Update, Delete

- **Create (Insert):**

  ```python
  result = mycollection.insert_one(customer_document)
  print(result.inserted_id)
  ```

- **Read (Find):**

  ```python
  # Find all:
  cursor = mycollection.find()

  # Find with a filter:
  customer = mycollection.find_one({"name": "Emily"})

  # Iterate over results:
  for doc in cursor:
      print(doc)
  ```

- **Update:**

```python
mycollection.update_one(
    {"name": "Emily"},
    {"$set": {"address.city": "New York"}}
)

# Update multiple documents
mycollection.update_many(
    {"orders": {"$exists": True}},  # Filter for those with an orders field
    {"$addToSet": {"orders": 99999}}  # Add to the orders array
)
```

- **Delete:**

```python
mycollection.delete_one({"name": "Emily"})

mycollection.delete_many({})  # Be careful! Deletes *all* documents
```

Scenario: Managing a Book Collection

Document Structure

```python
book_document = {
    "title": "The Hitchhiker's Guide to the Galaxy",
    "author": "Douglas Adams",
    "genre": "Science Fiction",
    "published_year": 1979,
    "in_stock": True
}
```

Python Script

```python
import pymongo

# Connect to MongoDB (replace with your connection string)
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["my_library"]
books_collection = db["books"]

# 1. Insert some documents
books = [
    {
        "title": "Pride and Prejudice",
        "author": "Jane Austen",
        "genre": "Classic",
        "published_year": 1813,
        "in_stock": True
    },
```

```
    {
        "title": "To Kill a Mockingbird",
        "author": "Harper Lee",
        "genre": "Classic",
        "published_year": 1960,
        "in_stock": False
    },
    # ... add more if you'd like
]
books_collection.insert_many(books)

# 2. Query with different filters
print("Science Fiction Books:")
for book in books_collection.find({"genre": "Science Fiction"}):
    print(book["title"])

print("\nBooks published before 1950:")
for book in books_collection.find({"published_year": {"$lt": 1950}}):
    print(book["title"])

# 3. Update a document
books_collection.update_one(
    {"title": "To Kill a Mockingbird"},
    {"$set": {"in_stock": True}}
)

# 4. Delete documents
books_collection.delete_many({"genre": "Classic"})
```

Explanation

1. **Setup:** Connects to MongoDB, gets the database 'my_library' and collection 'books'.
2. **Insert:** Creates a list of book documents and uses `insert_many` for efficiency if you have multiple items.
3. **Query:** Demonstrates using filters to find books by genre and published year. The `$lt` is a query operator for "less than."
4. **Update:** Uses `update_one` to find a specific book and uses the `$set` operator to modify its `in_stock` status.
5. **Delete:** `delete_many` removes all documents where the genre is "Classic."

Example 1: Data Analysis

```
import pymongo

# ... (assume we have a MongoDB connection)

# Find the average length of movies by genre
pipeline = [
    {"$group": {
```

```
        "_id": "$genre",
        "average_length": {"$avg": "$runtime_minutes" }
     }}
]
results = movies.aggregate(pipeline)

for result in results:
    print(f"{result['_id']} average length: {result['average_length']:.1f} minutes")
```

Example 2: User Preferences

```
import pymongo

# ... (assume we have a MongoDB connection)

# Add a new movie to a user's "favorites" list
user_id = 12345
movie_id = "6258b2f053c58b732a5c06c6"  # Example ObjectID

movies.update_one(
    {"_id": user_id },
    {"$addToSet": {"favorites": movie_id }}
)
```

Example 3: Inventory Management

```
import pymongo

# ... (assume we have a MongoDB connection)

# Mark all low-stock products as "order_soon"
threshold = 10
movies.update_many(
    {"stock_quantity": {"$lt": threshold}},
    {"$set": {"status": "order_soon"}}

)
```

Example 4: Text Search

```
import pymongo

# ... (assume we have a MongoDB connection and a text index on 'title')

search_term = "adventure"
results = movies.find({"$text": {"$search": search_term}})

for movie in results:
```

```
    print(movie["title"])
```

Example 5: Geospatial Queries (Requires Geospatial Index)

```
import pymongo

# ... (assume we have a MongoDB connection)

# Find theaters within 5 kilometers of a location
my_location = [-73.9667, 40.78]  # Example: New York City
max_distance = 5000  # In meters

result = theaters.find(
    {
        "location": {
            "$near": {
                "$geometry": {"type": "Point", "coordinates": my_location},
                "$maxDistance": max_distance
            }
        }
    }
)

for theater in result:
    print(theater['name'])
```

4. Working with Data in MongoDB

Setting Up MongoDB with Python

First, ensure you have the `pymongo` library installed. You can install it using pip if it's not already installed:

```
pip install pymongo
```

1. Data Types and Schemas

In MongoDB, data is stored in documents, which are BSON (Binary JSON) formatted. Common data types include:

- String
- Integer
- Boolean

- Date
- Array
- ObjectID (unique identifier for documents)
- etc.

MongoDB is schema-less, which means each document in a collection can have a different set of fields. However, for consistency and query efficiency, it's common to structure documents similarly within a collection.

2. Inserting Data

Let's insert some data into a collection called `bytes_of_intelligence`.

```
from pymongo import MongoClient

# Connection to MongoDB
client = MongoClient('localhost', 27017)
db = client['intelligence_db']
collection = db['bytes_of_intelligence']

# Insert documents
documents = [
    {"name": "Mejbah", "field": "Data Science", "technologies": ["Python", "MongoDB"], "experi
    {"name": "Ahammad", "field": "Machine Learning", "technologies": ["Python", "TensorFlow"],
]
result = collection.insert_many(documents)
print(f"Inserted document ids: {result.inserted_ids}")
```

3. Querying Data: Basic and Advanced Queries

Now, let's query the data we just inserted, using both basic and advanced queries.

Basic Query:

```
# Querying a single document
mejbah = collection.find_one({"name": "Mejbah"})
print(mejbah)

# Querying multiple documents using a basic filter
developers = collection.find({"experience_years": {"$gt": 2}})
for dev in developers:
    print(dev)
```

Advanced Query:

```python
# Using advanced query features like $in and $or
query = {
    "$or": [
        {"name": "Mejbah"},
        {"technologies": {"$in": ["TensorFlow"]}}
    ]
}
advanced_developers = collection.find(query)
for dev in advanced_developers:
    print(dev)
```

4. Updating and Deleting Data

Lastly, let's see how to update and delete data in the collection.

Updating Data:

```python
# Update a single document
collection.update_one(
    {"name": "Ahammad"},
    {"$set": {"experience_years": 4}}
)
print("Updated Ahammad's experience years.")

# Update multiple documents
collection.update_many(
    {"experience_years": {"$lt": 5}},
    {"$inc": {"experience_years": 1}}
)
print("Increased experience years for all developers with less than 5 years.")
```

Deleting Data:

```python
# Deleting a single document
collection.delete_one({"name": "Mejbah"})
print("Deleted Mejbah's document.")

# Deleting multiple documents
collection.delete_many({"experience_years": {"$lt": 5}})
print("Deleted all developers with less than 5 years of experience.")
```

5. Indexes and Performance

- **Understanding Indexes**

Indexes in MongoDB support the efficient execution of queries. Without indexes, MongoDB must perform a full collection scan, which can be slow if your collection has a lot of documents. Indexes store a portion of the collection's data in an easy-to-traverse form that the database can use to avoid scanning every document.

- **Creating and Managing Indexes**

Creating an Index:

You can create indexes on a collection to improve the performance of frequent queries. The following Python code demonstrates how to create an index using `pymongo`:

```
from pymongo import MongoClient

# Connection to the MongoDB server
client = MongoClient('localhost', 27017)

# Select the database and collection
db = client['intelligence']
collection = db['bytes']

# Creating a simple single field index on the 'topic' field
collection.create_index([('topic', 1)])  # 1 for ascending order

# Creating a compound index on 'topic' and 'date'
collection.create_index([('topic', 1), ('date', -1)])  # -1 for descending order
```

Managing Indexes:

You can list and drop indexes as needed:

```
# List all indexes on a collection
indexes = collection.list_indexes()
for index in indexes:
    print(index)

# Dropping an index
collection.drop_index('topic_1_date_-1')  # Index name format is generally 'field_order'
```

- **Performance Tuning Tips**

1. **Use Appropriate Indexes:**
   - Analyze your application's query patterns and index fields that are most frequently queried.
   - Remember that indexes take up extra disk space and can slow down write operations because they also need to be updated. It's a balance!
2. **Monitor Performance:**

- Utilize tools like MongoDB's Atlas platform or the `mongostat` and `mongotop` utilities to monitor the database's performance and understand where bottlenecks might be occurring.

3. **Index Management:**

   - Avoid having too many indexes. Regularly review and remove unused or less important indexes.
   - Use partial indexes when you only need to index a subset of the data that meets certain criteria, which can save space and improve write performance.

4. **Optimize Query Patterns:**

   - Aim to write queries that can take full advantage of the indexes.
   - Use the `.explain()` method in MongoDB to analyze the efficiency of your queries.

5. **Hardware Considerations:**

   - Ensure that your working set fits into RAM. MongoDB performs best when the frequently accessed data fits into memory.
   - Use SSDs for better I/O performance, which is crucial for write-heavy applications.

Here's how you might use the `.explain()` method to check a query's performance:

```
# Analyzing a query's execution plan
query_performance = collection.find({"topic": "machine learning"}).explain()
print(query_performance)
```

6. Data Aggregation

The MongoDB Aggregation Framework is a powerful set of tools for transforming and combining data in MongoDB. It operates through a pipeline model, where data passes through multiple stages, each transforming the data in some way. It's especially useful for complex queries and reporting.

Concepts and Usage

The aggregation pipeline is a framework in MongoDB designed to perform data transformation and summary. The data is processed as it passes through a sequence of stages, and each stage transforms the data in a specific way (e.g., filtering, grouping, sorting).

Pipeline Stages and Common Operators

Here are a few common stages in the aggregation pipeline:

- **`$match`**: Filters the documents to pass only those that match the given condition into the next stage.
- **`$group`**: Groups documents by specified expressions.

- `$sort` : Sorts documents.
- `$project` : Passes along the documents with the requested fields to the next stage, potentially adding new fields or removing existing fields.
- `$sum` : Calculates the sum of specified values from all documents in the collection.
- `$avg` : Calculates the average of specified values from all documents.

Practical Examples of Data Aggregation

Setup: Assuming we have a collection `bytes_of_intelligence` with documents structured like:

```
{
  "name": "Mejbah",
  "field": "Data Science",
  "technologies": ["Python", "MongoDB"],
  "experience_years": 5,
  "projects": 10
},
{
  "name": "Ahammad",
  "field": "Machine Learning",
  "technologies": ["Python", "TensorFlow"],
  "experience_years": 3,
  "projects": 5
}
```

Python Code for Aggregation:

```python
from pymongo import MongoClient

# Connecting to MongoDB
client = MongoClient('localhost', 27017)
db = client['intelligence_db']
collection = db['bytes_of_intelligence']

# Aggregation Pipeline
pipeline = [
    {"$match": {"technologies": "Python"}},  # Filter: Select docs with Python in technologies
    {"$group": {
        "_id": "$field",  # Group by field
        "average_experience": {"$avg": "$experience_years"},  # Average experience in each fie
        "total_projects": {"$sum": "$projects"}  # Total projects in each field
    }},
    {"$sort": {"average_experience": -1}}  # Sort by average experience descending
]

results = collection.aggregate(pipeline)
```

```
for result in results:
    print(result)
```

Explanation:

1. `$match` : This stage filters documents to include only those where 'technologies' array contains 'Python'.
2. `$group` : Documents are grouped by the 'field', calculating the average years of experience and total projects for each field.
3. `$sort` : The resulting groups are sorted based on the average experience in descending order.

8. Scaling and Management

As databases grow in size and demand, scaling and management become crucial for maintaining performance and availability. MongoDB provides two main mechanisms for scaling: replication and sharding. Along with these, effective monitoring and maintenance practices are vital.

Replication: Concepts and Configuration

Concepts:

- **Replication** in MongoDB is the process of synchronizing data across multiple servers.
- A **replica set** is a group of MongoDB servers that maintain the same data set, providing redundancy and high availability.
- **Primary** node receives all write operations. **Secondary** nodes replicate the primary's data set and can serve read operations to increase read capacity.

Configuration Example: Configuring a replica set typically involves setting up multiple instances of MongoDB, then configuring them to be aware of each other. Below is a conceptual setup using Python, but keep in mind actual deployment would require MongoDB server configurations and might not involve Python code for the setup phase.

```
# This is a pseudo-code/hypothetical example for understanding concepts
from pymongo import MongoClient

# Connecting to a replica set
# The MongoClient would connect using a connection string that specifies multiple servers.
client = MongoClient('mongodb://db1.example.net:27017,db2.example.net:27017,db3.example.net:27

db = client['intelligence_db']
collection = db['bytes_of_intelligence']
```

```
# Inserting a document into a replica set
collection.insert_one({"name": "Mejbah", "field": "Data Science"})
```

Notes:

- The connection string includes all members of the replica set.
- The `?replicaSet=myReplicaSet` argument specifies the name of the replica set.

Sharding: Strategies and Implementation

Concepts:

- **Sharding** distributes data across multiple machines to support deployments with very large data sets and high throughput operations.
- A **shard** is a single MongoDB instance that holds a portion of the dataset.
- **Shard keys** are chosen based on the document fields that determine the distribution of the collection's documents among shards.

Implementation Example: Setting up sharding is complex and generally handled at the database and infrastructure level. However, the concept and strategy can be outlined:

```
# Hypothetical Python code to demonstrate concept

# Assume 'shard_key' is set on a field that evenly distributes the data
# Example: Setting a shard key on 'field' if data is evenly distributed across different field
db.adminCommand({
    'shardCollection': 'intelligence_db.bytes_of_intelligence',
    'key': {'field': 1}  # This would be done in the MongoDB shell, not via Python
})
```

Monitoring and Maintenance

Concepts:

- Monitoring involves tracking the performance, health, and availability of MongoDB instances.
- Maintenance includes tasks such as backing up data, pruning old entries, and optimizing indexes.

Python Example: For monitoring, you might use MongoDB's own or third-party tools, but you can access some statistics via Python:

```
# Fetching database stats
db_stats = db.command("dbstats")
print(db_stats)
```

```python
# Fetching collection stats
collection_stats = db.command("collstats", "bytes_of_intelligence")
print(collection_stats)
```