Assignment 13 MondoDB

October 6, 2023

Q1. What is MongoDB? Explain non-relational databases in short. In which scenarios it is preferred to use MongoDB over SQL databases?

ANS:

MongoDB is a popular NoSQL (non-relational) database management system that is designed for storing, retrieving, and managing large volumes of data in a flexible and scalable manner. MongoDB is known for its document-oriented data model, which means it stores data in a format similar to JSON documents, making it suitable for a wide range of applications and use cases.

Here's a brief explanation of non-relational databases and when MongoDB is preferred over SQL databases:

Non-relational databases (NoSQL): Non-relational databases, often referred to as NoSQL databases, are a category of database management systems that depart from the traditional relational database model (SQL databases). They are designed to handle large and complex datasets and offer flexibility, scalability, and performance advantages for certain use cases. Key characteristics of NoSQL databases include:

Schema flexibility: NoSQL databases can handle unstructured or semi-structured data and do not require a fixed schema definition. Horizontal scalability: Many NoSQL databases can scale out horizontally by adding more servers or nodes to accommodate increasing data loads. Different data models: NoSQL databases support various data models, including document-oriented, key-value, column-family, and graph databases, to better match specific application needs. When MongoDB is preferred over SQL databases: MongoDB is often preferred over SQL databases in the following scenarios:

Semi-structured or rapidly evolving data: MongoDB's flexible schema allows you to store data without a predefined structure. This is beneficial when dealing with data that evolves over time or doesn't fit neatly into a tabular structure.

Scalability: MongoDB is well-suited for applications with high scalability requirements. It can easily distribute data across multiple servers, making it suitable for handling large amounts of data and high traffic loads.

Complex queries: While MongoDB doesn't support the complex querying capabilities of SQL (Structured Query Language), it can handle a wide range of queries. For applications where the need for complex joins and transactions is minimal, MongoDB's query language is sufficient.

Real-time analytics: MongoDB is often used in scenarios where real-time data analysis and processing are required, such as in Internet of Things (IoT) applications, social media platforms, and real-time monitoring systems.

Dynamic and agile development: MongoDB is favored by development teams that need to iterate quickly and make frequent changes to the data model. Its schema-less nature allows for agile development and faster time-to-market.

Geospatial data: MongoDB has built-in support for geospatial data and queries, making it a preferred choice for applications that involve location-based services.

[ ]:

Q2. State and Explain the features of MongoDB.

ANS:

MongoDB is a popular NoSQL database management system known for its flexibility, scalability, and ease of use. It offers a range of features that make it suitable for various application scenarios. Here are some of the key features of MongoDB, along with explanations:

Document-Oriented: MongoDB is a document-oriented database, which means it stores data in BSON (Binary JSON) format. Each data record is a JSON-like document, and documents within a collection (similar to a table in relational databases) can have varying structures, allowing for flexible and dynamic data modeling.

Schema Flexibility: MongoDB provides schema flexibility, making it ideal for applications with evolving or variable data structures. There's no need to define a fixed schema upfront, and fields can be added or modified in documents as needed.

Highly Scalable: MongoDB is designed for horizontal scalability, allowing you to distribute data across multiple servers or nodes to handle large volumes of data and high traffic loads. This is achieved through sharding, which partitions data across different machines.

Automatic Sharding: MongoDB provides automatic sharding capabilities, meaning it can automatically distribute data across shards (physical server clusters) to ensure data distribution and availability. This enables horizontal scaling without manual intervention.

Rich Query Language: MongoDB's query language supports a wide range of query operations, including filtering, sorting, and geospatial queries. It also provides support for full-text search, text indexing, and aggregation pipelines for complex data transformations.

Indexing: MongoDB supports indexing to optimize query performance. You can create custom indexes on fields to speed up data retrieval. MongoDB also includes a default _id index for fast document retrieval.

Geospatial Capabilities: MongoDB offers geospatial indexing and queries, making it suitable for location-based applications. You can store and query geospatial data, such as points, lines, and polygons, efficiently.

Replication: MongoDB supports data replication to ensure data availability and fault tolerance. It uses replica sets, allowing multiple copies of data across different servers, with one primary and multiple secondary nodes for redundancy.

Flexible Data Models: MongoDB supports various data models beyond traditional document storage, including key-value, column-family, and graph data models. This flexibility allows you to choose the right data model for your application.

ACID Transactions: MongoDB introduced multi-document ACID transactions in version 4.0, ensuring data consistency and reliability for operations that span multiple documents.

Security: MongoDB provides robust security features, including authentication, authorization, role-based access control, and encryption at rest and in transit, to protect sensitive data.

Aggregation Framework: MongoDB's aggregation framework allows for advanced data transformation and analysis. It provides operators and expressions for performing complex data manipulations and computations.

Community and Ecosystem: MongoDB has a large and active open-source community, extensive documentation, and a rich ecosystem of drivers, libraries, and tools for various programming languages and platforms.

Cloud Integration: MongoDB Atlas is a fully managed database service offered by MongoDB, Inc. It allows you to run MongoDB in the cloud with automated provisioning, scaling, and monitoring.

Monitoring and Management: MongoDB provides tools like MongoDB Compass for visual query building, monitoring tools, and management interfaces for easy database administration.

[ ]:

Q3. Write a code to connect MongoDB to Python. Also, create a database and a collection in MongoDB.

ANS:

To connect MongoDB to Python and create a database and a collection, you can use the pymongo library, which is a Python driver for MongoDB. Here's a step-by-step guide and code example:

Install the pymongo library if you haven't already:

[1]:
```
pip install pymongo
```

```
Collecting pymongo
  Downloading
pymongo-4.5.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (671
kB)
                           671.3/671.3 kB
14.4 MB/s eta 0:00:00a 0:00:01
Collecting dnspython<3.0.0,>=1.16.0
  Downloading dnspython-2.4.2-py3-none-any.whl (300 kB)
                           300.4/300.4 kB
36.5 MB/s eta 0:00:00
Installing collected packages: dnspython, pymongo
Successfully installed dnspython-2.4.2 pymongo-4.5.0
Note: you may need to restart the kernel to use updated packages.
```

Create a Python script to establish a connection to MongoDB, create a database, and create a collection within that database:

[ ]:
```
import pymongo

# Define the MongoDB connection parameters
mongodb_uri = "mongodb://localhost:27017/"
database_name = "mydatabase"
collection_name = "mycollection"

try:
    # Establish a connection to MongoDB
    client = pymongo.MongoClient(mongodb_uri)

    # Access the database (create if it doesn't exist)
    database = client[database_name]

    # Access the collection (create if it doesn't exist)
    collection = database[collection_name]

    # Verify if the collection was created
    if collection:
        print(f"Connected to MongoDB, created database '{database_name}', and␣
 ↪collection '{collection_name}'")

except pymongo.errors.ConnectionFailure as e:
    print(f"Failed to connect to MongoDB: {e}")
```

In this code:

We import the pymongo library to work with MongoDB. We define the MongoDB URI, which specifies the server's address and port (default is 27017). You may need to adjust this URI to match your MongoDB server's configuration. We specify the name of the database and the name of the collection we want to create. We use a try…except block to handle any connection errors. Inside the try block, we establish a connection to MongoDB, create/access the database, and create/access the collection within that database. Finally, we print a confirmation message if the connection and database/collection creation were successful.

[ ]:
```

```

Q4. Using the database and the collection created in question number 3, write a code to insert one record, and insert many records. Use the find() and find_one() methods to print the inserted record.

ANS:

[ ]:
```
import pymongo

# Define the MongoDB connection parameters
mongodb_uri = "mongodb://localhost:27017/"
database_name = "mydatabase"
collection_name = "mycollection"
```

```python
# Sample data for inserting multiple records
data_to_insert = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 25, "city": "Los Angeles"},
    {"name": "Charlie", "age": 35, "city": "San Francisco"},
]

try:
    # Establish a connection to MongoDB
    client = pymongo.MongoClient(mongodb_uri)

    # Access the database and collection
    database = client[database_name]
    collection = database[collection_name]

    # Insert one record
    record_to_insert = {"name": "David", "age": 28, "city": "Chicago"}
    collection.insert_one(record_to_insert)

    # Insert multiple records
    collection.insert_many(data_to_insert)

    # Find and print the inserted records
    print("Inserted Records:")

    # Using find() to retrieve all records
    for record in collection.find():
        print(record)

    # Using find_one() to retrieve a single record
    single_record = collection.find_one({"name": "David"})
    print("\nSingle Inserted Record:")
    print(single_record)

except pymongo.errors.ConnectionFailure as e:
    print(f"Failed to connect to MongoDB: {e}")
```

[ ]:

Q5. Explain how you can use the find() method to query the MongoDB database. Write a simple code to demonstrate this.

ANS:

The find() method in MongoDB is used to query and retrieve documents from a collection based on specified criteria or filters. It returns a cursor that can be iterated through to access the documents that match the query conditions. The basic syntax of the find() method is as follows:

[ ]:
```
cursor = collection.find(query, projection)
```

query: This parameter specifies the filter conditions to match documents. It is a dictionary where keys represent field names, and values represent the desired values or query operators (e.g., $eq, $gt, $lt, $in).

projection (optional): This parameter specifies which fields should be included or excluded in the returned documents. It is also a dictionary, where 1 indicates inclusion, and 0 indicates exclusion.

[ ]:
```
import pymongo

# Define the MongoDB connection parameters
mongodb_uri = "mongodb://localhost:27017/"
database_name = "mydatabase"
collection_name = "mycollection"

try:
    # Establish a connection to MongoDB
    client = pymongo.MongoClient(mongodb_uri)

    # Access the database and collection
    database = client[database_name]
    collection = database[collection_name]

    # Query documents with age greater than 30
    query = {"age": {"$gt": 30}}

    # Projection to include only name and age fields
    projection = {"name": 1, "age": 1, "_id": 0}

    # Use find() to retrieve matching documents
    cursor = collection.find(query, projection)

    # Iterate through the cursor and print the results
    print("Matching Records:")
    for document in cursor:
        print(document)

except pymongo.errors.ConnectionFailure as e:
    print(f"Failed to connect to MongoDB: {e}")
```

[ ]:
```

```

Q6. Explain the sort() method. Give an example to demonstrate sorting in MongoDB.

ANS:

The sort() method in MongoDB is used to specify the order in which documents should be returned in the result set when querying a collection. You can use this method to sort documents based on

one or more fields in ascending (ascending order) or descending (descending order) order.

[ ]:
```
cursor = collection.find(query).sort(sort_field, direction)
```

query: This is the filter condition for the documents you want to retrieve.

sort_field: This parameter specifies the field by which you want to sort the documents.

direction: This parameter specifies the sorting direction, and it can have the following values:

pymongo.ASCENDING (or 1): Sort in ascending order (default).

pymongo.DESCENDING (or -1): Sort in descending order.

Here's an example that demonstrates sorting in MongoDB using the sort() method:

[ ]:
```
import pymongo

# Define the MongoDB connection parameters
mongodb_uri = "mongodb://localhost:27017/"
database_name = "mydatabase"
collection_name = "students"

try:
    # Establish a connection to MongoDB
    client = pymongo.MongoClient(mongodb_uri)

    # Access the database and collection
    database = client[database_name]
    collection = database[collection_name]

    # Query and sort documents by the "score" field in descending order
    query = {}  # An empty query to retrieve all documents
    sort_field = "score"
    direction = pymongo.DESCENDING

    # Use find() with sort() to retrieve and sort matching documents
    cursor = collection.find(query).sort(sort_field, direction)

    # Iterate through the cursor and print the sorted results
    print("Sorted Records (Descending by Score):")
    for document in cursor:
        print(document)

except pymongo.errors.ConnectionFailure as e:
    print(f"Failed to connect to MongoDB: {e}")
```

[ ]:
```

```

Q7. Explain why delete_one(), delete_many(), and drop() is used.

ANS:

delete_one() Method:

delete_one() is used when you want to delete a single document that matches a specified filter condition. It removes the first document found that matches the filter and then stops searching. Useful for scenarios where you need to remove a specific document or a single occurrence of a document based on certain criteria. For example, you can use delete_one() to remove a single comment from a blog post or to delete a user profile by their unique identifier.

delete_many() Method:

delete_many() is employed when you want to delete multiple documents that match a specified filter condition. It removes all documents that meet the filter criteria, not just the first match. Valuable when you need to perform bulk deletion operations, such as removing all records that are no longer relevant or are marked as "archived." Commonly used in scenarios like cleaning up outdated log entries or purging expired user sessions.

drop() Method (Collection Deletion):

drop() is used to delete an entire collection, including all of its documents and associated indexes. It permanently removes the collection from the database, and there is no way to recover the data once it's dropped. Typically used when you want to get rid of an entire collection, for example, when you no longer need it or when you want to start with a fresh, empty collection. Be extremely cautious when using drop() because it irreversibly deletes all data in the collection.

[ ]: