Highlighting MongoDB: Flexible NoSQL Document Database

MongoDB is a popular open-source NoSQL document database. Unlike traditional relational databases, MongoDB stores data in flexible, JSON-like documents, making it ideal for semi-structured data, rapidly evolving schemas, and scenarios where data models are dynamic or complex. In your data platform, MongoDB can serve as a specialized storage layer for specific application use cases, event logs, or data that doesn't fit neatly into a rigid relational schema.
This guide will demonstrate basic and advanced use cases of MongoDB, leveraging your **Advanced Track** local environment setup and its integration with Python and Apache Spark.
Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**.

Basic Use Case: Storing Flexible Semi-Structured Data

Objective: To demonstrate how to store and retrieve semi-structured data into MongoDB, highlighting its ability to accept documents without a predefined, rigid schema.
Role in Platform: Provide a highly flexible and scalable storage solution for data that doesn't conform to strict relational models, such as diverse log events, user profiles with varying attributes, or metadata with dynamic properties.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root. This includes the mongodb service.
2. **Verify MongoDB is accessible:** Check Docker logs for the mongodb container (docker compose logs mongodb).
3. **Install MongoDB Shell (mongosh):** If not already installed on your host, you might want it for direct interaction. Alternatively, you can use docker exec to access the shell inside the container.
    - Install mongosh locally: MongoDB Shell Installation
    - Or execute inside container: docker exec -it mongodb mongosh --authenticationDatabase admin -u root -p password

Steps to Exercise:

1. **Connect to MongoDB:**
   mongosh "mongodb://localhost:27017" --authenticationDatabase admin -u root -p password

   (Replace localhost with mongodb if running from another container within the Docker network, e.g., from fastapi_ingestor container via docker exec).
2. **Insert a Simple Document:**

- Switch to a database (it will be created if it doesn't exist):
  use my_data_platform_db

- Insert a document into a collection (also created if it doesn't exist):
  db.financial_events.insertOne({
    "event_id": "EVT-001",
    "type": "login_attempt",
    "user_id": "USR-XYZ",
    "timestamp": ISODate("2024-06-14T10:30:00Z"),
    "ip_address": "192.168.1.100",
    "status": "success"
  })

3. **Insert a Document with Different Fields:**
   - Insert another document into the *same* financial_events collection, but with different or additional fields:
     db.financial_events.insertOne({
       "event_id": "EVT-002",
       "type": "failed_login",
       "user_id": "USR-ABC",
       "timestamp": ISODate("2024-06-14T10:31:00Z"),
       "ip_address": "192.168.1.101",
       "reason": "incorrect_password",
       "attempts": 3
     })

4. **Query the Documents:**
   db.financial_events.find().pretty()

Verification:

- **MongoDB Shell Output:** The insertOne commands return success acknowledgments. The find().pretty() command displays both documents, showcasing that MongoDB accepted records with different structures in the same collection without any schema definition errors.
- **Docker Logs:** The mongodb container logs should show successful insert operations.

Advanced Use Case 1: Dynamic Schema and Flexible Data Models

Objective: To explicitly demonstrate MongoDB's schemaless nature by inserting and querying documents with highly varying structures within the same collection, which is challenging for relational databases.
Role in Platform: Accommodate rapidly changing data formats, store diverse IoT data, social

media feeds, or complex nested documents where a rigid schema is impractical or constantly evolving.

Setup/Configuration:

1. **Ensure MongoDB is running** (as per Basic Use Case).

Steps to Exercise:

1. **Connect to MongoDB Shell:**
   mongosh "mongodb://localhost:27017" --authenticationDatabase admin -u root -p password
   use dynamic_insurance_data

2. **Insert a "Claim" Document (with nested object, array):**
   db.claims.insertOne({
   "claim_id": "C-001",
   "policy_id": "P-1001",
   "claim_date": ISODate("2024-05-15T00:00:00Z"),
   "status": "pending",
   "details": {
   "type": "auto",
   "incident_date": ISODate("2024-05-10T14:00:00Z"),
   "vehicle_vin": "ABC123XYZ456",
   "damages": ["front_bumper", "headlight"]
   },
   "attachments": [
   {"filename": "photo1.jpg", "type": "image"},
   {"filename": "report.pdf", "type": "document"}
   ]
   })

3. **Insert a "Customer Profile" Document (with a different structure):**
   db.customer_profiles.insertOne({
   "customer_id": "CUST-001",
   "name": "Alice Smith",
   "contact": {
   "email": "alice@example.com",
   "phone": "555-1234"
   },
   "addresses": [
   {"street": "123 Main St", "city": "Anytown", "zip": "12345", "type": "billing"},
   {"street": "456 Oak Ave", "city": "Anytown", "zip": "12345", "type": "shipping"}
   ],
   "preferences": {
   "newsletter": true,
   "product_updates": false

```
    }
})
```

Note: We used a different collection (customer_profiles) here, but the principle of flexible schema applies within a single collection as well, by simply inserting documents with varying fields.

4. **Query Documents with Specific Fields:**
   ```
   db.claims.find({"details.type": "auto"}).pretty()
   db.customer_profiles.find({"addresses.city": "Anytown"}).pretty()
   ```

   This demonstrates querying into nested structures and arrays.

Verification:

- **MongoDB Shell Output:** The documents are successfully inserted and retrieved despite their different, complex structures. Queries on nested fields work as expected. This proves MongoDB's flexibility with dynamic schemas and its ability to handle varied JSON-like data.

Advanced Use Case 2: Integration with Spark for Complex Document Processing

Objective: To demonstrate how Apache Spark can efficiently read and process semi-structured data directly from MongoDB collections, perform transformations (e.g., flatten nested fields, filter), and then write the processed data to a structured format like Delta Lake.

Role in Platform: Enable powerful distributed analytics and ETL on data stored in MongoDB, bridging the gap between flexible document storage and structured data lake analysis.

Setup/Configuration:

1. **Ensure MongoDB and Spark are running.**
2. **Install Spark-MongoDB Connector:** Ensure your Spark Docker image (or the spark-submit command) includes the MongoDB Spark Connector package. (e.g., org.mongodb.spark:mongo-spark-connector_2.12:10.0.0).
3. **Populate MongoDB:** Ensure my_data_platform_db.financial_events (from Basic Use Case) or dynamic_insurance_data.claims (from Adv Use Case 1) has some data.
4. **Spark Processing Script:** Create a Python script pyspark_jobs/mongo_processor.py.
   Example pyspark_jobs/mongo_processor.py (conceptual):
   ```
   # pyspark_jobs/mongo_processor.py
   import sys
   from pyspark.sql import SparkSession
   from pyspark.sql.functions import col, explode, current_timestamp

   def create_spark_session(app_name):
       """Helper function to create a SparkSession with Delta Lake and MongoDB packages."""
   ```

```
    return (SparkSession.builder.appName(app_name)
            .config("spark.jars.packages",
"io.delta:delta-core_2.12:2.4.0,org.mongodb.spark:mongo-spark-connector_2.12:10.0.0"
)
            .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
            .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
            .getOrCreate())

if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: mongo_processor.py <mongo_db_name> <mongo_collection_name>
<delta_output_path>")
        sys.exit(-1)

    mongo_db_name = sys.argv[1]
    mongo_collection_name = sys.argv[2]
    delta_output_path = sys.argv[3]

    spark = create_spark_session(f"MongoDBToDelta_{mongo_collection_name}")
    spark.sparkContext.setLogLevel("WARN")

    # MongoDB connection URI (use 'mongodb' service name in Docker Compose)
    mongo_uri = "mongodb://root:password@mongodb:27017/"

    # Read data from MongoDB
    print(f"Reading data from MongoDB: db={mongo_db_name},
collection={mongo_collection_name}")
    df_mongo = (spark.read.format("mongodb")
                .option("spark.mongodb.input.uri",
f"{mongo_uri}{mongo_db_name}.{mongo_collection_name}")
                .load())

    print("Schema of data read from MongoDB:")
    df_mongo.printSchema()
    df_mongo.show(5, truncate=False)

    # --- Example Transformation: Flattening nested 'details' and 'attachments' for
'claims' ---
    # Assuming the 'claims' collection with nested structures
    if mongo_collection_name == "claims":
        print("Applying transformations for claims data...")
        # Select relevant fields and flatten nested 'details' object
```

```
        df_transformed = df_mongo.select(
            col("claim_id"),
            col("policy_id"),
            col("claim_date"),
            col("status"),
            col("details.type").alias("claim_type"),
            col("details.incident_date").alias("incident_date"),
            col("details.vehicle_vin").alias("vehicle_vin"),
            explode(col("details.damages")).alias("damage_item"), # Explode damages array
            current_timestamp().alias("processing_timestamp")
        )
        print("Schema after flattening:")
        df_transformed.printSchema()
        df_transformed.show(5, truncate=False)

    elif mongo_collection_name == "financial_events":
        print("Applying transformations for financial events data...")
        df_transformed = df_mongo.select(
            col("event_id"),
            col("type"),
            col("user_id"),
            col("timestamp"),
            col("ip_address"),
            col("status"),
            col("reason").alias("failure_reason"), # Handle optional field
            col("attempts").alias("login_attempts"), # Handle optional field
            current_timestamp().alias("processing_timestamp")
        )
        print("Schema after selecting/renaming:")
        df_transformed.printSchema()
        df_transformed.show(5, truncate=False)
    else:
        print("No specific transformation defined for this collection. Writing as is.")
        df_transformed = df_mongo.withColumn("processing_timestamp",
current_timestamp())


    # Write the processed data to Delta Lake in MinIO
    print(f"Writing transformed data to Delta Lake: {delta_output_path}")
    df_transformed.write.format("delta").mode("overwrite").option("overwriteSchema",
"true").save(delta_output_path)
    print("Transformation and write to Delta Lake complete.")
```

```
spark.stop()
```

Steps to Exercise:

1. **Populate MongoDB:** Ensure you have inserted sample data into my_data_platform_db.financial_events (or dynamic_insurance_data.claims) as per Basic/Advanced Use Case 1.
2. **Submit Spark Job:** In a new terminal, submit the mongo_processor.py job.
    - **For financial_events:**
      ```
      docker exec -it spark spark-submit \
        --packages io.delta:delta-core_2.12:2.4.0,org.mongodb.spark:mongo-spark-connector_2.12:10.0.0 \
        --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
        --conf spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
        --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
        --conf spark.hadoop.fs.s3a.access.key=minioadmin \
        --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
        --conf spark.hadoop.fs.s3a.path.style.access=true \
        /opt/bitnami/spark/jobs/mongo_processor.py \
        my_data_platform_db financial_events s3a://curated-data-bucket/mongo_financial_events_curated
      ```
    - **For claims (if populated):**
      ```
      docker exec -it spark spark-submit \
        --packages io.delta:delta-core_2.12:2.4.0,org.mongodb.spark:mongo-spark-connector_2.12:10.0.0 \
        --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
        --conf spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
        --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
        --conf spark.hadoop.fs.s3a.access.key=minioadmin \
        --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
        --conf spark.hadoop.fs.s3a.path.style.access=true \
        /opt/bitnami/spark/jobs/mongo_processor.py \
        dynamic_insurance_data claims s3a://curated-data-bucket/mongo_insurance_claims_curated
      ```
3. **Monitor:** Observe the console output of the spark-submit command, looking for schema inference and transformation details.

Verification:

- **Spark Logs:** The logs will show Spark successfully connecting to MongoDB, reading the documents, applying the transformations, and writing to the specified Delta Lake path.
- **MinIO Console (http://localhost:9001):** Navigate to curated-data-bucket. You should see new directories (mongo_financial_events_curated or mongo_insurance_claims_curated) containing .parquet files and a _delta_log, confirming the data has been processed and stored.
- **Query Processed Data (Spark-SQL):**
  docker exec -it spark spark-sql \
    --packages io.delta:delta-core_2.12:2.4.0 \
    --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
    --conf
  spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
    --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
    --conf spark.hadoop.fs.s3a.access.key=minioadmin \
    --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
    --conf spark.hadoop.fs.s3a.path.style.access=true \
    -e "SELECT * FROM
  delta.\`s3a://curated-data-bucket/mongo_financial_events_curated\` LIMIT 5;"

  (Or for claims: mongo_insurance_claims_curated). The query results will show the flattened and transformed data, ready for further analysis.

Advanced Use Case 3: Scalability and High Availability (Conceptual)

Objective: To conceptually demonstrate MongoDB's native support for horizontal scalability through sharding and high availability through replica sets, which are critical for production deployments handling large data volumes and high query loads.
Role in Platform: Ensure the MongoDB layer can scale to accommodate growing data volumes and query throughput, and provide continuous availability even in the face of node failures.
Setup/Configuration (Conceptual Discussion):

1. **Replica Sets:**
   - In a production environment, MongoDB is typically deployed as a replica set, which is a group of mongod processes that maintain the same data set. This provides redundancy and high availability.
   - Your docker-compose.yml likely runs a single MongoDB instance for local development simplicity. To run a replica set locally, you'd need multiple mongodb services and an explicit replicaSet configuration in their command.
     Example docker-compose.yml snippet (conceptual, for a 3-node replica set):
     # services:
     # mongo1:

```
#     image: mongo:latest
#     command: mongod --replSet rs0 --bind_ip 0.0.0.0
#     volumes:
#       - mongo_data1:/data/db
#     ports:
#       - "27017:27017"
#   mongo2:
#     image: mongo:latest
#     command: mongod --replSet rs0 --bind_ip 0.0.0.0
#     volumes:
#       - mongo_data2:/data/db
#   mongo3:
#     image: mongo:latest
#     command: mongod --replSet rs0 --bind_ip 0.0.0.0
#     volumes:
#       - mongo_data3:/data/db
# # ... then initiate the replica set via mongosh on one node:
# # rs.initiate({ _id: "rs0", members: [{ _id: 0, host: "mongo1:27017" }, { _id: 1, host: "mongo2:27017" }, { _id: 2, host: "mongo3:27017" }]})
```

2. **Sharding:**
    - For horizontal scalability beyond what a single replica set can offer (e.g., handling petabytes of data or millions of operations per second), MongoDB supports sharding. Sharding distributes data across multiple shard clusters, each acting as an independent replica set.
    - This setup involves: config servers (metadata), mongos router (query routing), and shard replica sets. This is significantly more complex to simulate locally in Docker Compose and is typically a production-only architecture.

Steps to Exercise (Conceptual Discussion):

1. **Discuss Replica Set Benefits:**
    - **Automatic Failover:** If the primary node fails, an election occurs, and a new primary is chosen, ensuring continuous operation.
    - **Data Redundancy:** Multiple copies of data protect against data loss.
    - **Read Scalability:** Reads can be distributed across secondary nodes, offloading the primary.
2. **Discuss Sharding Benefits:**
    - **Horizontal Scalability:** Distribute data and load across many servers, allowing the cluster to grow almost indefinitely.
    - **High Throughput:** Handle massive query volumes by parallelizing operations across shards.
    - **Large Data Sets:** Store datasets that exceed the capacity of a single server.
3. **Explain Local vs. Production:** Emphasize that while your docker-compose.yml provides a single mongodb instance for development convenience, a production

deployment would always involve a replica set (at least three nodes) and potentially sharding for extreme scale.

Verification (Conceptual):

- Understanding the architectural patterns of MongoDB replica sets and sharding demonstrates a grasp of how the database layer contributes to the overall scalability and high availability of the data platform in a production environment. This directly translates to an AWS Engineer's understanding of services like Amazon DocumentDB, which manages these underlying complexities.

This concludes the guide for MongoDB.