MongoDB

Mimanshu Maheshwari

September 4, 2024

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Getting Started with MongoDB Atlas

1.1.1 Creating and Deploying on Atlas Cluster

1. Create a Organization
2. Create a Project under Organization
3. Create a cluster → Advanced Settings → Shared M0 cluster → Add username and password → Add you ip as allowed origin.
4. Open Database tab and navigate to your cluster
5. Load default data
6. Navigate to collections tab to see loaded clusters
7. Sample database fetch from analytics accounts table

```
{
  "_id": {
    "$oid": "5ca4bbc7a2dd94ee5816238c"
  },
  "account_id": {
    "$numberInt": "371138"
  },
  "limit": {
    "$numberInt": "9000"
  },
  "products": [
    "Derivatives",
    "InvestmentStock"
  ]
}
```

Listing 1.1: account test data

8. To filter on account id:

```
{
  account_id: 794875
}
```

Listing 1.2: filter on account id

9. To login into atlas login use `atlas auth login`
10. create cluster with user and password:

```
atlas setup --clusterName myAtlasClusterEDU --provider WS --currentIp --skipSampleData --
    ↪ username myAtlasDBUser --password myatlas-001 --projectId 66d19c1f26ef8b512df3b41 |
    ↪ tee atlas_cluster_details.txt
```

Listing 1.3: load cluster

11. Load sample data: `atlas clusters smapleData load myAtlasClusterEDU`

1.2 Introduction to MongoDB

1.2.1 Key Terms

Document

The basic using of data in MongoDB

Collections

A grouping of documents. Documents in collection are typically similar they don't necessarily have same structure.

Database

A container for Collections

1.3 MongoDB and The Document Model

1.3.1 The MongoDB Document Model

- MongoDB stores data in structures known as documents

Document

Docuement are displayed in JavaScript Object Notation (JSON) format but stored in Binary JavaScript Object Notation (BSON) format. BSON also supports additional data types that are unavailable in JSON

It can support All JSON data types as well as Dates, Numbers, Object Id's, and more!

ObjectID is used to create unique identifires. Every Document requires an $_id$ field, which acts as a primary key. If an inserted document doesn't include the $_id$ field, MongoDB automatically generates an ObjectId for the $_id$ field. MongoDB supports flexible schema model and polymorphic data, this allows us to store documents with different structure together. Documents may contain fields. Fields may contain different types.

Optional Schema validation

Set of constraints on the structure of documents.

1.4 Managing Databases, Colections, and Documents in Atlas Data Explorer

Explore Atlas

Chapter 2

Data Modeling

2.1 Introduction to Data Modeling

Data Modeling is process of defining how data is stored and the relationships and the relationships that exists among different entities in your data.
We refer to Organization of data inside a database as a **Schema**.
To develop schema think about your database and you need to ask these questions:

- What does my application do?
- What data will I store?
- How will users access this data?
- What data will be most valuable to me?

Asking these questions will help you to:

- descibe your task as well those of users.
- What you data looks like?
- the relationships among the data.
- The tooling you plan to have.
- The access patterns that might emerge.

A good data model can:

- Make it easier to manage data
- Make queries more efficient
- Use less memory and Computer Processing Unit (CPU)
- Reduce cost

As a general pricipal of MongoDB data that is accessed together should be stored together.
Collections do not enforce any document structure by default. That means document even those in the same collection can have different structures.
Each document at MongoDB can be different which is known as **Polymorphism**.
Embedded document model enables us to build complex relationships among data. You can normalize your data by using database references.
The key pricipal here is how your application will use the data rather than how it's stored in the database.

2.2 Types of Data Relationships

2.2.1 Different types of relationships data can have:

- One-to-one
- One-to-many
- Many-to-Many

One-to-One

A relationship where a data entity in one set is connected to exactly one data entity in another set.

One-to-Many

A relationship where a data entity in one set is connected to any number of entities in another set.

Many-to-Many

A relationship where any number of data entities in one set are connected to any number of data entities in another set.

2.2.2 Two main ways to model these relationships:

- Embeding
- Referencing

Embeding

We take related data and insert it into our document.

Referencing

When we refer to documents in another collection in our document. Reference is made by their respective `ObjectID`.

2.3 Modeling Data Relationships

2.3.1 Embedding Data in Documents

Also known as nested document.

- Avoids application joins.
- Provides better performance for read operatinos.
- Allows developers to update related data in a single write operations.

Issues:

- Embedding data into a single document can create large documents.
- Large documents have to be read into memory in full, which can result in a slow application performance for you end user.
- Continuesly adding data without limit creates **unbounded documents.**
- Unbounded Documents may exceed the BSON document threshold of 16MB. This is schema anit-patterns which you should avoid.

2.3.2 Referencing Data in Documents

When we want to store data in two different collections but also ensure that those collections are related, we can use **References**. References save the $_id$ field of one document in another document as a link between the two. They are simple and sufficient for most use cases. Using references is called **linking** or **data normalization**. It avoid duplication of data and in most cases smaller documents, however referencing data you'll need to query from multiple documents costing extyra resources and impacts read performance.

Embedding	Referencing
✓ Single query to retrieve data	✓ No duplication
✓ Single operation to update/delete data	✓ Smaller Documents
! Data Duplication	! Need to join data from multiple documents
! Large documents	

2.4 Scaling a Data Model

Optimum efficiency of:

- query result depends on
- memory usage
- CPU usage
- storage
- When creating a data model, avoid documents that are unbounded. Document size grows infinitely. Problems as teh array grows larger in embedded document.
  - It will take up more space in memory
  - may impact write performance as entire document is rewritten into MongoDB data storage.
  - it will be difficult to perform pagination of that array.
  - Maximum document size of 16MB will lead to storage problems.
- benefit is that we can retrieve all document in single read.
- Avoid:
  - More than the document size limit of 16 MB.
  - Poor query performance.
  - Poor write performance.
  - Too much memory being used.

2.5 Using Atlas Tools for Schema Help

Schema design patterns are guidelines that help developers plan, organize, and model data. Schema anti-patterns result in suboptimal performance and non scalable solutions. Eg:

- massive arrays.
- Massive number of collections.
- Bloated documents.
- Unnecessary indexes.

- Queries without indexes.
- Data that's accessed together, but stored in different collections.

Tools in Atlas are **Data Explorer** and **Performance Advisor**

Chapter 3

Connection to MongoDB Database

3.1 Using MongoDB Connection Strings

Connection string can be used to connect from MongoShell, MongoDB Compass, or any other application. It provides standered fromat and DNS seed list format.

Standered Format

Used to connect to standalone clusters, replica sets, or sharded clusters.

DNS Seed List Format

Released in MongoDB 3.6.

3.1.1 Connection String

It allows to provide a DNS server list to connection string. Gives more flexibility of deployment. Ability to change servers in rotation without reconfiguring clients.
In atlas it is present in Database tab → Connect → Connect to application.
Connection String: `mongodb+srv://<username>:<password>@cluster0.usqsf.mongodb.net/?retryWrites=true&w=majority` If port number is not specified MongoDB will default to 27017.

3.2 Connecting to a MongoDB Atlas Cluster with the Shell

mongsh: MongoDB shell is a nodejs read–eval–print loop (REPL) environment. It gives us access to JavaScript variables, functions, conditionals, loops and Control flow statements inside shell.

```
1 const greetingArray = ["hello", "world", "welcome"];
2 const loopArray = (array) => array.forEach(el => console.log(el));
3 loopArray(greetingArray);
```

Listing 3.1: shell example

In atlas it is present in Database tab → Connect → Connect with the MongoDB Shell.

```
1 atlas clusters connectionStrings describe myAtlasClusterEDU
2 MY_ATLAS_CONNECTION_STRING=$(atlas clusters connectionStrings describe myAtlasClusterEDU | sed "1
      ↪ d")
3 mongosh -u myAtlasDBUser -p myatlas-001 &MY_ATLAS_CONNECTION_STRING
```

Listing 3.2: connect to db

3.3 Connecting to a MongoDB Atlas Cluster with Compass

In atlas it is present in Database tab → Connect → Connect using MongoDB Compass.

3.4 Connecting to a MongoDB Atlas Cluster from an Application

MongoDB drivers

Connect our application to our database by using a connect string. You can find the mongodb drivers list here.

3.5 Troubleshooting MongoDB Atlas Connection Errors

3.6 Connecting to MongoDB in Java

MongoDB maintians Java Driver for Synchronous and Asynchronous java application code.

3.6.1 Java driver for MongoDB

```
<properties>
  <mongodb-driver-sync.version>4.7.1</mongodb-driver-sync.version>
  <mongodb-crypt.version>1.4.1</mongodb-crypt.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>${mongodb-driver-sync.version}</version>
  </dependency>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-crypt</artifactId>
    <version>${mongodb-crypt.version}</version>
  </dependency>
</dependencies>
```

Listing 3.3: pom.xml MongoDB Java Driver

3.6.2 Configure the application to use the MongoDB Java Synchronous Driver

```
package com.mongodb.quickstart;

import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.MongoException;
import com.mongodb.ServerApi;
import com.mongodb.ServerApiVersion;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;

public class Connection {
  public static void main(String[] args){
    ConnectionString conncetionString = new ConnectionString("mongodb+srv://student-joe:<password>
        ↪ @sandbox.svlx7.mongodb.net/?retryWrites=true&w=majority");
    MongoClientSettings settings = MongoClientSettings.builder()
      .applyConnectionString(connectionString)
      .serverApi(ServerApi.builder()
```

```
        .version(ServerApiVersion.V1)
        .build())
      .build();
      try(MongoClient mongoClient = MongoClients.create(settings)){
        MongoDatabase database = mongoClient.getDatabase("admin");
        database.runCommand(new Document("ping", 1));
        System.out.println("Pinged your deployment. You successfully connected to MongoDB!");
      } catch (MongoException e){
        e.printStackTrace();
      }
    }
  }
}
```

Listing 3.4: Java connection

```
package com.mongodb.quickstart;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import org.bson.Document;

import java.util.ArrayList;
import java.util.List;
public class Connection {
  public static void main(String[] args){
    String conncetionString = System.getProperty("mongodb.uri");
    try(MongoClient mongoClient = MongoClients.create(connectionString)){
    MongoClientSettings settings = MongoClientSettings.builder()
      List<Document> databases = mongoClient.listDatabases().into(new ArrayList<>());
      database.forEach(db -> System.out.println(db.toJson()));
    }
  }
}
```

Listing 3.5: Java client database fetch documents

```
mvn compile exec:java -Dexec.mainClass="com.mongodb.quickstart.Connection" -Dmongodb.uri="mongodb
    ↪ +srv://m001-student:sandbox1995@cluster0.6sfcv.mongodb.net/test?w=majority"
```

Listing 3.6: maven command

Chapter 4

MongoDB CRUD Operations: Insert and Find Documents

4.1 Inserting Documents in MongoDB Collections

Their are two methods `insertOne()` and `insertMany()`.
Usage: `db.<collection>.insertOn()`. If the collection doesn't exist then MongoDB creates the collection when we use `insertOne()`. Eg:

```
db.grades.insertOne({
  student_id: 546799,
  scores: [
  {
    type: "quiz",
    score: 50,
  },
  {
    type: "homework",
    score: 70,
  }
  ]
})
```

Listing 4.1: insertOne using mongosh

```
{
  acknowledged: true,
  insertedId: ObjectId("...")
}
```

Listing 4.2: output of insertOne using mongosh

```
db.grades.insertMany([
{
  student_id: 546799,
  scores: [
  {
    type: "quiz",
    score: 50,
  },
  {
    type: "homework",
    score: 70,
  }
  ]
```

```
  },
  {
    student_id: 546800,
    scores: [
    {
      type: "quiz",
      score: 50,
    },
    {
      type: "homework",
      score: 70,
    }
    ]
  }
  ])
```

Listing 4.3: insertMany using mongosh

```
{
  acknowledged: true,
  insertedIds: [
  '0': ObjectId("..."),
  '1': ObjectId("..."),
  ]
}
```

Listing 4.4: output of insertMany using mongosh

4.2 Finding documents in MongoDB Collections

Usage: `db.<collection>.find()`

```
use training; // database
db.zips.find(); // zips are collection
// use "it" for iterating from more result
```

Listing 4.5: find method in mongosh

4.2.1 eq operator

```
    { field: { $eq: <value> } }
    # or
    { field: <value> }
```

4.2.2 in operator

```
    { field: { $in: [<value1>, <value2>, <value3>,..] } }
```

4.3 Finding Documents by using Comparison Operators

4.3.1 Usage:

```
{ field: { <operator>: <value> } }
```

Listing 4.6: Comparison Operators

4.3.2 gt: greater than

```
{ field: { $gt : <value> } }
```

Listing 4.7: greater than gt

4.3.3 lt: less than

```
{ field: { $lt : <value> } }
```

Listing 4.8: less than lt

4.3.4 gte: greater than or equal

```
{ field: { $gte : <value> } }
```

Listing 4.9: greater than equal gte

4.3.5 lte: less than or equal

```
{ field: { $lte: <value> } }
```

Listing 4.10: less than equal lte

4.4 Querying on Array elements in MongoDB

```
db.accounts.find({"products": "InvestmentStock"})
```

`$elemMatch` ensures product field is an array.

```
db.accounts.find({
   products: {
      $elemMatch: {
         <query1>,
         <query2>,
         <query3>,
         ...
      }
   }
})
```

Listing 4.11: elemMatch operator

```
db.accounts.find({
   products: {
      $elemMatch: { $eq: "InvestmentStock" }
   }
})
```

4.5 Finding Documents by Using Logical Operators

4.5.1 and operator

Explicit syntax:

```
db.accounts.find({
   $and: [
   {<expression>},
   {<expression>},
   ...
   ]
})
```

Listing 4.12: and operator explicit

Implicit syntax:

```
db.accounts.find({ {<expression>}, {<expression>} })
```

Listing 4.13: and operator implicit

4.5.2 and operator

```
db.accounts.find({
   $or: [
   {<expression>},
   {<expression>},
   ...
   ]
})
```

Listing 4.14: or operator

4.5.3 Combining OR and AND

Both operators can be combined. But we can not have two fields with same name in same JSON object. So if we try implicit and with two or queries the first or will be overwritten by second or query.

```
db.accounts.find({
   $and: [
   { $or: [
      {<expression>},
      {<expression>},
      ...
   ]},
```

```
        { $or: [
           {<expression>},
           {<expression>},
           ...
        ]}
        ]
    })
```

But if we write this in implicit and way:

```
    db.accounts.find({
        $or: [ {<expression>}, {<expression>}, ... ],
        $or: [ {<expression>}, {<expression>}, ... ]
    })
```

It will not give same output as second or will overwrite first or.

Chapter 5

MongoDB CRUD Operations: Replace and Delete Documents

5.1 Replacing a Document in MongoDB

5.1.1 replaceOne()

```
db.collection.replaceOne(filter, replacement, options)
```

Listing 5.1: replaceOne method syntax

Example:

```
db.collection.replaceOne({_id: ObjectId("...")}, {
   title: "Deep Dive into React Hooks",
   ISBN: "0-3182-1299-4",
   thumbnailUrl: "http://via.placeholder.com/640x360",
   publicationData: ISODate("2022-07-28T02:20:21.000Z"),
   authors: ["Ada Lovelace"],
})
```

Listing 5.2: replaceOne example

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Listing 5.3: replaceOne example output

5.2 Updating MongoDB Documents by Using updateOne()

```
db.collection.updateOne(<filter>, <udpate>, {options})
```

Listing 5.4: updateOne method syntax

It has main two methods `$set`

- Adds new fields and values to document.
- Replaces the value of a field with a specified value.

and `$push`

- Appends a value to an array
- if absent, $push adds the array field with the value as its element.

Example:

```
db.podcast.findOne({ title: "the MongoDB Podcast"})
db.podcast.updateOne(
   {_id: ObjectId("...")},
   { $set: {subscribers: 98563 }}
)
```

Listing 5.5: updateOne example

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Listing 5.6: updateOne example output

5.2.1 Upsert

Insert a document with provided information if matching documents don't exist. the update operations will be carried out. Example:

```
db.podcast.updateOne(
   {title: "The Developer Hub"},
   {$set: {topics: ["databases", "MongoDB"]}},
   {upsert: true}
)
```

Listing 5.7: updateOne upsert example

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 1
}
```

Listing 5.8: updateOne upsert example output

```
db.pirate.updateOne( {_id: ObjectId("...")}, {$push: {hosts: "Nico Robin"}})
```

Listing 5.9: updateOne push example

Output:

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Listing 5.10: updateOne upsert push output

```
db.pirate.updateOne( {_id: ObjectId("...")}, {$push: {hosts: {$each: ["Nico
    ↪ Robin", "Brook", "Jimbe"]}}})
```

Listing 5.11: updateOne push each example

```
db.pirate.updateOne( {common_name: "Robin Redbreast"}, {$inc: {sightings: 1},
    ↪ $set: {last_updated: new Date()}},{upsert: true})
```

Listing 5.12: updateOne inc example

5.3 Updating MongoDB Documents by Using findAndModify()

Used to return document that is just updated. Ensures that correct version of document is returned after modification. As updateOne and findOne will take two trips to update and return the document, but findAndModify will take only one. Hence, ensuring that no other thread has modified the document in between these actions.

```
db.collection.findAndModify({
   query: { _id: ObjectId("...")},
   udpate: {$inc: {downloads: 1}},
   upsert: true,
   new: true
})
```

Listing 5.13: findAndModify method syntax

`new` option is set to true so that modified document is returned else the original will be returned.

5.4 Updating MongoDB Documents by Using updateMany()

```
db.collection.updateMany(<filter>, <update>, <options>)
```

Listing 5.14: updateMany method syntax

Not and all-or-nothing operations. Will not roll back. Updates will be visible as soon as they're performed. Not appropriate for some use cases such as finance.

5.5 Deleting Documents in MongoDB

```
db.collection.deleteOne(<filter>, <options>)
```

Listing 5.15: deleteOne method syntax

```
db.collection.deleteMany(<filter>, <options>)
```

Listing 5.16: deleteMany method syntax

Chapter 6

MongoDB CRUD Operations: Modifying Query Results

6.1 Sorting and Limiting Query Results in MongoDB

Cursor is a pointer to the result set of a query. Two method in cursor to sort or limit result of query `cursor.sort()` and `cursor.limit()` `db.collection.find()` returns a cursor that points to the documents that match that query. Cursor methods which are chained to queries can be used to perform actions on the resulting set.

6.2 Returning Specific Data from a Query in MongoDB

6.3 Counting Documents in MongoDB Collection