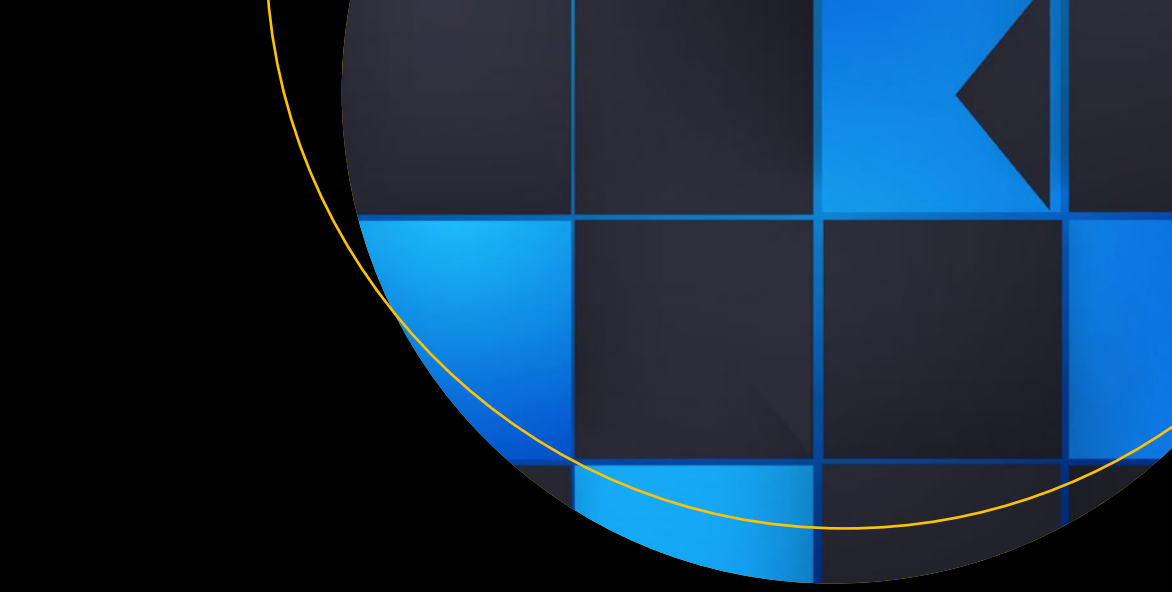


MongoDB Recipes

With Data Modeling and Query Building Strategies

Subhashini Chellappan
Dharanitharan Ganesan

Apress®

MongoDB Recipes

With Data Modeling and Query Building Strategies

Subhashini Chellappan
Dharanitharan Ganesan

Apress®

MongoDB Recipes: With Data Modeling and Query Building Strategies

Subhashini Chellappan
Bangalore, India

Dharanitharan Ganesan
Krishnagiri, Tamil Nadu, India

ISBN-13 (pbk): 978-1-4842-4890-4
ISBN-13 (electronic): 978-1-4842-4891-1
https://doi.org/10.1007/978-1-4842-4891-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestine Suresh John
Development Editor: Matthew Moodie
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4890-4. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Authors

Subhashini Chellappan is a technology enthusiast with expertise in the big data and cloud space. She has rich experience in both academia and the software industry. Her areas of interest and expertise are centered on business intelligence, big data analytics, and cloud computing.

Dharanitharan Ganesan has an MBA in technology management with high level of exposure and experience in big data—Apache Hadoop, Apache Spark, and various Hadoop ecosystem components. He has a proven track record of improving efficiency and productivity through the automation of various routine and administrative functions in business intelligence and big data technologies. His areas of interest and expertise are centered on machine learning algorithms, blockchain in big data, statistical modeling, and predictive analytics.

About the Technical Reviewer

Manoj Patil has been working in the software industry for nearly 20 years. He holds an engineering degree from COEP, Pune (India) and since then has enjoyed an exciting IT journey.

As a Principal Architect at TatvaSoft, Manoj has undertaken many initiatives in the organization ranging from training and mentoring teams leading the data science and machine learning practice, to successfully designing client solutions from different functional domains.

Starting as a Java programmer, he is one of the fortunate to have worked on multiple frameworks with multiple languages as a full-stack developer. In last five years, he has worked extensively in the field of business intelligence, big data, and machine learning with the technologies like Hitachi Vantara (Pentaho), Hadoop ecosystem, tensorflow, Python-based libraries, and more.

He is passionate about learning new technologies, trends, and reviewing books. When he is not working, he is either working out or reading infinitheism literature.

Manoj would like to thank Apress for providing this opportunity to review this title and his two daughters Ayushee and Ananyaa for their understanding during the process.

Acknowledgments

The making of this book was a journey that we are glad we undertook. The journey spanned a few months, but the experience will last a lifetime. We had our families, friends, colleagues, and other well-wishers onboard for this journey and we wish to express our deepest gratitude to each one of them.

We would like to express our special thanks to our families, friends, and colleagues who provided the support that allowed us to complete this book within a limited time frame.

Special thanks are extended to our technical reviewers for the vigilant review and filling in with their expert opinion.

We would like to thank Celestin Suresh John at Apress for signing us up for this wonderful creation. We wish to acknowledge and appreciate all our coordinating editors and the team who guided us through the entire process of preparation and publication.

Introduction

Why This Book?

MongoDB is an open source, document-oriented NoSQL database written in C++. MongoDB provides high availability, automatic scaling, and high performance. The book has the following within its scope:

- Introduction to NoSQL databases, features of MongoDB, and installation procedures
- Complete CRUD operations and MongoDB query language examples
- Demonstrations of the aggregation framework to group data and perform aggregation operations
- Indexing details to improve the performance of a query
- Replication and sharding features to increase data availability and avoid data loss in case of a failure
- Monitoring the database and backup strategies
- Security features to implement proper authentication and authorization

Who Is This Book For?

The audience for this book includes all levels of IT professionals and anyone who wants to get a good understanding of MongoDB.

How Is This Book Organized?

Chapter 1 describes NoSQL databases, various features of MongoDB, the installation procedure, and how to interact with MongoDB.

Chapter 2 describes MongoDB's rich query language to support create, read, update, and delete (CRUD) operations.

Chapter 3 describes the aggregation framework to perform aggregation operations. The aggregation framework can group data and perform a variety of operations on that grouped data.

Chapter 4 describes indexes, types of index, index properties, and the various indexing strategies to be considered. Indexes are used to improve the performance of a query.

Chapter 5 describes the various replication and sharding strategies in MongoDB. Replication is the process of creating and managing a duplicate version of a database across servers to provide redundancy and increase the availability of data. Sharding distributes data across servers to increase availability.

Chapter 6 describes the multi-document transactions feature. Multi-document transactions help us to achieve all-or-nothing execution to maintain data integrity.

Chapter 7 describes the details of various tools to monitor a MongoDB database and procedures for backup operations. These MongoDB monitoring tools help us to understand what is happening with a database at various levels.

Chapter 8 describes the various security features in MongoDB to verify the identity of the user and access controls to determine the verified user's access to resources and operations.

How Can I Get the Most Out of This Book?

It is easy to leverage this book for maximum gain by reading the chapters thoroughly. Get hands-on by following the step-by-step instructions provided in the recipes. Do not skip any of the demonstrations. If need be, repeat them a second time or until the concept is firmly etched in your mind. Happy learning!

Subhashini Chellappan
Dharanitharan Ganesan

CHAPTER 1

MongoDB Features and Installation

This chapter describes NoSQL databases, various features of MongoDB, and how to interact with MongoDB. We are going to focus on the following topics:

- Why NoSQL?
- What are NoSQL databases?
- CAP theorem.
- BASE approach.
- Types of NoSQL databases.
- MongoDB features.
- MongoDB installation on Windows.
- MongoDB installation on Linux.
- MongoDB Compass installation on Windows.
- Terms used in MongoDB.
- Data types in MongoDB.
- Database commands.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_1

The Need for NoSQL Databases

There are several database challenges for modern applications, including the following.

- Modern application storage capacity exceeds the storage capabilities of the traditional relational database management system (RDBMS).
- Modern applications require unknown level of scalability.
- Modern applications should be available 24/7.
- Data needs to be distributed globally.
- Users should be able to read and write data anywhere.
- Users always seek reduction of software and hardware costs.

All these challenges have given birth to the NoSQL databases.

What Are NoSQL Databases?

NoSQL databases are open source, nonrelational, distributed databases that allow organizations to analyze huge volumes of data.

The following are some of the key advantages of NoSQL databases:

- Availability.
- Fault tolerance.
- Scalability.

NoSQL databases have the following characteristics:

- They do not use SQL as a query language.
- Most NoSQL databases are designed to run on clusters.
- They operate without a schema, freely adding fields to the database without having to define any changes in structure first.
- They are polygot persistent, meaning there are different ways to store the data based on the requirements.
- They are designed in such a way that they can be scaled out.

CAP Theorem

The CAP theorem states that any distributed system can satisfy only two of these three properties:

- ***Consistency*** implies that every read fetches the last write.
- ***Availability*** implies that reads and writes always succeed. In other words, each nonfailing node will return a response in a reasonable amount of time.
- ***Partition tolerance*** implies that the system will continue to function even when there is a data loss or system failure.

Figure 1-1 is a graphical representation of the CAP theorem.

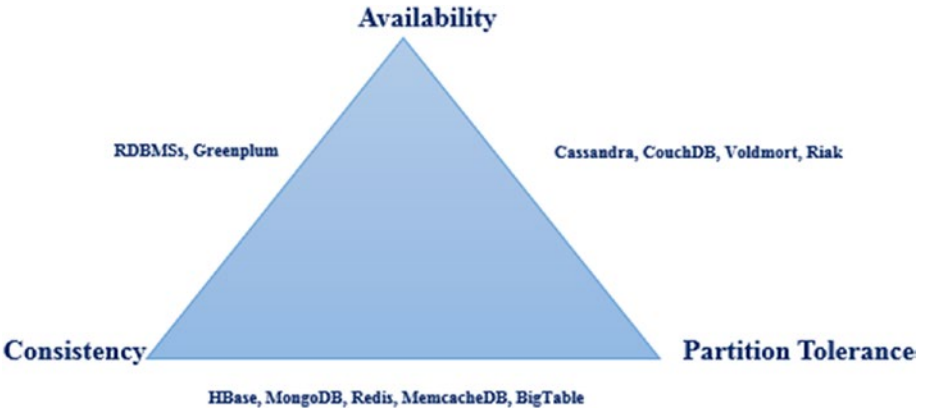


Figure 1-1. *CAP theorem*

CAP therorem categorizes a system according to three categories:

1. Consistency and availability.
2. Consistency and partition tolerance.
3. Availability and partition tolerance.

Note The partition tolerance property is a must for NoSQL databases.

BASE Approach

NoSQL databases are based on the BASE approach. BASE stands for:

- ***Basic availability:*** The database should be available most of the time.
- ***Soft state:*** Tempory inconsistency is allowed.
- ***Eventual consistency:*** The system will come to a consistent state after a certain period.

Types of NoSQL Databases

NoSQL databases are generally classified into four types, as shown in Table 1-1.

Table 1-1. *Types of NoSQL Databases*

Data Model	Example	Description
Key/Value Store	Dynamo DB, Riak	• Least complex NoSQL options. • Key and a value.
Column Store	HBase, Big Table	• Also known as wide column store. • Storing data tables as sections of columns of data.
Document Store	MongoDB, CouchDB	• Extends key/value idea. • Storing data as a document. • Complex NoSQL options. • Each document contains a unique key that is used to retrieve the document.
Graph Database	Neo4j	• Based on graph theory. • Storing data as nodes, edges, and properties.

MongoDB Features

MongoDB is an open source, document-oriented NoSQL database written in C++. MongoDB provides high availability, automatic scaling, and high performance. The following sections introduce the features of MongoDB.

Document Database

In MongoDB, a record is represented as a document. A document is a combination of field and value pairs. MongoDB documents are similar to JavaScript Object Notation (JSON) documents. Figure 1-2 represents a document.

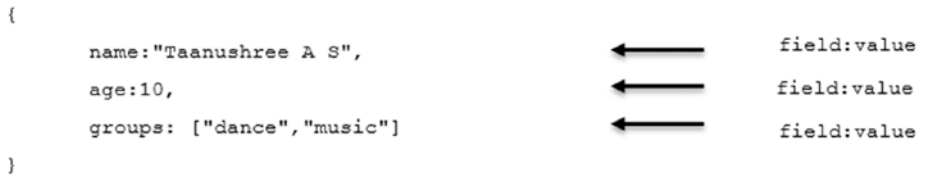


Figure 1-2. *A document*

These are the advantages for using documents.

- In many programming languages, a document corresponds to native data types.
- Embedded documents help in reducing expensive joins.

MongoDB Is Schemaless

MongoDB is a schemaless database, which means a collection (like a table in an RDBMS) can hold different documents (like records in an RDBMS). This way, MongoDB provides flexibility in dealing with the schemas in a database. Refer to the following collection, `Person`.

```
{name:"Aruna M S", age:12 } //document with two fields.
{ssn:100023412, name:"Anbu M S",groups:["sports","news"]}
//document with three fields.
```

Here, the collection `Person` has two documents, each with different fields, contents, and size.

MongoDB Uses BSON

JSON is an open source standard format for data interchange. Document databases use JSON format to store records. Here is an example of a JSON document.

```
{
      name:"John",
      addresses:[
                {
                    address:"123,River Road",
                    status:"office"
                },
                {
                    address:"345,Mount Road",
                    status:"personal"
                 }
                 ]
    }
```

MongoDB represents JSON documents in a binary-encoded format, Binary JSON (BSON), behind the scenes. BSON extends the JSON model to provide additional data types such as dates that are not supported by JSON. BSON uses the `_id` field called `ObjectId` as the primary key. The value of the `_id` field is generated either by the MongoDB service or by an application program. Here is an example of `ObjectId`:

```
"_id": ObjectId("12e6789f4b01d67d71da3211")
```

Rich Query Language

MongoDB provides a rich query language to support create, read, update, and delete (CRUD) operations, data aggregation, and text searches. Let us understand how to query a collection in MongoDB with an example. Consider the following collection, `Employee`.

```
{_id: 10001, name:"Subhashini",unit:"Hadoop"}
{_id: 10002, name:"Shobana", unit:"Spark"}
```

To display employee details, the MongoDB query would be

```
db.Employee.find();
```

We discuss MongoDB query language in detail in Chapter 2.

Aggregation Framework

MongoDB provides an aggregation framework to perform aggregation operations. The aggregation framework can group data from multiple documents and perform variety operations on that grouped data. MongoDB provides an aggregation pipeline, the `map-reduce` function, and single-purpose aggregation methods to perform aggregation operations. We discuss the aggregation framework in detail in Chapter 3.

Indexing

Indexes improve the performance of query execution. MongoDB uses indexes to limit the number of documents scanned. We discuss various indexes in Chapter 4.

GridFS

GridFS is a specification for storing and retrieving large files. GridFS can be used to store files that exceed the BSON maximum document size of 16 MB. GridFS divides the file into parts known as chunks and stores them as separate documents. GridFS divides the file into chunks of 255 KB, except the last chunk, the size of which is based on the file size.

Replication

Replication is a process of copying an instance of a database to a different database server to provide redundancy and high availability. Replication provides fault tolerance against the loss of a single database server. In MongoDB, the replica set is a group of `mongod` processes that maintain the same data set. We discuss how to create replica sets in Chapter 5.

Sharding

A single server can be a challenge when we need to work with large data sets and high throughput applications in terms of central processing unit (CPU) and inout/output (I/O) capacity. MongoDB uses sharding to support large data sets and high-throughput operations. Sharding is a method for distributing data across multiple systems, discussed in detail in Chapter 5.

The mongo Shell

The `mongo` shell is an interactive JavaScript interface to MongoDB. The `mongo` shell is used to query, update data, and perform administrative operations. The `mongo` shell is a component of MongoDB distributions. When you start the `mongod` process, the `mongo` shell will be connected to a MongoDB instance.

Terms Used in MongoDB

Let us understand the terms used in MongoDB, provided in Table 1-2.

Table 1-2. *MongoDB Terms*

Terms	MongoDB
MongoDB server	mongod
MongoDB client	mongo

Table 1-3 shows the equivalent terms used in RDBMS and MongoDB.

Table 1-3. *Equivalent Terms for RDBMS and MongoDB*

RDBMS	MongoDB
Database	Database
Table	Collection
Record	Document
Columns	Fields or key/value pairs
ACID transactions	ACID transactions
Secondary index	Secondary index
JOINs	Embedded document, `$lookup`
GROUP_BY	Aggregation pipeline

Data Types in MongoDB

Table 1-4 provides a description of the data types in MongoDB.

Table 1-4. *MongoDB Data Types*

String	Strings are UTF-8
Integer	Can be 32-bit or 64-bit
Double	To store floating-point values
Arrays	To store a list of values into one key
Timestamps	For MongoDB internal use; values are 64-bit Record when a document has been modified or added
Date	A 64-bit integer that represents the number of milliseconds since the Unix epoch (January 1, 1970)
ObjectId	Small, unique, fast to generate, and ordered Consist of 12 bytes, where the first four bytes are a timestamp that reflects the ObjectId's creation
Binary data	To store binary data (images, binaries, etc.)
Null	To store `NULL` value

Note Refer to the following link for data types in MongoDB:
`https://docs.mongodb.com/manual/reference/bson-types/`

MongoDB Installation

Let us learn how to install MongoDB.

Recipe 1-1. Install MongoDB on Windows

In this recipe, we are going to discuss how to install MongoDB on Windows.

Problem

You want to install MongoDB on Windows.

Solution

Download the MongoDB `msi` installer from https://www.mongodb.com/download-center#enterprise.

How It Works

Let's follow the steps in this section to install MongoDB on Windows.

Step 1: Install the `msi` Installer

Right-click the Windows installer, select Run as Administrator, and follow the instructions to install MongoDB.

Step 2: Create a Data Directory

Create `\data\db` directory in `C:\` to store MongoDB data. Directory looks like "`C:\data\db`".

Step 3: Start the MongoDB Server

Open the command prompt and navigate to the MongoDB installation folder. Issue the following command to start the MongoDB server, as shown in Figure 1-3.

```
mongod.exe
```

```
C:\Program Files\MongoDB\Server\4.0\bin>mongod.exe
2018-09-25T19:53:00.113+0530 I CONTROL  [main] Automatically disabling TLS 1.0, to force-
enable TLS 1.0 specify --sslDisabledProtocols 'none'
2018-09-25T19:53:00.690+0530 I CONTROL  [initandlisten] MongoDB starting : pid=552 port=2
7017 dbpath=C:\data\db\ 64-bit host=BDC8-LX-JCJPZM2
2018-09-25T19:53:00.690+0530 I CONTROL  [initandlisten] targetMinOS: Windows 7/Windows Se
rver 2008 R2
2018-09-25T19:53:00.691+0530 I CONTROL  [initandlisten] db version v4.0.2
2018-09-25T19:53:00.691+0530 I CONTROL  [initandlisten] git version: fc1573ba18aee42f97a3
bb13b67af7d837826b47
2018-09-25T19:53:00.691+0530 I CONTROL  [initandlisten] allocator: tcmalloc
```

Figure 1-3. *Starting MongoDB Server*

You should get a message that states "Waiting for connection on port 27017."

Step 4: Start the MongoDB Client

Open another command prompt and navigate to the MongoDB installation folder. Issue the following command to start the MongoDB client, as shown in Figure 1-4.

```
mongo.exe
```

```
C:\Program Files\MongoDB\Server\4.0\bin>mongo.exe
MongoDB shell version v4.0.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 4.0.2
Server has startup warnings:
2018-09-25T19:28:14.334+0530 I CONTROL  [initandlisten]
2018-09-25T19:28:14.334+0530 I CONTROL  [initandlisten] ** WARNING: Access control is
 enabled for the database.
2018-09-25T19:28:14.334+0530 I CONTROL  [initandlisten] **          Read and write acc
 to data and configuration is unrestricted.
2018-09-25T19:28:14.334+0530 I CONTROL  [initandlisten]
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and disp
```

Figure 1-4. *Starting MongoDB Client*

We can see the mongo shell.

This confirms, we have installed MongoDB on windows.

Recipe 1-2. Install MongoDB on Ubuntu

In this recipe, we are going to discuss how to install MongoDB on Ubuntu.

Problem

You want to install MongoDB on Ubuntu.

Solution

Download the MongoDB tarball from https://www.mongodb.com/download-center/community.

How It Works

Let's follow the steps in this section to install MongoDB on Ubuntu.

Step 1: Extract the tarball

Issue the following command to untar the tarball:

```
tar -xzf mongodb-linux-x86_64-ubuntu1604-4.0.8.tgz
```

Step 2: Create a Data Directory

Create a `/data/db` directory as shown in Recipe 1-1.

Step 3: Start the MongoDB Server

Open the terminal, navigate to the MongoDB installation folder, and issue the following command to start the MongoDB server.

```
./mongod --dbpath <data directory path>
```

Step 4: Start the MongoDB Client

Open another terminal, navigate to the MongoDB installation folder, and issue the following command to start the MongoDB client.

```
./mongo.exe
```

Recipe 1-3. Install MongoDB Compass on Windows

In this recipe, we are going to discuss how to install MongoDB Compass on Windows. MongoDB Compass is a simple-to-use graphical user interface (GUI) for interacting with MongoDB.

Problem

You want to install MongoDB Compass on Windows.

Solution

Download the MongoDB Compass `msi` installer from https://www.mongodb.com/download-center#compass.

How It Works

Let's follow the steps in this section to install MongoDB Compass on Windows.

Step 1: Install the MongoDB Compass `msi` Installer

Right-click the MongoDB Compass Windows installer, select Run as Administrator, and follow the instructions to install MongoDB Compass. Figure 1-5 shows the MongoDB Compass GUI.

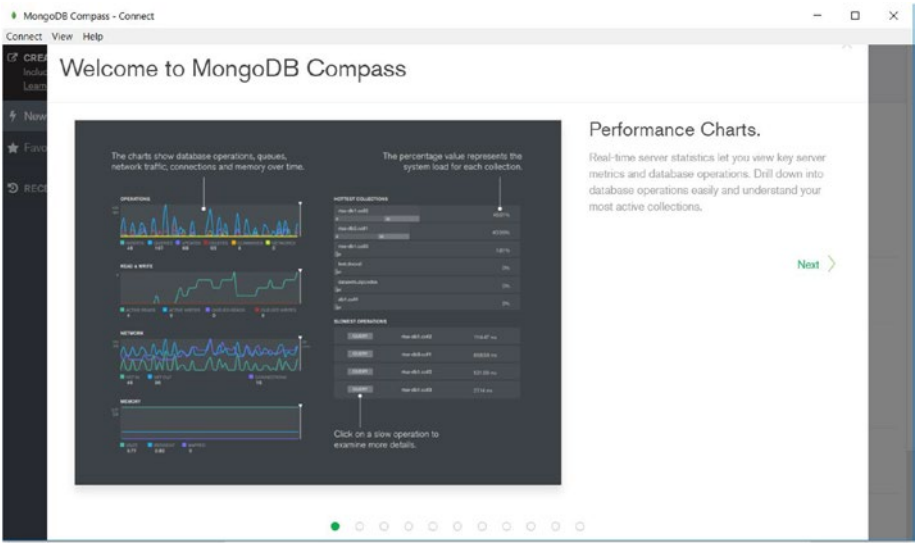


Figure 1-5. *MongoDB Compass GUI*

Step 2: Start the MongoDB Server

Open the command prompt, navigate to the MongoDB installation folder, and issue the following command to start the MongoDB server.

```
mongod.exe
```

You should see a message that states "Waiting for connection on port 27017."

Note The default port number for MongoDB is 27017.

Step 3: Connect MongoDB Compass with MongoDB Server

Click the New Connection tab. Enter the required details, as shown in Figure 1-6.

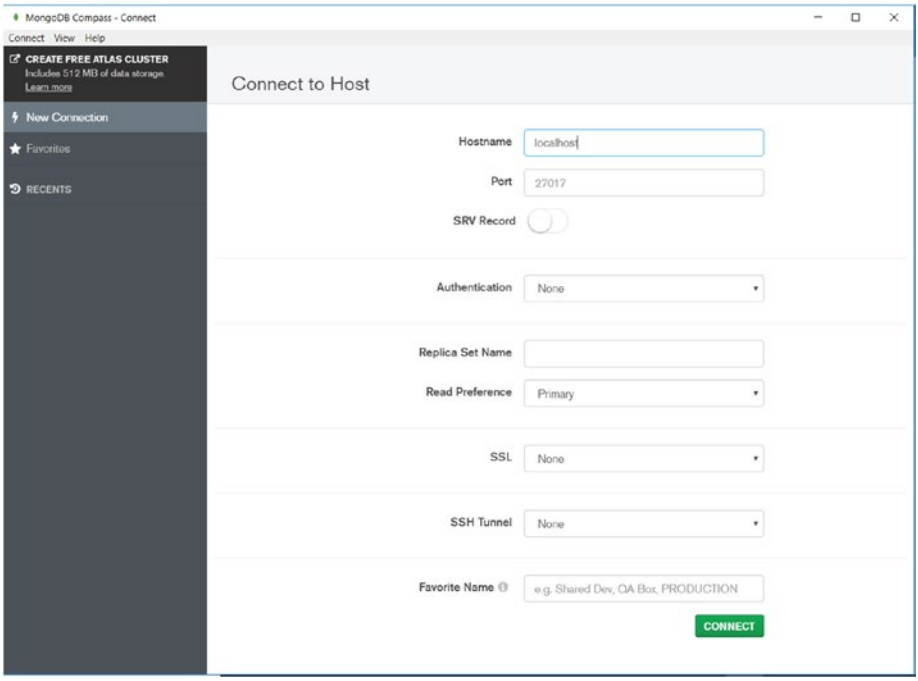


Figure 1-6. *New Connection page*

Click Connect to connect to the MongoDB Server as shown in Figure 1-7.

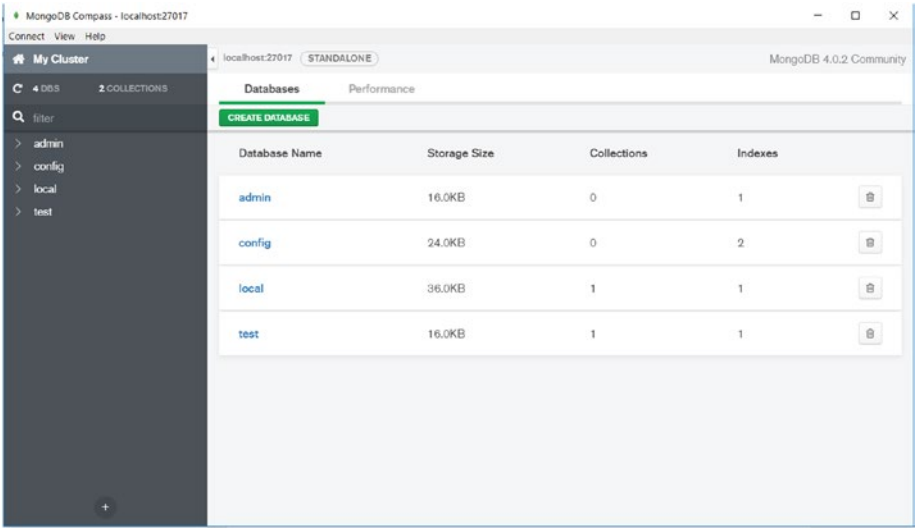


Figure 1-7. *Connecting MongoDB Compass to the MongoDB server*

We have now connected MongoDB Compass to the MongoDB server.

Working with Database Commands

Next, we discuss database commands in MongoDB.

Recipe 1-4. Create Database

In this recipe, we are going to discuss how to create a database in MongoDB.

Problem

You want to create a database in MongoDB.

Solution

Use the following syntax to create a database in MongoDB.

```
use <database name>
```

How It Works

Let's follow the steps in this section to create a database in MongoDB.

Step 1: Create a database

To create a database named `mydb`, use this command:

```
use mydb
```

Here is the output,

```
> use mydb
switched to db mydb
>
```

To confirm the existence of the database, type the following command in the `mongo` shell.

```
> db
mydb
>
```

This shows that you are working in the `mydb` database, so you know we have created a database by that name.

Recipe 1-5. Drop Database

In this recipe, we are going to discuss how to drop a database in MongoDB.

Problem

You want to drop a database in MongoDB.

Solution

Use the following syntax to drop a database in MongoDB.

```
db.dropDatabase()
```

How It Works

Let's follow the steps in this section to drop a database in MongoDB.

Step 1: Drop a Database

To drop a database, first ensure that you are working in the database that you want to drop. Use the `db.dropDatabase()` method.

```
use mydb
```

Here is the output,

```
> use mydb
switched to db mydb
>
```

```
> db.dropDatabase()
{ "ok" : 1 }
>
```

We have thus dropped the database named `mydb`.

Note If no database is selected, the default database `test` is dropped.

Recipe 1-6. Display List of Databases

In this recipe, we are going to discuss how to display a list of databases.

Problem

You want to display a list of databases.

Solution

Use the following syntax to display a list of databases.

```
show dbs
show databases
```

How It Works

Let's follow the steps in this section to display a list of databases.

Step 1: Display a List of Databases

Type the following command in the `mongo` shell.

```
show dbs
```

```
show databases
```

Here is the output,

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
test    0.000GB
>

> show databases;
admin   0.000GB
config  0.000GB
local   0.000GB
>
```

We can now see the list of databases.

Note The newly created database `mydb` is not shown in the list. This is because the database needs to have at least one collection to display in the list. The default database is `test`.

Recipe 1-7. Display the Version of MongoDB

In this recipe, we are going to discuss how to display the version of MongoDB.

Problem

You want to display the version of MongoDB.

Solution

Use the following syntax to display the version of MongoDB.

```
db.version()
```

How It Works

Let's follow the steps in this section to display the version of MongoDB.

Step 1: Display the Version of MongoDB

Type the following command in the `mongo` shell.

```
db.version()
```

Here is the output,

```
> db.version()
4.0.2
>
```

We can see that the version of MongoDB is 4.0.2.

Recipe 1-8. Display a List of Commands

In this recipe, we are going to see how to display the list of MongoDB commands.

Problem

You want to display the list of MongoDB commands.

Solution

Use the following syntax to display a list of commands.

```
db.help()
```

How It Works

Let's follow the steps in this section to display a list of commands.

Step 1: Display a List of Commands

Type the following command in the mongo shell.

```
db.help()
```

Here is the output,

```
> db.help()
dB methods:
        db.adminCommand(nameOrDocument) - switches to 'admin'
        db, and runs command [just calls db.runCommand(...)]
```

We can now see the list of MongoDB commands.

Note A few scenarios where we can apply MongoDB are e-commerce product catalogs, blogs, and content management.

In next chapter, we discuss how to perform CRUD operations using MongoDB query language.

CHAPTER 2

MongoDB CRUD Operations

In Chapter 1, we discussed MongoDB features and how to install MongoDB on Windows and Linux. We also discussed terms used in MongoDB and how to create a database. In this chapter, we are going to discuss how to perform create, read, update, and delete (CRUD) operations in MongoDB. This chapter also describes how to work with embedded document and arrays. We are going to discuss the following topics:

- Collections.
- MongoDB CRUD operations.
- Bulk write operations.
- MongoDB import and export.
- Embedded documents.
- Working with arrays.
- Array of embedded documents.
- Projection.
- Dealing with null and missing values.
- Working with `limit()` and `skip()`.
- Working with `Node.js` and MongoDB.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_2

Collections

MongoDB collections are similar to tables in RDBMS. In MongoDB, collections are created automatically when we refer to them in a command. For an example, consider the following command.

```
db.person.insert({_id:100001,name:"Taanushree A S",age:10})
```

This command creates a collection named `person` if it is not present. If it is present, it simply inserts a document into the `person` collection.

Recipe 2-1. Create a Collection

In this recipe, we are going to discuss how to create a collection.

Problem

You want to create a collection named `person` using the `db.createCollection()` command.

Solution

Use the following syntax to create a collection.

```
db.createCollection (<name>)
```

How It Works

Let's follow the steps in this section to create a collection named `person`.

Step 1: Create a Collection

To create a collection by the name person use the following command.

```
db.createCollection("person")
```

Here is the output,

```
> db.createCollection("person")
{ "ok" : 1 }
>
```

To confirm the existence of the collection, type the following command in the mongo shell.

```
> show collections
person
>
```

We can now see the collection named person.

Recipe 2-2. Create Capped Collections

In this recipe, we are going to discuss what a capped collection is and how to create one. Capped collections are fixed-size collection for which we can specify the maximum size of the collection in bytes and the number of documents allowed. Capped collections work similar to a circular buffer: Once it fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

There are some limitations of capped collections:

- You can't shard capped collections.
- You can't use the aggregation pipeline operator $out to write results to a capped collection.

- You can't delete documents from a capped collection.
- Creating an index allows you to perform efficient update operations.

Problem

You want to create a capped collection named `student` using the `db.createCollection()` method.

Solution

Use the following syntax to create a collection.

```
db.createCollection (<name>,{capped:<boolean>,size:<number>,max
:<number>})
```

How It Works

Let's follow the steps in this section to create a capped collection named `student`.

Step 1: Create a Capped Collection

To create a capped collection named `student`, use the following command.

```
db.createCollection("student",{capped: true,size:1000,max:2})
```

Here, `size` denotes the maximum size of the collection in bytes and `max` denotes the maximum number of documents allowed.
Here is the output,

```
> db.createCollection("student",{capped: true,size:1000,max:2})
{ "ok" : 1 }
>
```

To check if a collection is capped, type the following command in the mongo shell.

```
> db.student.isCapped()
true
>
```

We can see that the student collection is capped. Now, we will insert few documents into the student collection, as shown next.

```
db.student.insert([{ "_id" : 10001, "name" : "Taanushree A S",
"age" : 10 },{ "_id" : 10002, "name" : "Aruna M S", "age" : 14 }])
```

Here is the output,

```
> db.student.insert([{ "_id" : 10001, "name" : "Taanushree A
S", "age" : 10 },{ "_id" : 10002, "name" : "Aruna M S", "age" :
14 }])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
>
```

Step 2: Query a Capped Collection

To query a capped collection, use the following command.

```
db.student.find()
```

Here is the output,

```
> db.student.find()
{ "_id" : 10001, "name" : "Taanushree A S", "age" : 10 }
{ "_id" : 10002, "name" : "Aruna M S", "age" : 14 }
>
```

Here, MongoDB retrieves the results in the order of insertion. To retrieve documents in the reverse order, use `sort()` and set the `$natural` parameter to -1 as shown here.

```
> db.student.find().sort({$natural:-1})
{ "_id" : 10002, "name" : "Aruna M S", "age" : 14 }
{ "_id" : 10001, "name" : "Taanushree A S", "age" : 10 }
>
```

We can see the results are now in the reverse order.

Step 3: Insert a Document into a Full Capped Collection

Let's see what happens when we try to insert a document into a capped collection that has reached its capacity already using the following command.

```
db.student.insert({_id:10003,name:"Anba V M",age:16})
```

Here is the output,

```
> db.student.insert({_id:10003,name:"Anba V M",age:16})
WriteResult({ "nInserted" : 1 })
```

To query the `student` collection, use the following command.

```
> db.student.find()
{ "_id" : 10002, "name" : "Aruna M S", "age" : 14 }
{ "_id" : 10003, "name" : "Anba V M", "age" : 16 }
>
```

Notice that the first document is overwritten by the third document, as the maximum number of documents allowed in the `student` capped collection is two.

Create Operations

Create operations allow us to insert documents into a collection. These insert operations target a single collection. All write operations are atomic at the level of a single document.

MongoDB provides several methods for inserting documents, listed in Table 2-1.

Table 2-1. *Insert Methods*

`db.collection.insertOne`	Inserts a single document
`db.collection.insertMany`	Inserts multiple documents
`db.collection.insert`	Inserts a single document or multiple documents

Recipe 2-3. Insert Documents

In this recipe, we are going to discuss various methods to insert documents into a collection.

Problem

You want to insert documents into the collection `person`.

Solution

Use the following syntax to insert documents.

db.collection.insertOne()
db.collection.insertMany()

How It Works

Let's follow the steps in this section to insert documents into the collection `person`.

Step 1: Insert a Single Document

To insert a single document into the collection `person`, use the following command.

```
db.person.insertOne({_id:1001,name:"Taanushree AS",age:10})
```

Here is the output,

```
> db.person.insertOne({_id:1001,name:"Taanushree AS",age:10})
{ "acknowledged" : true, "insertedId" : 1001 }
>
```

`insertOne()` returns a document that contains the newly inserted document's `_id`.

If you don't specify an `_id` field, MongoDB generates an `_id` field with an `ObjectId` value. The `_id` field act as the primary key.

Here is an example:

```
> db.person.insertOne({name:"Aruna MS",age:14})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5bac7a5113572c1fb994d2fe")
}
>
```

Step 2: Insert Multiple Documents

To insert multiple documents into the collection person, use the following command.

```
db.person.insertMany([{_id:1003,name:"Anba V M",age:16},{_id:10
04,name:"shobana",age:44}])
```

Pass an array of documents to the insertMany() method to insert multiple documents.
Here is the output,

```
> db.person.insertMany([{_id:1003,name:"Anba V M",age:16},{_id:
1004,name:"shobana",age:44}])
{ "acknowledged" : true, "insertedIds" : [ 1003, 1004 ] }
>
```

Read Operations

Read operations allow us to retrieve documents from a collection. MongoDB provides the find() method to query documents.

The syntax for the find() command is

```
db.collection.find()
```

You can specify query filters or criteria inside the `find()` method to query documents based on conditions.

Recipe 2-4. Query Documents

In this recipe, we are going to discuss how to retrieve documents from a collection.

Problem

You want to retrieve documents from a collection.

Solution

Use the following syntax to retrieve documents from a collection.

```
db.collection.find()
```

How It Works

Let's follow the steps in this section to query documents in a collection.

Step 1: Select All Documents in a collection

To select all documents in a collection, use the following command.

```
db.person.find({})
```

You need to pass an empty document as a query parameter to the `find()` method.

Here is the output,

```
> db.person.find({})
{ "_id" : 1001, "name" : "Taanushree AS", "age" : 10 }
{ "_id" : ObjectId("5bac86dc773204ddade95819"), "name" : "Aruna
MS", "age" : 14 }
{ "_id" : 1003, "name" : "Anba V M", "age" : 16 }
{ "_id" : 1004, "name" : "shobana", "age" : 44 }
>
```

Step 2: Specify Equality Conditions

To specify an equality condition, you need to use `<field>:<value>` expressions in the query to filter documents.

To select a document from the collection `person` where the name equals `"shobana"`, here is the command:

```
db.person.find({name:"shobana"})
```

Here is the output,

```
> db.person.find({name:"shobana"})
{ "_id" : 1004, "name" : "shobana", "age" : 44 }
>
```

Step 3: Specify Conditions Using Query Operator

To select all documents from the collection `person` where age is greater than 10, here is the command:

```
db.person.find({age:{$gt:10}})
```

Here is the output,

```
> db.person.find({age:{$gt:10}})
{ "_id" : ObjectId("5bac86dc773204ddade95819"), "name" : "Aruna
MS", "age" : 14 }
{ "_id" : 1003, "name" : "Anba V M", "age" : 16 }
{ "_id" : 1004, "name" : "shobana", "age" : 44 }
>
```

Step 4: Specify AND Conditions

In a compound query, you can specify conditions for more than one field. To select all documents in the `person` collection where the name equals `"shobana"` and the age is greater than 10, here is the command:

```
db.person.find({ name:"shobana",age:{$gt:10}})
```

Here is the output,

```
> db.person.find({ name:"shobana",age:{$gt:10}})
{ "_id" : 1004, "name" : "shobana", "age" : 44 }

>
```

Step 5: Specify OR Conditions

You can use `$or` in a compound query to join each clause with a logical or conjunction.

The operator `$or` selects the documents from the collection that match at least one of the selected conditions.

To select all documents in the `person` collection where the name equals `"shobana"` or age equals 20, here is the command:

```
db.person.find( { $or: [ { name: "shobana" }, { age: { $eq: 20
} } ] } )
```

Here is the output,

```
> db.person.find( { $or: [ { name: "shobana" }, { age: { $eq:
20 } } ] } )
{ "_id" : 1004, "name" : "shobana", "age" : 44 }

>
```

Update Operations

Update operations allow us to modify existing documents in a collection. MongoDB provides the update methods listed in Table 2-2.

Table 2-2. *Update Methods*

db.collection.updateOne	To modify a single document
db.collection.updateMany	To modify multiple documents
db.collection.replaceOne	To replace the first matching document in the collection that matches the filter

In MongoDB, update operations target a single collection.

Recipe 2-5. Update Documents

In this recipe, we are going to discuss how to update documents.

Problem

You want to update documents in a collection.

Solution

The following methods are used to update documents in a collection.

db.collection.updateOne()
db.collection.updateMany()
db.collection.replaceOne()

Execute the following code in the mongo shell to create the student collection.

```
db.student.insertMany([{_id:1001,name:"John",marks:{english:35,
maths:38},result:"pass"},{_id:1002,name:"Jack",marks:{english:1
5,maths:18},result:"fail"},{_id:1003,name:"John",marks:{english
:25,maths:28},result:"pass"},{_id:1004,name:"James",marks:{engl
ish:32,maths:40},result:"pass"},{_id:1005,name:"Joshi",marks:{e
nglish:15,maths:18},result:"fail"},{_id:1006,name:"Jack",marks:
{english:35,maths:36},result:"pass"}])
```

How It Works

Let's follow the steps in this section to update documents in a collection.

Step 1: Update a Single Document

MongoDB provides modify operators to modify field values such as $set.

To use the update operators, pass to the update methods an update document of the form:

```
{
  <update operator>: { <field1>: <value1>, ... },
  <update operator>: { <field2>: <value2>, ... },
  ...
}
```

To update the first document where name equals joshi, execute the following command.

```
db.student.updateOne({name: "Joshi"},{$set:{"marks.english": 20}})
```

Here is the output,

```
> db.student.updateOne({name: "Joshi"},{$set:{"marks.english": 20}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
>
```

Confirm that the field marks has been updated in the student collection:

```
> db.student.find({name:"Joshi"})
{ "_id" : 1005, "name" : "Joshi", "marks" : { "english" : 20,
"maths" : 18 }, "result" : "fail" }
>
```

Note You cannot update the _id field.

Step 2: Update Multiple Documents

Use the following code to update all documents in the student collection where result equals fail.

```
db.student.updateMany( { "result":"fail" }, { $set: { "marks.
english": 20, marks.maths: 20 }  })
```

Here is the output,

```
> db.student.updateMany( { "result":"fail" }, { $set: {"marks.
english": 20, "marks.maths": 20 }})
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
>
```

Here, the modifiedCount is 2, which indicates that the preceding command modified two documents in the student collection.

Step 3: Replace a Document

You can replace the entire content of a document except the _id field by passing an entirely new document as the second argument to db.collection.replaceOne() as shown here.

Execute the following command to replace the first document from the student collection where name equals "John".

```
db.student.replaceOne( { name: "John" }, {_id:1001,name:"John",
marks:{english:36,maths:39},result:"pass"})
```

Here is the output,

```
> db.student.replaceOne( { name: "John" }, {_id:1001,name:"John
",marks:{english:36,maths:39},result:"pass"})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" :
1 }
>
```

Note Do not include update operators in the replacement document. You can omit the _id field in the replacement document because the _id field is immutable. However, if you want to include the _id field, use the same value as the current value.

Delete Operations

Delete operations allow us to delete documents from a collection. MongoDB provides two delete methods, as shown in Table 2-3.

Table 2-3. *Delete Methods*

`db.collection.deleteOne`	To delete a single document
`db.collection.updateMany`	To delete multiple documents

In MongoDB, delete operations target a single collection.

Recipe 2-6. Delete Documents

In this recipe, we discuss how to delete documents.

Problem

You want to delete documents from a collection.

Solution

The following methods are used to delete documents from a collection.

`db.collection.deleteOne()`
`db.collection.deleteMany()`

How It Works

Let's follow the steps in this section to delete documents in a collection.

Step 1: Delete Only One Document That Matches a Condition

To delete a document from the collection `student` where `name` is `John`, use this code.

```
db.student.deleteOne({name: "John"})
```

Here is the output,

```
> db.student.deleteOne({name: "John"})
{ "acknowledged" : true, "deletedCount" : 1 }
>
```

Step 2: Delete All Documents That Match a Condition

To delete all documents from the collection `student` where `name` is `Jack`, use this code.

```
db.student.deleteMany({name: "Jack"})
```

Here is the output,

```
> db.student.deleteMany({name: "Jack"})
{ "acknowledged" : true, "deletedCount" : 2 }
>
```

Step 3: Delete All Documents from a Collection

To delete all documents from the collection `student`, pass an empty `filter` document to the `db.student.deleteMany()` method.

```
db.student.deleteMany({})
```

Here is the output,

```
> db.student.deleteMany({})
{ "acknowledged" : true, "deletedCount" : 3 }
>
```

MongoDB Import and Export

The MongoDB import tool allows us to import content from JSON, comma-separated value (CSV), and tab-separated value (TSV) files. MongoDB import only supports files that are UTF-8 encoded.

The MongoDB export tool allows us to export data stored in MongoDB instances as JSON or CSV files.

Recipe 2-7. Work with Mongo Import

In this recipe, we are going to discuss how to import data from a CSV file.

Problem

You want to import data from the `student.csv` file to the `students` collection.

Solution

The following command is used to perform a Mongo import.

```
mongoimport
```

Note To work with the Mongo import command, start the `mongod` process and open another command prompt to issue the mongo import command.

How It Works

Let's follow the steps in this section to work with Mongo import.

Create a `student.csv` file using the following data in the `C:\Sample\` directory.

```
_id,name,class
1,John,II
2,James,III
3,Joshi,I
```

Execute the following command to import data from the `student.csv` file to the `students` collection.

```
mongoimport --db student --collection students --type csv
--headerline --file c:\Sample\student.csv
```

Here is the output,

```
C:\Program Files\MongoDB\Server\4.0\bin>mongoimport --db
student --collection students --type csv --headerline --file
c:\Sample\student.csv
2018-09-28T07:00:20.909+0530    connected to: localhost
2018-09-28T07:00:20.969+0530    imported 3 documents

C:\Program Files\MongoDB\Server\4.0\bin>
```

To confirm the existence of the `students` collection, issue the following commands.

```
> use student
switched to db student
> show collections
students
> db.students.find()
{ "_id" : 1, "name" : "John", "class" : "II" }
{ "_id" : 3, "name" : "Joshi", "class" : "I" }
{ "_id" : 2, "name" : "James", "class" : "III" }
>
```

Recipe 2-8. Work with Mongo Export

In this recipe, we are going to discuss how to export data from the `students` collection to a `student.json` file.

Problem

You want to export data from the `students` collection to a `student.json` file.

Solution

The following command is used to perform a Mongo export.

```
mongoexport
```

Note To work with the Mongo export command, start the `mongod` process and open another command prompt to issue the Mongo export command.

How It Works

Let's follow the steps in this section to work with Mongo export.

Execute the following command to export data from the `students` collection to a `student.json` file.

```
mongoexport --db student --collection students --out C:\Sample\
student.json
```

Here is the output,

```
C:\Program Files\MongoDB\Server\4.0\bin>mongoexport --db
student --collection students --out C:\Sample\student.json
2018-09-28T07:11:19.446+0530    connected to: localhost
2018-09-28T07:11:19.459+0530    exported 3 records
```

The confirm the export, open the `student.json` file.

```
{"_id":1,"name":"John","class":"II"}
{"_id":2,"name":"James","class":"III"}
{"_id":3,"name":"Joshi","class":"I"}
```

Embedded Documents in MongoDB

Using embedded documents allows us to embed a document within another document. Consider this example.

```
{_id:1001,name:"John",marks:{english:35,maths:38},result:"pass"}
```

Here, the `marks` field contains an embedded document.

Execute the following code to create an `employee` database and `employee` collection.

```
use employee
```

```
db.employee.insertMany([{_id:1001,name:"John",address:{previous
:"123,1st Main",current:"234,2nd Main"},unit:"Hadoop"},
{_id:1002,name:"Jack", address:{previous:"Cresent
Street",current:"234,Bald Hill Street"},unit:"MongoDB"},
{_id:1003,name:"James", address:{previous:"Cresent
Street",current:"234,Hill Street"},unit:"Spark"}])
```

Recipe 2-9. Query Embedded Documents

In this recipe, we are going to discuss how to query embedded documents.

Problem

You want to query embedded documents.

Solution

The following command is used to query embedded documents.

`db.collection.find()`

You need to pass the filter document `{<field>:<value>}` to the `find()` method where `<value>` is the document to match.

How It Works

Let's follow the steps in this section to query embedded documents.

Step 1: Match an Embedded or Nested Document

To perform an equality match on the embedded document, you need to specify the exact match document in the `<value>` document, including the field order.

To select all documents where the address field equals the document `{previous:"Cresent Street",current:"234,Bald Hill Street"}`:

```
db.employee.find( { address: { previous:"Cresent
Street",current:"234,Bald Hill Street" }} )
```

Here is the output,

```
> db.employee.find( { address: { previous:"Cresent
Street",current:"234,Bald Hill Street" }} )
{ "_id" : 1002, "name" : "Jack", "address" : { "previous" :
"Cresent Street", "current" : "234,Bald Hill Street" }, "unit"
: "MongoDB" }
>
```

Step 2: Query on a Nested Field

We can use dot notation to query a field of the embedded document.

To select all documents where the `previous` field nested in the `address` field equals `"Cresent Street"`,

```
db.employee.find( { "address.previous": "Cresent Street" } )
```

Here is the output,

```
> db.employee.find( { "address.previous": "Cresent Street" } )
{ "_id" : 1002, "name" : "Jack", "address" : { "previous" :
"Cresent Street", "current" : "234,Bald Hill Street" }, "unit"
: "MongoDB" }
{ "_id" : 1003, "name" : "James", "address" : { "previous" :
"Cresent Street", "current" : "234,Hill Street" }, "unit" :
"Spark" }
>
```

Working with Arrays

In MongoDB, we can specify field values as an array.

Recipe 2-10. Working with Arrays

In this recipe, we are going to discuss how to query an array.

Problem

You want to query an array.

Solution

The following command is used to query an array.

```
db.collection.find()
```

How It Works

Let's follow the steps in this section to query embedded documents.

Step 1: Match an Array

Execute the code shown here to create the `employeedetails` collection under the `employee` database.

```
db.employeedetails.insertMany([
   { name: "John", projects: ["MongoDB", "Hadoop","Spark"],
   scores:[25,28,29] }, { name: "James", projects: [
   "Cassandra","Spark"], scores:[26,24,23]}, { name: "Smith",
   projects: [ "Hadoop","MongoDB"], scores:[22,28,26]} ] )
```

To specify an equality condition on an array, you need to specify the exact array to match in the `<value>` of the query document `{<field>:<value>}`.

To select all documents where the `projects` value is an array with exactly two elements, `"Hadoop"` and `"MongoDB"`, in the specified order, use the following commands.

```
db.employeedetails.find( { projects: ["Hadoop", "MongoDB"] } )
```

Here is the output,

```
> db.employeedetails.find( { projects: ["Hadoop", "MongoDB"] } )
{ "_id" : ObjectId("5badcbd5f10ab299920f072a"), "name" :
"Smith", "projects" : [ "Hadoop", "MongoDB" ], "scores" : [ 22,
28, 26 ] }
>
```

Step 2: Query an Array for an Element

To query all documents where `projects` is an array that contains the string `"MongoDB"` as one of its elements, execute the following command.

```
db.employeedetails.find( { projects: "MongoDB" } )
```

Here is the output,

```
> db.employeedetails.find( { projects: "MongoDB" } )
{ "_id" : ObjectId("5badcbd5f10ab299920f0728"), "name" :
"John", "projects" : [ "MongoDB", "Hadoop", "Spark" ], "scores"
: [ 25, 28, 29 ] }
{ "_id" : ObjectId("5badcbd5f10ab299920f072a"), "name" : "Smith",
"projects" : [ "Hadoop", "MongoDB" ], "scores" : [ 22, 28, 26 ] }
>
```

Step 3: Specify Query Operators

To query all documents where the scores field is an array that contains at least one element whose value is greater than 26 we can use the $gt operator.

```
db.employeedetails.find( { scores:{$gt:26} } )
```

Here is the output,

```
> db.employeedetails.find( { scores:{$gt:26} } )
{ "_id" : ObjectId("5badcbd5f10ab299920f0728"), "name" :
"John", "projects" : [ "MongoDB", "Hadoop", "Spark" ], "scores"
: [ 25, 28, 29 ] }
{ "_id" : ObjectId("5badcbd5f10ab299920f072a"), "name" : "Smith",
"projects" : [ "Hadoop", "MongoDB" ], "scores" : [ 22, 28, 26 ] }
>
```

Step 4: Query an Array with Compound Filter Conditions on the Array Elements

A compound filter condition on the array can be specified as shown here.

```
db.employeedetails.find( { scores: { $gt: 20, $lt: 24 } } )
```

Here is the output,

```
> db.employeedetails.find( { scores: { $gt: 20, $lt: 24 } } )
{ "_id" : ObjectId("5badcbd5f10ab299920f0729"), "name" :
"James", "projects" : [ "Cassandra", "Spark" ], "scores" :
[ 26, 24, 23 ] }
{ "_id" : ObjectId("5badcbd5f10ab299920f072a"), "name" :
"Smith", "projects" : [ "Hadoop", "MongoDB" ], "scores" :
[ 22, 28, 26 ] }
>
```

Here, one element of the `scores` array can satisfy the greater than 20 condition and another element can satisfy the less than 24 condition, or a single element can satisfy both the conditions.

Step 5: Using the `$elemMatch` operator

The `$elemMatch` operator allows us to specify multiple conditions on the array elements such that at least one array element should satisfy all of the specified conditions.

```
db.employeedetails.find( { scores: { $elemMatch: { $gt: 23,
$lt: 27 } } } )
```

Here is the output,

```
> db.employeedetails.find( { scores: { $elemMatch: { $gt: 23,
$lt: 27 } } } )
{ "_id" : ObjectId("5badcbd5f10ab299920f0728"), "name" :
"John", "projects" : [ "MongoDB", "Hadoop", "Spark" ], "scores"
: [ 25, 28, 29 ] }
{ "_id" : ObjectId("5badcbd5f10ab299920f0729"), "name" : "James",
"projects" : [ "Cassandra", "Spark" ], "scores" : [ 26, 24, 23 ] }
{ "_id" : ObjectId("5badcbd5f10ab299920f072a"), "name" : "Smith",
"projects" : [ "Hadoop", "MongoDB" ], "scores" : [ 22, 28, 26 ] }
>
```

Step 6: Query an Array Element by Index Position

Use dot notation to query an array element by its index position.

Use this code to select all documents where the third element in the `scores` array is greater than 26.

```
db.employeedetails.find( { "scores.2": { $gt: 26 } } )
```

Here is the output,

```
> db.employeedetails.find( { "scores.2": { $gt: 26 } } )
{ "_id" : ObjectId("5badcbd5f10ab299920f0728"), "name" :
"John", "projects" : [ "MongoDB", "Hadoop", "Spark" ], "scores"
: [ 25, 28, 29 ] }
>
```

Step 7: Using the `$size` Operator

You can use the `$size` operator to query an array by number of elements.

To select all documents where the `projects` array has two elements, use this code.

```
db.employeedetails.find( { "projects": { $size: 2 } } )
```

Here is the output,

```
> db.employeedetails.find( { "projects": { $size: 2 } } )
{ "_id" : ObjectId("5badcbd5f10ab299920f0729"), "name" :
"James", "projects" : [ "Cassandra", "Spark" ], "scores" : [
26, 24, 23 ] }
{ "_id" : ObjectId("5badcbd5f10ab299920f072a"), "name" :
"Smith", "projects" : [ "Hadoop", "MongoDB" ], "scores" : [ 22,
28, 26 ] }
```

Step 8: Using the `$push` Operator

You can use the `$push` operator to add a value to an array.

To add work location details, use this code.

```
db.employeedetails.update({name:"James"},{$push:{Location: "US"}})
```

Here is the query to check the result:

```
> db.employeedetails.find({name:"James"})
{ "_id" : ObjectId("5c04bef3540e90478dd92f4e"), "name" :
"James", "projects" : [ "Cassandra", "Spark" ], "scores" :
[ 26, 24, 23 ], "Location" : [ "US" ] }
```

To append multiple values to an array, use the `$each` operator.

```
db.employeedetails.update({name: "Smith"},{$push:{Location:{$ea
ch:["US","UK"]}}})
```

Here is the query to check the result:

```
> db.employeedetails.find({name:"Smith"})
{ "_id" : ObjectId("5c04bef3540e90478dd92f4f"), "name" :
"Smith", "projects" : [ "Hadoop", "MongoDB" ], "scores" : [ 22,
28, 26 ], "Location" : [ "US", "UK" ] }
>
```

Step 9: Using the `$addToSet` Operator

You can use the `$addToSet` operator to add a value to an array. `$addToSet` adds a value to an array only if a value is not present. If a value is present, it does not do anything.

To add `hobbies` to the `employeedetails` collection, use this code.

```
db.employeedetails.update( {name: "James"},  { $addToSet:
{hobbies: [ "drawing", "dancing"]} })
```

Here is the query to check the result:

```
> db.employeedetails.find({name:"James"})
{ "_id" : ObjectId("5c04bef3540e90478dd92f4e"), "name" : "James",
"projects" : [ "Cassandra", "Spark" ], "scores" : [ 26, 24, 23 ],
"Location" : [ "US" ], "hobbies" : [ [ "drawing", "dancing" ] ] }
>
```

Step 10: Using the $pop Operator

You can use the $pop operator to remove the first or last element of an array.

To remove the first element in the scores array, use this code.

```
db.employeedetails.update( {name: "James"},{ $pop:
{scores:-1}})
```

Here is the query to check the result:

```
> db.employeedetails.find({name:"James"})
{ "_id" : ObjectId("5c04bef3540e90478dd92f4e"), "name" :
"James", "projects" : [ "Cassandra", "Spark" ], "scores" : [
24, 23 ], "Location" : [ "US" ], "hobbies" : [ [ "drawing",
"dancing" ] ] }
>
```

To remove the last element in the scores array, use this code.

```
db.employeedetails.update( {name: "James"},{ $pop: {scores:1}})
```

Here is the query to check the result:

```
> db.employeedetails.find({name:"James"})
{ "_id" : ObjectId("5c04bef3540e90478dd92f4e"), "name" :
"James", "projects" : [ "Cassandra", "Spark" ], "scores" : [ 24 ],
"Location" : [ "US" ], "hobbies" : [ [ "drawing", "dancing" ] ] }
>
```

Recipe 2-11. Query an Array of Embedded Documents

In this recipe, we are going to discuss how to query an array of embedded documents.

Problem

You want to query an array of embedded documents.

Solution

The following command is used to query an array of embedded documents.

```
db.collection.find()
```

How It Works

Let's follow the steps in this section to query an array of embedded documents.

Step 1: Query for a Document Nested in an Array

Execute the following code to create a `student` database and `studentmarks` collection.

```
> use student
```

```
> db.studentmarks.insertMany([{name:"John",marks:[{cla
ss: "II", total: 489},{ class: "III", total: 490 }]},{name:
"James",marks:[{class: "III", total: 469 },{class: "IV",
total: 450}]},{name:"Jack",marks:[{class:"II", total: 489 },{
```

```
class: "III", total: 390}]},{name:"Smith", marks:[{class:
"III", total: 489}, {class: "IV", total: 490}]},
{name:"Joshi",marks:[{class: "II", total: 465}, { class: "III",
total: 470}]}])
```

The command shown here selects all documents in the array field `marks` that match the specified document.

```
db.studentmarks.find( { "marks": {class: "II", total: 489}})
```

Here is the output,

```
> db.studentmarks.find( { "marks": {class: "II", total: 489}})
{ "_id" : ObjectId("5bae10e6f10ab299920f073f"), "name" :
"John", "marks" : [ { "class" : "II", "total" : 489 }, {
"class" : "III", "total" : 490 } ] }
{ "_id" : ObjectId("5bae10e6f10ab299920f0741"), "name" : "Jack",
"marks" : [ { "class" : "II", "total" : 489 }, { "class" :
"III", "total" : 390 } ] }
>
```

Note Equality matching on the array of an embedded document requires an exact match of the specified document, including field order.

Step 2: Query for a Field Embedded in an Array of Documents

You can use dot notation to query a field embedded in an array of documents. The following command selects all documents in the `marks` array where at least one embedded document contains the field `total` that has a value that is less than 400.

```
db.studentmarks.find( { 'marks.total': { $lt: 400 } } )
```

Here is the output,

```
> db.studentmarks.find( { 'marks.total': { $lt: 400 } } )
{ "_id" : ObjectId("5bae10e6f10ab299920f0741"), "name" :
"Jack", "marks" : [ { "class" : "II", "total" : 489 }, {
"class" : "III", "total" : 390 } ] }
>
```

Step 3: Array Index to Query for a Field in the Embedded Document

The command shown here selects all the documents in the `marks` array where the first element of the document contains the field `class`, which has a value that is equal to `"II"`.

```
db.studentmarks.find( { 'marks.0.class': "II" } )
```

Here is the output,

```
> db.studentmarks.find( { 'marks.0.class': "II" } )
{ "_id" : ObjectId("5bae10e6f10ab299920f073f"), "name" :
"John", "marks" : [ { "class" : "II", "total" : 489 },
{ "class" : "III", "total" : 490 } ] }
{ "_id" : ObjectId("5bae10e6f10ab299920f0741"), "name" :
"Jack", "marks" : [ { "class" : "II", "total" : 489 },
{ "class" : "III", "total" : 390 } ] }
{ "_id" : ObjectId("5bae10e6f10ab299920f0743"), "name" :
"Joshi", "marks" : [ { "class" : "II", "total" : 465 },
{ "class" : "III", "total" : 470 } ] }
>
```

Project Fields to Return from Query

By default, queries in MongoDB return all fields in matching documents. You can restrict the fields to be returned using a projection document.

Execute the following code to create a `studentdetails` collection under the `student` database.

```
db.studentdetails.insertMany( [ {name: "John", result: "pass",
marks: { english: 25, maths: 23, science: 25 }, grade: [ {
class: "A", total: 73 } ] },{name: "James", result: "pass",
marks: { english: 24, maths: 25, science: 25 }, grade: [ {
class: "A", total: 74 } ] },{name: "James", result: "fail",
marks: { english: 12, maths: 13, science: 15 }, grade: [ {
class: "C", total: 40 } ] }])
```

Recipe 2-12. Restricting the Fields Returned from a Query

In this recipe, we are going to discuss how to restrict the fields to return from a query.

Problem

You want to restrict the fields to return from a query.

Solution

The following command is used to restrict the fields to return from a query.

```
db.collection.find()
```

How It Works

Let's follow the steps in this section to restrict the fields to return from a query.

Step 1: Return the Specified Fields and the _id Field Only

You can project the required fields by setting the <field> value to 1 in the projection document.

The command shown here returns only the _id, marks, and result fields where name equals "John" in the result set.

```
db.studentdetails.find( { name: "John" }, { marks: 1, result: 1
} )
```

Here is the output,

```
> db.studentdetails.find( { name: "John" }, { marks: 1, result:
1 } )
{ "_id" : ObjectId("5bae3ed2f10ab299920f0744"), "result" :
"pass", "marks" : { "english" : 25, "maths" : 23, "science" :
25 } }
>
```

Step 2: Suppress the _id Field

You can suppress the _id field by setting its exclusion <field> to 0 in the projection document.

The following command suppresses the _id field in the result set.

```
db.studentdetails.find( { name: "John" }, { marks: 1, result:
1,_id:0 } )
```

Here is the output,

```
> db.studentdetails.find( { name: "John" }, { marks: 1, result:
1,_id:0 } )
{ "result" : "pass", "marks" : { "english" : 25, "maths" : 23,
"science" : 25 } }
>
```

Step 3: Exclude More Than One Field

You can exclude fields by setting <field> to 0 in the projection document.

The following command suppresses the _id and result fields in the result set.

```
db.studentdetails.find( { name: "John" }, { result:0,_id:0 } )
```

Here is the output,

```
> db.studentdetails.find( { name: "John" }, { result:0,_id:0 }
)
{ "name" : "John", "marks" : { "english" : 25, "maths" : 23,
"science" : 25 }, "grade" : [ { "class" : "A", "total" : 73 } ]
}
>
```

Step 4: Return a Specific Field in an Embedded Document

You can use dot notation to refer to the embedded field and set the <field> to 1 in the projection document.

Here is an example:

```
db.studentdetails.find( { name: "John }, { result:1, grade: 1,
"marks.english": 1 })
```

Here is the output,

```
> db.studentdetails.find( { name: "John" }, { result:1, grade:
1, "marks.english": 1 })
{ "_id" : ObjectId("5bae3ed2f10ab299920f0744"), "result" : "pass",
"marks" : { "english" : 25 }, "grade" : [ { "class" : "A",
"total" : 73 } ] }
>
```

Query for Null or Missing Fields

Execute the following code to create the `sample` collection.

```
db.sample.insertMany([ { _id: 1, name: null }, { _id: 2 }])
```

Recipe 2-13. To Query Null or Missing Fields

In this recipe, we are going to discuss how to query null or missing values in MongoDB.

Problem

You want to query null or missing values.

Solution

The following command is used to query null or missing values.

`db.collection.find()`

How It Works

Let's follow the steps in this section to query null or missing values.

Step 1: Equality Filter

The {name:null} query matches documents that either contain the name field with a value of null or do not contain the name field.

```
db.sample.find( { name: null } )
```

Here is the output,

```
> db.sample.insertMany([{ _id: 1, name: null }, { _id: 2 }])
{ "acknowledged" : true, "insertedIds" : [ 1, 2 ] }
>
```

Here, it returns both the documents.

Step 2: Type Check

The query shown here returns the documents that only contain the name field with a value of null. In BSON, set the null value to 10.

```
db.sample.find( { name: { $type: 10 } } )
```

Here is the output,

```
> db.sample.find( { name: { $type: 10 } } )
{ "_id" : 1, "name" : null }
>
```

Step 3: Existence Check

You can use the `$exist` operator to check for the existence of a field.

The following query returns the documents that do not contain the `name` field.

```
db.sample.find( { name: { $exists: false } } )
```

Here is the output,

```
> db.sample.find( { name: { $exists: false } } )
{ "_id" : 2 }
>
```

Iterate a Cursor

The `db.collection.find()` method of MongoDB returns a cursor. You need to iterate a cursor to access the documents. You can manually iterate a cursor in the `mongo` shell by assigning a cursor returned from the `find()` method to a variable using the `var` keyword.

If you are not assigning a cursor to a variable using the `var` keyword, it is automatically iterated up to 20 times to print the first 20 documents.

Recipe 2-14. Iterate a Cursor

In this recipe, we are going to discuss how to iterate a cursor in the `mongo` shell.

Problem

You want to iterate a cursor in the `mongo` shell.

Solution

The following syntax is used to iterate a cursor.

```
var myCursor = db.collection.find()
myCursor
```

How It Works

Let's follow the steps in this section to iterate a cursor. First, create a `numbers` collection.

```
db.numbers.insert({_id:1,number:1});
db.numbers.insert({_id:2,number:2});
db.numbers.insert({_id:3,number:3});
db.numbers.insert({_id:4,number:4});
db.numbers.insert({_id:5,number:5});
db.numbers.insert({_id:6,number:6});
db.numbers.insert({_id:7,number:7});
db.numbers.insert({_id:8,number:8});
db.numbers.insert({_id:9,number:9});
db.numbers.insert({_id:10,number:10});
db.numbers.insert({_id:11,number:11});
db.numbers.insert({_id:12,number:12});
db.numbers.insert({_id:13,number:13});
db.numbers.insert({_id:14,number:14});
db.numbers.insert({_id:15,number:15});
db.numbers.insert({_id:16,number:16});
db.numbers.insert({_id:17,number:17});
db.numbers.insert({_id:18,number:18});
db.numbers.insert({_id:19,number:19});
db.numbers.insert({_id:20,number:20});
db.numbers.insert({_id:21,number:21});
```

```
db.numbers.insert({_id:22,number:22});
db.numbers.insert({_id:23,number:23});
db.numbers.insert({_id:24,number:24});
db.numbers.insert({_id:25,number:25});
```

Issue the following commands to iterate a cursor.

```
var myCursor = db.numbers.find()
myCursor
```

Here is the output,

```
> var myCursor = db.numbers.find()
> myCursor
{ "_id" : 1, "number" : 1 }
{ "_id" : 2, "number" : 2 }
{ "_id" : 3, "number" : 3 }
{ "_id" : 4, "number" : 4 }
{ "_id" : 5, "number" : 5 }
{ "_id" : 6, "number" : 6 }
{ "_id" : 7, "number" : 7 }
{ "_id" : 8, "number" : 8 }
{ "_id" : 9, "number" : 9 }
{ "_id" : 10, "number" : 10 }
{ "_id" : 11, "number" : 11 }
{ "_id" : 12, "number" : 12 }
{ "_id" : 13, "number" : 13 }
{ "_id" : 14, "number" : 14 }
{ "_id" : 15, "number" : 15 }
{ "_id" : 16, "number" : 16 }
{ "_id" : 17, "number" : 17 }
{ "_id" : 18, "number" : 18 }
{ "_id" : 19, "number" : 19 }
```

```
{ "_id" : 20, "number" : 20 }
Type "it" for more

> it
{ "_id" : 21, "number" : 21 }
{ "_id" : 22, "number" : 22 }
{ "_id" : 23, "number" : 23 }
{ "_id" : 24, "number" : 24 }
{ "_id" : 25, "number" : 25 }
>
```

You can also use the cursor method `next()` to access the document.

```
var myCursor=db.numbers.find({});
while(myCursor.hasNext()){
  print(tojson(myCursor.next()));
}
```

Here is the output,

```
> var myCursor=db.numbers.find({});
> while(myCursor.hasNext()){
... print(tojson(myCursor.next()));
... }
{ "_id" : 1, "number" : 1 }
{ "_id" : 2, "number" : 2 }
{ "_id" : 3, "number" : 3 }
{ "_id" : 4, "number" : 4 }
{ "_id" : 5, "number" : 5 }
{ "_id" : 6, "number" : 6 }
{ "_id" : 7, "number" : 7 }
{ "_id" : 8, "number" : 8 }
{ "_id" : 9, "number" : 9 }
{ "_id" : 10, "number" : 10 }
```

```
{ "_id" : 11, "number" : 11 }
{ "_id" : 12, "number" : 12 }

{ "_id" : 13, "number" : 13 }
{ "_id" : 14, "number" : 14 }
{ "_id" : 15, "number" : 15 }
{ "_id" : 16, "number" : 16 }
{ "_id" : 17, "number" : 17 }
{ "_id" : 18, "number" : 18 }
{ "_id" : 19, "number" : 19 }
{ "_id" : 20, "number" : 20 }
{ "_id" : 21, "number" : 21 }
{ "_id" : 22, "number" : 22 }
{ "_id" : 23, "number" : 23 }
{ "_id" : 24, "number" : 24 }
{ "_id" : 25, "number" : 25 }
>
```

You can also use `forEach` to iterate the cursor and access the document.

```
var myCursor = db.numbers.find()
myCursor.forEach(printjson);
```

```
> var myCursor = db.numbers.find()
> myCursor.forEach(printjson);
{ "_id" : 1, "number" : 1 }
{ "_id" : 2, "number" : 2 }
{ "_id" : 3, "number" : 3 }
{ "_id" : 4, "number" : 4 }
{ "_id" : 5, "number" : 5 }
{ "_id" : 6, "number" : 6 }
{ "_id" : 7, "number" : 7 }
```

```
{ "_id" : 8, "number" : 8 }
{ "_id" : 9, "number" : 9 }
{ "_id" : 10, "number" : 10 }
{ "_id" : 11, "number" : 11 }
{ "_id" : 12, "number" : 12 }
{ "_id" : 13, "number" : 13 }
{ "_id" : 14, "number" : 14 }
{ "_id" : 15, "number" : 15 }
{ "_id" : 16, "number" : 16 }
{ "_id" : 17, "number" : 17 }
{ "_id" : 18, "number" : 18 }
{ "_id" : 19, "number" : 19 }
{ "_id" : 20, "number" : 20 }
{ "_id" : 21, "number" : 21 }
{ "_id" : 22, "number" : 22 }
{ "_id" : 23, "number" : 23 }
{ "_id" : 24, "number" : 24 }
{ "_id" : 25, "number" : 25 }
>
```

Working with the `limit()` and `skip()` Methods

The `limit()` method is used to limit the number of documents in the query results and the `skip()` method skips the given number of documents in the query result.

Recipe 2-15. `limit()` and `skip()` Methods

In this recipe, we are going to discuss how to work with the `limit()` and `skip()` methods.

Problem

You want to limit and skip the documents in a collection.

Solution

The following syntax is used to limit the number of documents in the query result.

```
db.collection.find().limit(2)
```

The following syntax is used to skip a given number of documents in the query result.

```
db.collection.find().skip(2)
```

How It Works

Let's follow the steps in this section to work with the `limit()` and `skip()` methods. Consider the `numbers` collection created in Recipe 2-14.

To display the first two documents, use this code.

```
db.numbers.find().limit(2)
```

Here is the output,

```
> db.numbers.find().limit(2)
{ "_id" : 1, "number" : 1 }
{ "_id" : 2, "number" : 2 }
>
```

To skip the first five documents, use this code.

```
db.numbers.find().skip(5)
```

Here is the output,

```
> db.numbers.find().skip(5)
{ "_id" : 6, "number" : 6 }
{ "_id" : 7, "number" : 7 }
{ "_id" : 8, "number" : 8 }
{ "_id" : 9, "number" : 9 }
{ "_id" : 10, "number" : 10 }
{ "_id" : 11, "number" : 11 }
{ "_id" : 12, "number" : 12 }
{ "_id" : 13, "number" : 13 }
{ "_id" : 14, "number" : 14 }
{ "_id" : 15, "number" : 15 }
{ "_id" : 16, "number" : 16 }
{ "_id" : 17, "number" : 17 }
{ "_id" : 18, "number" : 18 }
{ "_id" : 19, "number" : 19 }
{ "_id" : 20, "number" : 20 }
{ "_id" : 21, "number" : 21 }
{ "_id" : 22, "number" : 22 }
{ "_id" : 23, "number" : 23 }
{ "_id" : 24, "number" : 24 }
{ "_id" : 25, "number" : 25 }
>
```

Working with Node.js and MongoDB

MongoDB is one of the most popular databases used with Node.js.

Recipe 2-16. Node.js and MongoDB

In this recipe, we are going to discuss how to work with Node.js and MongoDB.

Problem

You want to perform CRUD operations using Node.js.

Solution

Download the Node.js installer from https://nodejs.org/en/download/ and install it.

Note The given link might be changed in the future.

Next, open the command prompt and issue the below command to install mongodb database driver.

```
npm install mongodb
```

How It Works

Let's follow the steps in this section to work with Node.js and MongoDB.

Step 1: Establishing a Connection and Creating a Collection

Save the following code in a file named connect.js.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
```

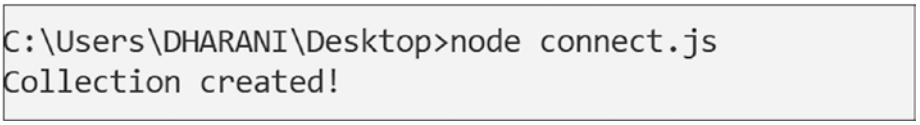
```
MongoClient.connect(url,{useNewUrlParser:true,useUnifiedTopolog
y:true},function(err, db) {
  if (err) throw err;
  var mydb = db.db("mydb");
  mydb.createCollection("employees", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

Run the code by issuing this command at the terminal.

```
node connect.js
```

Here is the output,

```
C:\Users\DHARANI\Desktop>node connect.js
Collection created!
```



Step 2: Insert a Document

Save the code shown here in a file called `insert.js`.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, {useNewUrlParser:true, useUnifiedTopol
ogy:true},function(err, db) {
  if (err) throw err;
  var mydb = db.db("mydb");
```

```
  var myobj = { name: "Subhashini", address: "E 603 Alpyne" };
  mydb.collection("employees").insertOne(myobj, function
  (err, res) {
    if (err) throw err;
    console.log("Inserted");
    db.close();
  });
});
```

Run the code by issuing the following command at the command prompt.

```
node insert.js
```

Here is the output,

```
> show collections
mydb
> db.employees.find()
{ "_id" : ObjectId("5c050dd8c63a253134cbd375"), "name" :
"Subhashini", "address" : "E 603 Alpyne" }
>
```

To insert multiple documents, use this command.

```
var myobj = [{name:"Shobana",address: "E 608
Alpyne"},{name:"Taanu", address:"Valley Street"}];
```

Step 3: Query a Document

Save the following code in a file called `find.js`.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
```

```
MongoClient.connect(url, {useNewUrlParser:true,
useUnifiedTopology:true}, function(err, db) {
  if (err) throw err;
  var mydb = db.db("mydb");
  var query = {name:"Subhashini" };
  mydb.collection("employees").find(query).toArray(function
  (err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Run the code by issuing this command at the command prompt.

```
node find.js
```

Here is the output,

```
 [ { _id: 5c050dd8c63a253134cbd375,
    name: 'Subhashini',
    address: 'E 603 Alpyne' } ]
```

Step 4: Update a Document

Save the code shown here in a file called `update.js`.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, {useNewUrlParser:true,
useUnifiedTopology:true}, function(err, db) {
  if (err) throw err;
  var mydb = db.db("mydb");
  var query = {address:"E 603 Alpyne" };
```

```
  var newvalues = { $set: {address:"New Street"}};
  mydb.collection("employees").updateOne(query, newvalues,
  function(err, res) {
    if (err) throw err;
    console.log("Document Updated");
    db.close();
  });
});
```

Run the code by issuing the following command at the command prompt.

```
node update.js
```

Here is the output,

```
> db.employees.find({name:"Subhashini"})
{ "_id" : ObjectId("5c050dd8c63a253134cbd375"), "name" :
"Subhashini", "address" : "New Street" }
>
```

Step 5: Delete a Document

Save the code shown here in a file called `delete.js`.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, {useNewUrlParser:true,
useUnifiedTopology:true}, function(err, db) {
  if (err) throw err;
  var mydb = db.db("mydb");
  var query = {name:"Subhashini"};
  mydb.collection("employees").deleteOne(query, function(err, obj) {
    if (err) throw err;
```

```
    console.log("Document deleted");
    db.close();
  });
});
```

Run the code by issuing the following command at the command prompt.

```
node delete.js
```

Here is the output,

```
Document deleted
```

CHAPTER 3

Data Modeling and Aggregation

In Chapter 2, we discussed MongoDB CRUD operations, embedded documents, and arrays. In this chapter, we cover the following topics.

- Data models.
- Data model relationship between documents.
- Modeling tree structures.
- Aggregation operations.
- SQL aggregation terms and corresponding MongoDB aggregation operations.

Data Models

MongoDB provides two data model designs for data modeling:

- Embedded data models.
- Normalized data models.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_3

Embedded Data Models

In MongoDB, you can embed related data in a single document. This schema design is known as *denormalized* models. Consider the example shown in Figure 3-1.

Figure 3-1. *A denormalized model*

This embedded document model allows applications to store the related piece of information in the same records. As a result, the application requires only few queries and updates to complete common operations.

We can use embedded documents to represent both one-to-one relationships (a "contains" relationship between two entities) and one-to-many relationships (when many documents are viewed in the context of one parent).

Embedded documents provide better results in these cases:

- For read operations.
- When we need to retrieve related data in a single database operation.

Embedded data models update related data in a single atomic operation. Embedded document data can be accessed using dot notation.

Normalized Data Models

Normalized data models describe relationships using references, as illustrated in Figure 3-2.

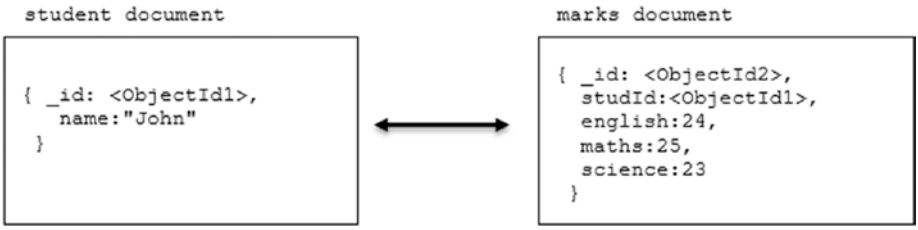


Figure 3-2. *Normalized data model*

Normalized data models can best be used in the following circumstances:

- When embedding data model results duplication of data.
- To represent complex many-to-many relationships.
- To model large hierarchical data sets.

Normalized data models do not provide good read performance.

Data Model Relationship Between Documents

Let's explore a data model that uses an embedded document and references.

Recipe 3-1. Data Model Using an Embedded Document

In this recipe, we are going to discuss a data model using an embedded document.

Problem

You want to create a data model for a one-to-one relationship.

Solution

Use an embedded document.

How It Works

Let's follow the steps in this section to design a data model for a one-to-one relationship.

Step 1: One-to-One Relationships

Consider this example.

```
{
   _id: "James",
   name: "James William"
}

{
   student_id: "James",
   street: "123 Hill Street",
   city: "New York",
   state: "US",
 }
```

Here, we have student and address relationships in which an address belongs to the student. If we are going to retrieve address data with the name frequently, then referencing requires multiple queries to resolve references. In this scenario, we can embed address data with the student data to provide a better data model, as shown here.

```
{
   _id: "James",
   name: "James William",
   address: {
              street: "123 Hill Street",
              city: "New York",
              state: "US",
              }
}
```

With this data model, we can retrieve complete student information with one query.

Step 2: One-to-Many Relationships

Consider this example.

```
{
   _id: "James",
   name: "James William"
}
```

```
{
   student_id: "James",
   street: "123 Hill Street",
   city: "New York",
   state: "US",
 }
```

```
{
   student_id: "James",
   street: "234 Thomas Street",
   city: "New Jersey",
   state: "US",
 }
```

Here, we have a student and multiple address relationships (a student has multiple addresses). If we are going to retrieve address data with the name frequently, then referencing requires multiple queries to resolve references. In this scenario, the optimal way to design the schema is to embed address data with the student data as shown here.

```
{
   _id: "James",
   name: "James William",
   address: [{
              street: "123 Hill Street",
              city: "New York",
              state: "US",
              },
             {
              street: "234 Thomas Street",
              city: "New Jersey",
              state: "US",
             }]
 }
```

This data model allows us to retrieve complete student information with one query.

Recipe 3-2. Data Model Using Document References

In this recipe, we are going to discuss a data model using document references.

Problem

You want to create a data model for a one-to-many relationship.

Solution

Use a document reference.

How It Works

Let's follow the steps in this section to design a data model for a one-to-many relationship.

Step 1: One-to-Many Relationships

Consider the following data model that maps a publisher and book relationship.

```
{
   title: "Practical Apache Spark",
   author: [ "Subhashini Chellappan", "Dharanitharan Ganesan" ],
   published_date: ISODate("2018-11-30"),
   pages: 300,
   language: "English",
   publisher: {
              name: "Apress",
              founded: 1999,
              location: "US"
            }
}
```

```
{
   title: "MongoDB Recipes",
   author: [ "Subhashini Chellappan"],
   published_date: ISODate("2018-11-30"),
   pages: 120,
   language: "English",
   publisher: {
              name: "Apress",
              founded: 1999,
              location: "US"
            }
}
```

Here, the publisher document is embedded inside the book document, which leads to repetition of the publisher data model.

In this scenario, we can document references to avoid repetition of data. In document references, the growth of relationships determines where to store the references. If the number of books per publisher is small, then we can store the book reference inside the publisher document as shown here.

```
{
   name: "Apress",
   founded: 1999,
   location: "US",
   books: [123456, 456789, ...]
}

{
    _id: 123456,
    title: "Practical Apache Spark",
    author: [ "Subhashini Chellappan", "Dharanitharan Ganesan"
],
```

```
    published_date: ISODate("2018-11-30"),
    pages: 300,
    language: "English"
}
{
   _id: 456789,
   title: "MongoDB Recipes",
   author: [ "Subhashini Chellappan"],
   published_date: ISODate("2018-11-30"),
   pages: 120,
   language: "English"
}
```

If the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays. We can avoid this situation by storing the publisher reference inside the book document as shown here.

```
{
   _id:"Apress",
   name: "Apress",
   founded: 1999,
   location: "US"
}
{
   _id: 123456,
   title: "Practical Apache Spark",
   author: [ "Subhashini Chellappan", "Dharanitharan Ganesan" ],
   published_date: ISODate("2018-11-30"),
   pages: 300,
```

```
   language: "English",
   publisher_id: "Apress"
}

{
   _id: 456789,
   title: "MongoDB Recipes",
   author: [ "Subhashini Chellappan"],
   published_date: ISODate("2018-11-30"),
   pages: 120,
   language: "English",
   publisher_id: "Apress"
}
```

Step 2: Query Document References

Let's discuss how to query document references. Consider the following collections.

```
db.publisher.insert({_id:"Apress",name: "Apress", founded:
1999,location:"US"})
```

```
db.authors.insertMany([{_id: 123456,title: "Practical Apache
Spark",author:["Subhashini Chellappan", "Dharanitharan Ganesan"
], published_date: ISODate("2018-11-30"),pages: 300,language:
"English",publisher_id: "Apress"},{_id: 456789,title: "MongoDB
Recipes",  author: [ "Subhashini Chellappan"],published_
date: ISODate("2018-11-30"),  pages: 120,language:
"English",publisher_id: "Apress"}])
```

To perform a left outer join, use `$lookup` as shown here.

```
db.publisher.aggregate([{$lookup:{from:"authors",localField:"_id",
foreignField:"publisher_id",as:"authors_docs"}}])
```

Here is the output,

```
> db.publisher.aggregate([{$lookup:{from:"authors",localFie
ld:"_id",
... foreignField:"publisher_id",as:"authors_docs"}}])
{ "_id" : "Apress", "name" : "Apress", "founded" : 1999,
"location" : "US", "authors_docs" : [ { "_id" : 123456,
"title" : "Practical Apache Spark", "author" : [ "Subhashini
Chellappan", "Dharanitharan Ganesan" ], "published_date" :
ISODate("2018-11-30T00:00:00Z"), "pages" : 300, "language"
: "English", "publisher_id" : "Apress" }, { "_id" : 456789,
"title" : "MongoDB Recipes", "author" : [ "Subhashini
Chellappan" ], "published_date" : ISODate("2018-11-
30T00:00:00Z"), "pages" : 120, "language" : "English",
"publisher_id" : "Apress" } ] }
>
```

Modeling Tree Structures

Let's look at a data model that describes a tree-like structure.

Recipe 3-3. Model Tree Structure with Parent References

In this recipe, we are going to discuss a tree structure data model using parent references.

Problem

You want to create a data model for a tree structure with parent references.

Solution

Use the parent references pattern.

How It Works

Let's follow the steps in this section to design a data model for a tree structure with parent references.

Step 1: Tree Structure with Parent References

The *parent references* pattern stores each tree node in a document; in addition to the tree node, the document stores the `_id` of the node's parent.

Consider the following `author` tree model with parent references.

```
db.author.insert( { _id: "Practical Apache Spark", parent:
"Books" } )
db.author.insert( { _id: "MongoDB Recipes", parent: "Books" } )
db.author.insert( { _id: "Books", parent: "Subhashini" } )
db.author.insert( { _id: "A Framework For Extracting
Information From Web Using VTD-XML ' s XPath", parent:
"Article" } )
db.author.insert( { _id: "Article", parent: "Subhashini" } )
db.author.insert( { _id: "Subhashini", parent: null } )
```

The tree structure of this `author` collection is shown in Figure 3-3.

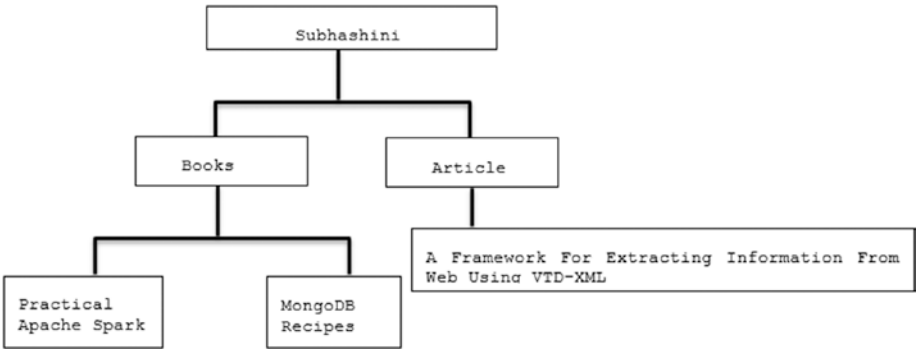


Figure 3-3. *Tree structure for the author collection*

The following command retrieves the parent of a node MongoDB Recipes.

```
db.author.findOne( { _id: "MongoDB Recipes" } ).parent
```

Here is the output,

```
> db.author.findOne( { _id: "MongoDB Recipes" } ).parent
Books
>
```

The next command retrieves the immediate children of the parent.

```
db.author.find( { parent: "Subhashini" } )
```

Here is the output,

```
> db.author.find( { parent: "Subhashini" } )
{ "_id" : "Books", "parent" : "Subhashini" }
{ "_id" : "Article", "parent" : "Subhashini" }
>
```

Step 2: Tree Structure with Child References

The *child references* pattern stores each tree node in a document; in addition to the tree node, the document stores in an array the _id value(s) of the node's children.

Consider the following author tree model with child references.

```
db.author.insert( { _id: "Practical Apache Spark", children: [] } )
db.author.insert( { _id: "MongoDB", children: [] } )
db.author.insert( { _id: "Books", children: [ "Practical Apache
Spark", "MongoDB Recipes" ] } )
db.author.insert( { _id: " A Framework For Extracting
Information From Web Using VTD-XML ' s XPath ", children: [] } )
db.author.insert( { _id: "Article", children: [ " A Framework For
Extracting Information From Web Using VTD-XML ' s XPath " ] } )
db.categories.insert( { _id: "Subhashini", children: [
"Books","Article" ] } )
```

The following command retrieves the immediate children of node Books.

```
db.author.findOne( { _id: "Books" } ).children
```

Here is the output,

```
> db.author.findOne( { _id: "Books" } ).children
[ "Practical Apache Spark", "MongoDB Recipes" ]
>
```

The next command retrieves the MongoDB Recipes parent node and its siblings.

```
db.author.find( { children: "MongoDB Recipes" } )
```

Here is the output,

```
> db.author.find( { children: "MongoDB Recipes" } )
{ "_id" : "Books", "children" : [ "Practical Apache Spark",
"MongoDB Recipes" ] }
>
```

Child references are a good choice to work with tree storage when there are no subtree operations.

Step 3: Tree Structure with an Array of Ancestors

The *array of ancestors* pattern stores each tree node in a document; in addition to the tree node, the document stores in an array the `_id` value(s) of the node's ancestors or path.

Consider this `author` tree model with an array of ancestors.

```
db.author.insert( { _id: "Practical Apache Spark", ancestors: [
"Subhashini", "Books" ], parent: "Books" } )
db.author.insert( { _id: "MongoDB Recipes", ancestors: [
"Subhashini", "Books" ], parent: "Books" } )
db.author.insert( { _id: "Books", ancestors: [ "Subhashini" ],
parent: "Subhashini" } )
db.author.insert( { _id: " A Framework For Extracting
Information From Web Using VTD-XML ", ancestors: [
"Subhashini", "Article" ], parent: "Article" } )
db.author.insert( { _id: "Article", ancestors: [ "Subhashini"
], parent: "Subhashini" } )
db.author.insert( { _id: "Subhashini", ancestors: [ ], parent:
null } )
```

The array of ancestors field stores the `ancestors` field and reference to the immediate parent.

To retrieve the ancestors, use this command.

```
db.author.findOne( { _id: "MongoDB Recipes" } ).ancestors
```

Here is the output,

```
> db.author.findOne( { _id: "MongoDB Recipes" } ).ancestors
[ "Subhashini", "Books" ]
>
```

Use this command to find all its descendants.

```
db.author.find( { ancestors: "Subhashini" } )
```

Here is the output,

```
> db.author.find( { ancestors: "Subhashini" } )
{ "_id" : "Practical Apache Spark", "ancestors" : [
"Subhashini", "Books" ], "parent" : "Books" }
{ "_id" : "MongoDB Recipes", "ancestors" : [ "Subhashini",
"Books" ], "parent" : "Books" }
{ "_id" : "Books", "ancestors" : [ "Subhashini" ], "parent" :
"Subhashini" }
{ "_id" : " A Framework For Extracting Information From Web
Using VTD-XML ", "ancestors" : [ "Subhashini", "Article" ],
"parent" : "Article" }
{ "_id" : "Article", "ancestors" : [ "Subhashini" ], "parent" :
"Subhashini" }
>
```

This pattern provides an efficient solution to find all descendants and the ancestors of a node. The array of ancestors pattern is a good choice for working with subtrees.

Aggregation

Aggregation operations group values from multiple documents and can perform variety of operations on the grouped values to return a single result. MongoDB provides following aggregation operations:

- Aggregation pipeline.
- Map-reduce function.
- Single-purpose aggregation methods.

Aggregation Pipeline

The aggregation pipeline is a framework for data aggregation. It is modeled based on the concept of data processing pipelines. Pipelines execute an operation on some input and use that output as an input to the next operation. Documents enter a multistage pipeline that transforms them into an aggregated result.

Recipe 3-4. Aggregation Pipeline

In this recipe, we are going to discuss how the aggregation pipeline works.

Problem

You want to work with aggregation functions.

Solution

Use this method.

```
db.collection.aggregate()
```

How It Works

Let's follow the steps in this section to work with the aggregation pipeline.

Step 1: Aggregation Pipeline

Execute the following `orders` collection to perform aggregation.

```
db.orders.insertMany([{custID:"10001",amount:500,status:"A"},{c
ustID:"10001",amount:250,status:"A"},{custID:"10002",amount:200
,status:"A"},{custID:"10001",amount: 300, status:"D"}]);
```

To project only customer IDs, use this syntax.

```
db.orders.aggregate( [ { $project : { custID : 1 , _id : 0 } } ] )
```

Here is the output,

```
> db.orders.aggregate( [ { $project : { custID : 1 , _id : 0 }
} ] )
{ "custID" : "10001" }
{ "custID" : "10001" }
{ "custID" : "10002" }
{ "custID" : "10001" }
>
```

To group on `custID` and compute the sum of `amount` use the following command.

```
db.orders.aggregate({$group:{_id:"$custID",TotalAmount:{$sum:"$
amount"}}});
```

In the preceding example, the value of the variable is accessed by using the $ sign.

Here is the output,

```
> db.orders.aggregate({$group:{_id:"$custID",TotalAmount:{$sum:
"$amount"}}});
{ "_id" : "10002", "TotalAmount" : 200 }
{ "_id" : "10001", "TotalAmount" : 1050 }
>
```

To filter on `status: "A"` and then group it on `"custID"` and compute the sum of `amount`, use the following command.

```
db.orders.aggregate({$match:{status:"A"}},{$group:
{_id:"$custID",TotalAmount:{ $sum:"$amount"}}});
```

Here is the ouput:

```
>db.orders.aggregate({$match:{status:"A"}},{$group:
{_id:"$custID",TotalAmount:{ $sum:"$amount"}}});
{ "_id" : "10002", "TotalAmount" : 200 }
{ "_id" : "10001", "TotalAmount" : 750 }
>
```

To group on `"custID"` and compute the average of the `amount` for each group, use this command.

```
db.orders.aggregate({$group:{_id:"$custID",AverageAmount:
{$avg:"$amount"}}});
```

Here is the output,

```
> db.orders.aggregate({$group:{_id:"$custID",AverageAmount:
{$avg:"$amount"}}});
{ "_id" : "10002", "AverageAmount" : 200 }
{ "_id" : "10001", "AverageAmount" : 350 }
>
```

Map-Reduce

MongoDB also provides map-reduce to perform aggregation operations. There are two phases in map-reduce: a map stage that processes each document and outputs one or more objects and a reduce stage that combines the output of the map operation.

A custom JavaScript function is used to perform map and reduce operations. Map-reduce is less efficient and more complex compared to the aggregation pipeline.

Recipe 3-5. Map-Reduce

In this recipe, we are going to discuss how to perform aggregation operations using map-reduce.

Problem

You want to work with aggregation operations using map-reduce.

Solution

Use a customized JavaScript function.

How It Works

Let's follow the steps in this section to work with map-reduce.

Step 1: Map-Reduce

Execute the following orders collection to perform aggregation operations.

```
db.orders.insertMany([{custID:"10001",amount:500,status:"A"},{c
ustID:"10001",amount:250,status:"A"},{custID:"10002",amount:200
,status:"A"},{custID:"10001",amount: 300, status:"D"}]);
```

To filter on status:"A" and then group it on custID and compute the sum of amount, use the following map-reduce function.

Map function:

```
var map = function(){
emit (this.custID, this.amount);}
```

Reduce function:

```
var reduce = function(key, values){ return Array.sum(values) ; }
```

To execute the query:

```
db.orders.mapReduce(map, reduce,{out: "order_
totals",query:{status:"A"}});

db.order_totals.find()
```

Here is the output,

```
> var map = function(){
... emit (this.custID, this.amount);}
> var reduce = function(key, values){ return Array.sum(values)
; }
> db.orders.mapReduce(map, reduce,{out: "order_
totals",query:{status:"A"}});
{
        "result" : "order_totals",
        "timeMillis" : 82,
```

```
        "counts" : {
                "input" : 3,
                "emit" : 3,
                "reduce" : 1,
                "output" : 2
        },
        "ok" : 1
}
> db.order_totals.find()
{ "_id" : "10001", "value" : 750 }
{ "_id" : "10002", "value" : 200 }
>
```

Single-Purpose Aggregation Operations

MongoDB also provides single-purpose aggregation operations such as `db.collection.count()` and `db.collection.distinct()`. These aggregate operations aggregate documents from a single collection. This functionality provides simple access to common aggregation processes.

Recipe 3-6. Single-Purpose Aggregation Operations

In this recipe, we are going to discuss how to use single-purpose aggregation operations.

Problem

You want to work with single-purpose aggregation operations.

Solution

Use these commands.

db.collection.count()
db.collection.distinct()

How It Works

Let's follow the steps in this section to work with single-purpose aggregation operations.

Step 1: Single-Purpose Aggregation Operations

Execute the following `orders` collection to perform single-purpose aggregation operations.

```
db.orders.insertMany([{custID:"10001",amount:500,status:"A"},{c
ustID:"10001",amount:250,status:"A"},{custID:"10002",amount:200
,status:"A"},{custID:"10001",amount: 300, status:"D"}]);
```

Use the following syntax to find a distinct `"custID"`.

```
db.orders.distinct("custID")
```

Here is the output,

```
> db.orders.distinct("custID")
[ "10001", "10002" ]
>
```

To count the number of documents, use this code.

```
db.orders.count()
```

Here is the output,

```
> db.orders.count()
4
>
```

SQL Aggregation Terms and Corresponding MongoDB Aggregation Operators

Table 3-1 shows SQL aggregation terms and their corresponding MongoDB aggregation operators.

Table 3-1. *SQL Aggregation Terms and Corresponding MongoDB Operators*

SQL Term	MongoDB Operator
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM	$sum
COUNT	$sum
JOIN	$lookup

Recipe 3-7. Matching SQL Aggregation to MongoDB Aggregation Operations

In this recipe, we are going to discuss examples of matching MongoDB operations to equivalent SQL aggregation terms.

Problem

You want to understand the equivalent of MongoDB queries for any SQL queries.

Solution

Refer to Table 3-1 and use the equivalent MongoDB operator for a respective SQL clause.

How It Works

Let's follow the steps in this section to understand the MongoDB queries for certain SQL operations.

Step 1: Converting SQL Aggregation Operations to MongoDB

Execute the following query to check the details of the `orders` collection.

```
> db.orders.find()
```

Here is the output,

```
> db.orders.find()
{ "_id" : ObjectId("5d636112eea2dccfdeafa522"), "custID" :
"10001", "amount" : 500, "status" : "A" }
{ "_id" : ObjectId("5d636112eea2dccfdeafa523"), "custID" :
"10001", "amount" : 250, "status" : "A" }
{ "_id" : ObjectId("5d636112eea2dccfdeafa524"), "custID" :
"10002", "amount" : 200, "status" : "A" }
{ "_id" : ObjectId("5d636112eea2dccfdeafa525"), "custID" :
"10001", "amount" : 300, "status" : "D" }
```

Now, let's find the count of records from the `orders` colletion.

Imagine this is the same as an `orders` table in any RDBMS and the SQL to get the count of records from the table is as follows:

```
SELECT COUNT(*) AS count FROM orders
```

Use the following query to get the count of documents from the collection in MongoDB.

```
> db.orders.aggregate( [ { $group: {  _id: null,  count: {
$sum: 1 } } } ] )
```

Here is the output,

```
> db.orders.aggregate( [ { $group: {  _id: null,  count: {
$sum: 1 } } } ] )
{ "_id" : null, "count" : 4 }
```

CHAPTER 4

Indexes

In Chapter 3, we discussed data modeling patterns and the aggregation framework. In this chapter, we to discuss the following topics:

- Indexes.
- Types of indexes.
- Index properties.
- The various indexing strategies to be considered.

Indexes are used to improve the performance of the query. Without indexes, MongoDB must search the entire collection to select those documents that match the query statement. MongoDB therefore uses indexes to limit the number of documents that it must scan.

Indexes are special data structures that store a small portion of the collection's data set in an easy-to-transform format. The index stores a set of fields ordered by the value of the field. This ordering helps to improve the performance of equality matches and range-based query operations.

MongoDB defines indexes at the collection level and indexes can be created on any field of the document. MongoDB creates an index for the `_id` field by default.

Note MongoDB creates a default unique index `_id` field, which helps to prevent inserting two documents with the same value of the `_id` field.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_4

Recipe 4-1. Working with Indexes

In this recipe, we are going to discuss how to work with indexes in MongoDB.

Problem

You want to create an index.

Solution

Use the following command.

```
db.collection.createIndex()
```

How It Works

Let's follow the steps in this section to create an index.

Consider the following `employee` collection.

```
db.employee.insert({empId:1,empName:"John",state:"KA",country:"
India"})
db.employee.insert({empId:2,empName:"Smith",state:"CA",country:
"US"})
db.employee.insert({empId:3,empName:"James",state:"FL",country:
"US"})
db.employee.insert({empId:4,empName:"Josh",state:"TN",country:"
India"})
db.employee.insert({empId:5,empName:"Joshi",state:"HYD",country
:"India"})
```

Step 1: Create an Index

To create single-field index on the empId field, use the following command.

```
db.employee.createIndex({empId:1})
```

Here, the parameter value "1" indicates that empId field values will be stored in ascending order.
Here is the output,

```
> db.employee.createIndex({empId:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

To create an index on multiple fields, known as a compound index, use this command.

```
db.employee.createIndex({empId:1,empName:1})
```

Here is the output,

```
> db.employee.createIndex({empId:1,empName:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
>
```

To display a list of indexes, this is the syntax.

```
db.employee.getIndexes()
```

Here is the output,

```
> db.employee.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.employee"
        },
        {
                "v" : 2,
                "key" : {
                        "empId" : 1
                },
                "name" : "empId_1",
                "ns" : "test.employee"
        },
        {
                "v" : 2,
                "key" : {
                        "empId" : 1,
                        "empName" : 1
                },
```

```
                "name" : "empId_1_empName_1",
                "ns" : "test.employee"
        }
]
>
```

To drop a compound index, use the following command.

```
db.employee.dropIndex({empId:1,empName:1})
```

Here is the output,

```
> db.employee.dropIndex({empId:1,empName:1})
{ "nIndexesWas" : 3, "ok" : 1 }
>
```

To drop all the indexes, use this command.

```
db.employee.dropIndexes()
```

Here is the output,

```
> db.employee.dropIndexes()
{
        "nIndexesWas" : 2,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
>
```

Note We can't drop `_id` indexes. MongoDB creates an index for the `_id` field by default.

Recipe 4-2. Index Types

In this recipe, we are going to discuss various types of indexes.

Problem

You want to create different types of indexes.

Solution

Use the following command.

```
db.collection.createIndex()
```

How It Works

Let's follow the steps in this section to create the different types of indexes shown in Figure 4-1.

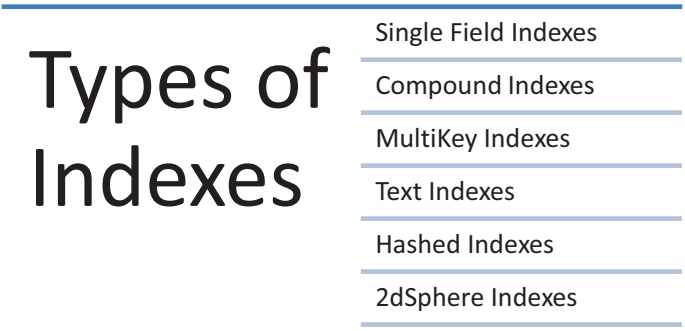


Figure 4-1. *Types of indexes*

Step 1: Multikey Index

Multikey indexes are useful to create an index for a field that holds an array value. MongoDB creates an index key for each element in the array.

Consider the below collection, you can create if the collection is not available.

```
db.employeeproject.insert({empId:1001,empName:"John",projects:[
"Hadoop","MongoDB"]})
db.employeeproject.insert({empId:1002,empName:"James",projects:
["MongoDB","Spark"]})
```

To create an index on the `projects` field, use the following command.

```
db.employeeproject.createIndex({projects:1})
```

Here is the output,

```
> db.employeeproject.createIndex({projects:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Note You can't create a compound multikey index.

Step 2: Text Indexes

MongoDB provides text indexes to support text search queries on string content. You can create a text index on a field that takes as its value a string or an array of string elements.

Consider this post collection.

```
db.post.insert({
   "post_text": "Happy Learning",
   "tags": [
      "mongodb",
      "10gen"
   ]
})
```

To create text indexes, use the following command.

```
db.post.createIndex({post_text:"text"})
```

This command creates a text index for the field post_text.
Here is the output,

```
> db.post.createIndex({post_text:"text"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

To perform a search, use the command shown here.

```
db.post.find({$text:{$search:"Happy"}})
```

Here is the output,

```
> db.post.find({$text:{$search:"Happy"}})
{ "_id" : ObjectId("5bb215286d8d957bcedc225e"), "post_text" :
"Happy Learning", "tags" : [ "mongodb", "10gen" ] }
>
```

Step 3: Hashed Indexes

The size of the indexes can be reduced with the help of hashed indexes. Hashed indexes store the hashes of the values of the indexed field. Hashed indexes support sharding using hashed shard keys. In hashed-based sharding, a hashed index of a field is used as the shard key to partition data across the sharded cluster. We discuss sharding in Chapter 5.

Hashed indexes do not support multikey indexes.

Consider the `user` collection shown here.

```
db.user.insert({userId:1,userName:"John"})
db.user.insert({userId:2,userName:"James"})
db.user.insert({userId:3,userName:"Jack"})
```

To create a hashed-based index on the field `userId`, use the following command.

```
db.user.createIndex( { userId: "hashed" } )
```

Here is the output,

```
> db.user.createIndex( { userId: "hashed" } )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Step 4: 2dsphere Index

The 2dsphere index is useful to return queries on geospatial data.

Consider the schools collection given here.

```
db.schools.insert( {
   name: "St.John's School",
   location: { type: "Point", coordinates: [ -73.97, 40.77 ] },
   } );
db.schools.insert( {
   name: "St.Joseph's School",
   location: { type: "Point", coordinates: [ -73.9928, 40.7193 ] },
   } );
db.schools.insert( {
   name: "St.Thomas School",
   location: { type: "Point", coordinates: [ -73.9375, 40.8303 ] },
   } );
```

Use the following syntax to create a 2dsphere index.

```
db.schools.createIndex( { location : "2dsphere" } )
```

The following code uses the $near operator to return documents that are at least 500 meters from and at most 1,500 meters from the specified GeoJSON point.

```
db.schools.find({location:{$near:{$geometry: { type:
"Point",  coordinates: [ -73.9667, 40.78 ] },$minDistance:
500,$maxDistance: 1500}}})
```

Here is the output,

```
> db.schools.find({location:{$near:{$geometry: { type:
"Point",  coordinates: [ -73.9667, 40.78 ] },$minDistance:
500,$maxDistance: 1500}}})
{ "_id" : ObjectId("5ca47a184b034d4cc1345f45"), "name" : "St.
John's School", "location" : { "type" : "Point", "coordinates"
: [ -73.97, 40.77 ] } }
>
```

Recipe 4-3. Index Properties

Indexes can also have properties. The index properties define certain characteristics and behaviors of an indexed field at runtime. For example, a unique index ensures the indexed fields do not support duplicates. In this recipe, we are going to discuss various index properties.

Problem

You want to work with index properties.

Solution

Use this command.

```
db.collection.createIndex()
```

How It Works

Let's follow the steps in this section to work with index properties.

Step 1: TTL Indexes

Time to Live (TTL) indexes are single-field indexes that are used to remove documents from a collection after a certain amount of time. Data expiration is useful for certain types of information such as logs, machine-generated data, and so on.

Consider the `sample` collection shown here.

```
db.credit.insert({credit:16})
db.credit.insert({credit:18})
db.credit.insert({credit:12})
```

To create a TTL index on the field credit, issue the following command.

```
db.credit.createIndex( { credit: 1 }, { expireAfterSeconds: 35 } );
```

Here is the output,

```
> db.credit.createIndex( { credit: 1 }, { expireAfterSeconds:35
} );
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Step 2: Unique Indexes

A unique index ensures that the indexed fields do not contain any duplicate values. By default, MongoDB creates a unique index on the `_id` field.

Consider the following student collection.

```
db.student.insert({_id:1,studid:101,studname:"John"})
db.student.insert({_id:2,studid:102,studname:"Jack"})
db.student.insert({_id:3,studid:103,studname:"James"})
```

To create unique index for the field studId, use this command.

```
db.student.createIndex({"studid":1}, {unique:true})
```

Here is the output,

```
> db.student.createIndex({"studid":1}, {unique:true})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

When we try to insert the value studid:101,

```
db.student.insert([{_id:1,studid:101,studname:"John"}])
```

throws the following error message:

```
"errmsg" : "E11000 duplicate key error collection: test.student
index: _id_ dup key: { : 1.0 }",
```

Step 3: Partial Indexes

Partial indexes are useful when you want to index the documents in a collection that meet a specific filter condition. The filter condition could be specified using any operators. For example, `db.person.find( { age: { $gt: 15 } } )` can be used to find the documents that have an age

greater than 15 in the person collection. Partial indexes reduce storage requirements and performance costs because they store only a subset of the documents.

Use the db.collection.createIndex() method with the partialFilterExpression option to create a partial index.

The partialFilterExpression option accepts a document that specifies the filter condition using:

- equality expressions (i.e., field: value or using the $eq operator).
- $exists: true expression.
- $gt, $gte, $lt, and $lte expressions.
- $type expressions.
- $and operator at the top level only.

Consider the following documents of a person collection.

```
db.person.insert({personName:"John",age:16})
db.person.insert({personName:"James",age:15})
db.person.insert({personName:"John",hobbies:["sports","music"]})
```

To index only those documents in the person collection where the value in the age field is greater than 15, use the following command.

```
db.person.createIndex(  { age: 1},{partialFilterExpression: {
age: { $gt: 15 }}})
```

The query shown here uses a partial index to return those documents in the person collection where the age value is greater than 15.

```
db.person.find( { age: { $gt: 15 } } )
```

Here is the output,

```
> db.person.find( { age: { $gt: 15 } } )
{ "_id" : ObjectId("5ca453954b034d4cc1345f3b"), "personName" :
"John", "age" : 16 }
>
```

Step 4: Sparse Indexes

Sparse indexes store entries only for the documents that have the indexed field, even if it contains `null` values. A sparse index skips any documents that do not have the indexed field. The index is considered sparse because it does not include all the documents of a collection.

Consider the following `person` collection.

```
db.person.insert({personName:"John",age:16})
db.person.insert({personName:"James",age:15})
db.person.insert({personName:"John",hobbies:["sports","mus
ic"]})
```

To create a sparse index on the `age` field, issue this command.

```
db.person.createIndex( { age: 1 }, { sparse: true } );
```

Here is the output,

```
> db.person.createIndex( { age: 1 }, { sparse: true } );
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

To return all documents in the collection named person using the index on the age field, use hint () to specify a sparse index.

```
db.person.find().hint( { age: 1 } ).count();
```

Here is the output,

```
> db.person.find().hint( { age: 1 } ).count();
2
>
```

To perform a correct count, use this code.

```
db.person.find().count();
```

Here is the output,

```
> db.person.find().count();
3
>
```

Note Partial indexes determine the index entries based on the filter condition, whereas sparse indexes select the documents based on the existence of the indexed field.

Recipe 4-4. Indexing Strategies

We must follow different strategies to create the right index for our requirements. In this recipe, we are going to discuss various indexing strategies.

Problem

You want to learn about indexing strategies to ensure you are creating the right type of index for different purposes.

Solution

The best indexing strategy is determined by different factors, including

- Type of executing query.
- Number of read/write operations.
- Available memory.

Figure 4-2 illustrates the different indexing strategies.

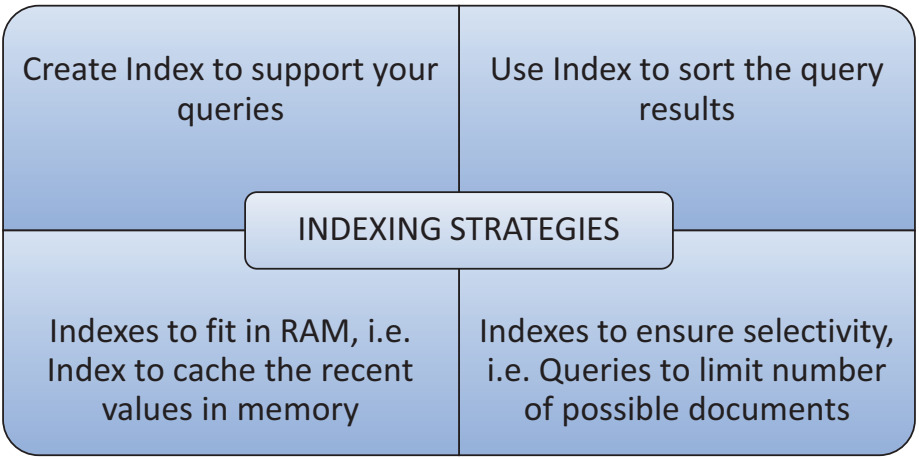


Figure 4-2. *Indexing strategies*

How It Works

Let's follow the steps in this section to work with different indexing strategies.

Step 1: Create an Index to Support Your Queries

Creating the right index to support the queries increases the query execution performance and results in great performance.

Create a single-field index if all the queries use the same single key to retrieve the documents.

```
> db.employee.createIndex({empId:1})
```

Create a multifield compound index if all the queries use more than one key (multiple filter condition) to retrieve the documents.

```
> db.employee.createIndex({empId:1,empName:1})
```

Step 2: Using an Index to Sort the Query Results

Sort operations use indexes for better performance. Indexes determine the sort order by fetching the documents based on the ordering in the index.

Sorting can be done in the following scenarios:

- Sorting with a single-field index.
- Sorting on multiple fields.

Sorting with a Single-Field Index

The index can support ascending or descending order on a single field while retrieving the documents.

```
> db.employee.createIndex({empId:1})
```

The preceding index can support ascending order sorting.

```
> db.employee.find().sort({empId:1})
```

Here is the output,

```
MongoDB Enterprise > db.employee.createIndex({empId:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
MongoDB Enterprise > db.employee.find()
{ "_id" : ObjectId("5d50ed688dcf280c50fde439"), "empId" : 1,
"empName" : "John", "state" : "KA", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43a"), "empId" : 2,
"empName" : "Smith", "state" : "CA", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43b"), "empId" : 3,
"empName" : "James", "state" : "FL", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43d"), "empId" : 5,
"empName" : "Joshi", "state" : "HYD", "country" : "India" }
MongoDB Enterprise > db.employee.find().sort({empId:1})
{ "_id" : ObjectId("5d50ed688dcf280c50fde439"), "empId" : 1,
"empName" : "John", "state" : "KA", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43a"), "empId" : 2,
"empName" : "Smith", "state" : "CA", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43b"), "empId" : 3,
"empName" : "James", "state" : "FL", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43d"), "empId" : 5,
"empName" : "Joshi", "state" : "HYD", "country" : "India" }
```

The same index can also support sorting of documents in descending order.

```
MongoDB Enterprise > db.employee.find().sort({empId:-1})
{ "_id" : ObjectId("5d50ed688dcf280c50fde43d"), "empId" : 5,
"empName" : "Joshi", "state" : "HYD", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43b"), "empId" : 3,
"empName" : "James", "state" : "FL", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43a"), "empId" : 2,
"empName" : "Smith", "state" : "CA", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde439"), "empId" : 1,
"empName" : "John", "state" : "KA", "country" : "India" }
```

Sorting on Multiple Fields

We can create a compund index to support sorting on multiple fields.

```
> db.employee.createIndex({empId:1,empName:1})
```

Here is the output,

```
MongoDB Enterprise > db.employee.createIndex({empId:1,empName:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
MongoDB Enterprise > db.employee.find().
sort({empId:1,empName:1})
{ "_id" : ObjectId("5d50ed688dcf280c50fde439"), "empId" : 1,
"empName" : "John", "state" : "KA", "country" : "India" }
```

```
{ "_id" : ObjectId("5d50ed688dcf280c50fde43a"), "empId" : 2,
"empName" : "Smith", "state" : "CA", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43b"), "empId" : 3,
"empName" : "James", "state" : "FL", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43d"), "empId" : 5,
"empName" : "Joshi", "state" : "HYD", "country" : "India" }
```

Index to Hold Recent Values in Memory

When using multiple collections, we must consider the size of indexes on all collections and ensure the index fits in memory to avoid the system reading the index from the disk.

Use the following query to check the size of the index for any collection.

```
> db.employee.totalIndexSize()
```

When we ensure the index fits entirely into the RAM, that ensures faster system processing.
Here is the output,

```
MongoDB Enterprise > db.employee.totalIndexSize()
49152
```

Create Queries to Ensure Selectivity

The ability of any query to narrow down the results using the created index is called *selectivity*. Writing queries that limit the number of possible documents with the indexed field and the queries that are appropriately selective relative to your indexed data ensures selectivity.

```
> db.employee.find({empId:{$gt:1},country:"India"})
```

This query must scan all the documents to return the result of empId values greater than 1.
Here is the output,

```
MongoDB Enterprise > db.employee.find({empId:{$gt:1},
country:"India"})
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43d"), "empId" : 5,
"empName" : "Joshi", "state" : "HYD", "country" : "India" }
> db.employee.find({empId:4})
```

This query must scan only one document to return the result empId:4.
Here is the output,

```
MongoDB Enterprise > db.employee.find({empId:4})
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
```

CHAPTER 5

Replication and Sharding

In Chapter 4, we discussed various indexes in MongoDB. In this chapter, we cover the following topics:

- Replication.
- Sharding.

Replication

Replication is the process of creating and managing a duplicate version of a database across servers to provide redundancy and increase availability of data.

In MongoDB, replication is achieved with the help of a replica set, a group of `mongod` instances that maintain the same data set. A replica set contains one primary node that is responsible for all write operations and one or more secondary nodes that replicate the primary's oplog and apply the operations to their data sets to reflect the primary's data set. Figure 5-1 is an illustration of a replica set.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_5

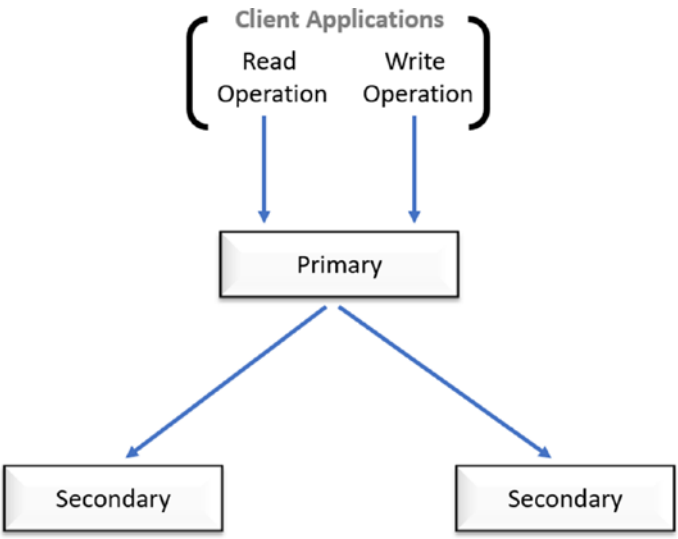


Figure 5-1. *A replica set*

Recipe 5-1. Set Up a Replica Set

In this recipe, we are going to discuss how to set up a replica set (one primary and two secondaries) in Windows.

Problem

You want to create a replica set.

Solution

Use a group of `mongod` instances.

How It Works

Let's follow the steps in this section to set up a three-member replica set.

Step 1: Three-Member Replica Set

First, create three data directories:

```
md c:\mongodb\repset\rs1
md c:\mongodb\repset\rs2
md c:\mongodb\repset\rs3
```

Here is the output,

```
c:\>md c:\mongodb\repset\rs1

c:\>md c:\mongodb\repset\rs2

c:\>md c:\mongodb\repset\rs3
```

Next, start three mongod instances as shown here (see Figures 5-2, 5-4, and 5-6).

```
start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs1
--port 20001 --replSet myrs
```

```
C:\Program Files\MongoDB\Server\4.0\bin>start mongod --bin
d_ip localhost --dbpath c:\mongodb\repset\rs1 --port 20001
 --replSet myrs
```

Figure 5-2. *Starting mongod with replica set 1*

Note hostname must be replaced as with ipaddress or localhost if it is the same local machine.

Refer to Figure 5-3 for a mongod instance that is waiting for connection on port 20001.

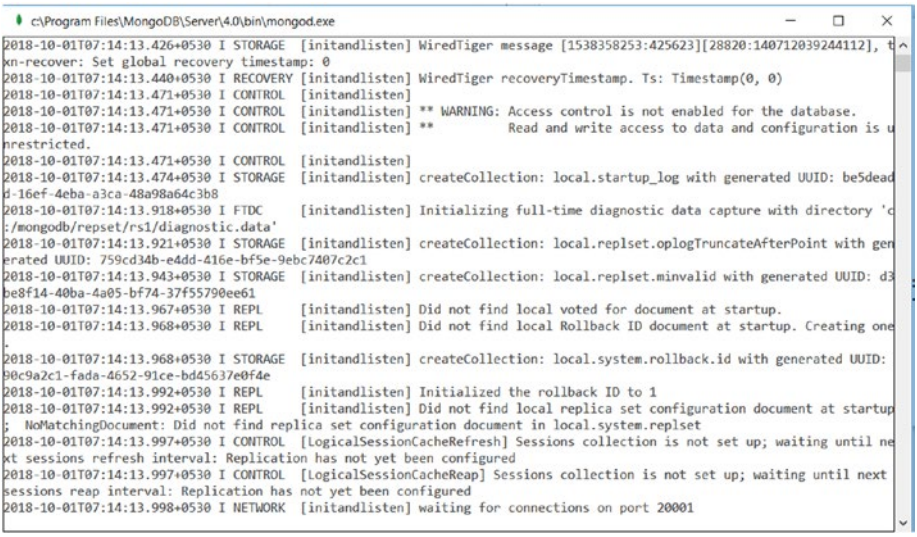


Figure 5-3. *mongod instance waiting for connection on port 20001*

```
start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs2
--port 20002 --replSet myrs
```

```
C:\Program Files\MongoDB\Server\4.0\bin>start mongod --bin
d_ip localhost --dbpath c:\mongodb\repset\rs2 --port 20002
 --replSet myrs
```

Figure 5-4. *Starting mongod with replica set 2*

Refer to Figure 5-5 for a mongod instance that is waiting for connection on port 20002.

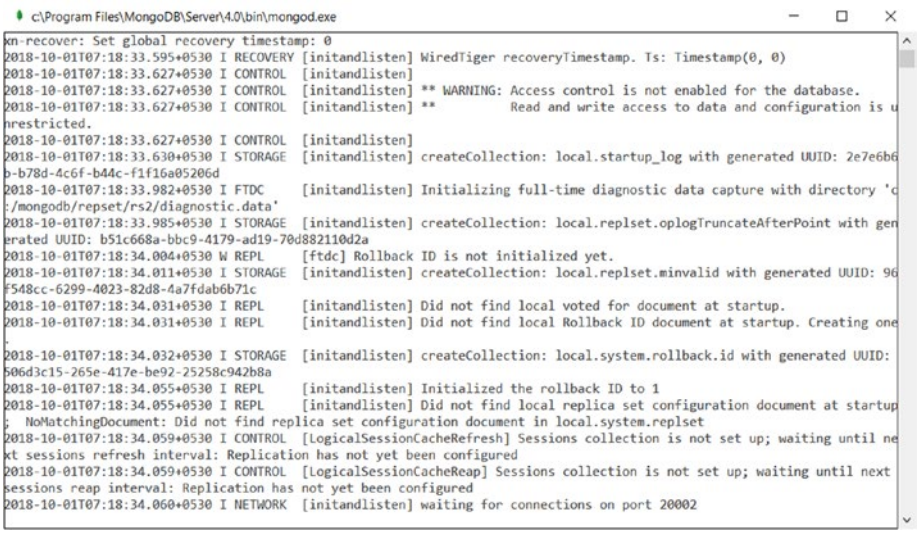


Figure 5-5. *mongod instance waiting for connection on port 20002*

```
start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs3
--port 20003 --replSet myrs
```

```
C:\Program Files\MongoDB\Server\4.0\bin>start mongod --bin
d_ip localhost --dbpath c:\mongodb\repset\rs3 --port 20003
 --replSet myrs
```

Figure 5-6. *Starting mongod with replica set 3*

Refer Figure 5-7 for a mongod instance that is waiting for connection on port 20002.

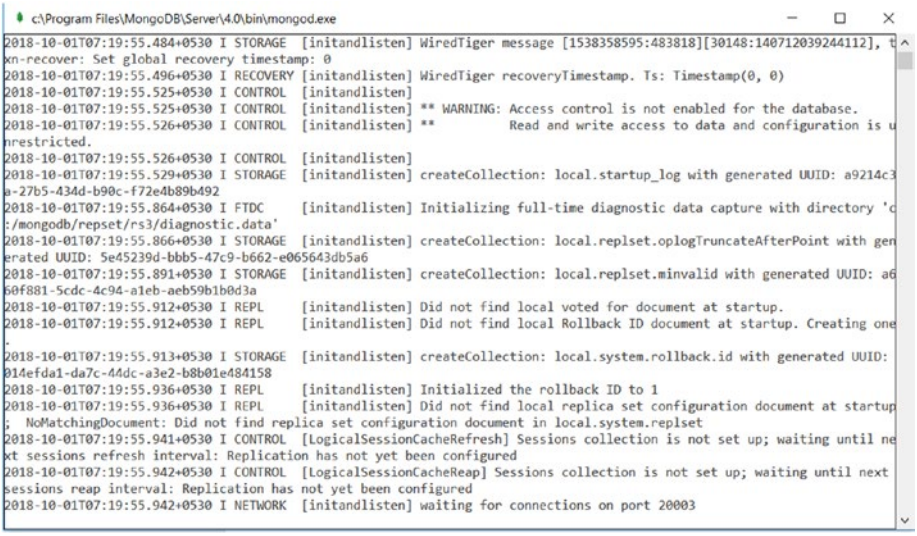


Figure 5-7. *mongod instance waiting for connection on port 20003*

Next, issue the following command to connect to a `mongod` instance running on port 20001.

```
mongo hostname:20001
```

Figure 5-8 shows the `mongo` shell that is running on port 20001.

```
C:\Program Files\MongoDB\Server\4.0\bin>mongo localhost:20001
MongoDB shell version v4.0.11
connecting to: mongodb://localhost:20001/test?gssapiServiceNam
e=mongodb
Implicit session: session { "id" : UUID("58718da2-749c-410e-9e
3b-336ee8ae5a3a") }
MongoDB server version: 4.0.11
Server has startup warnings:
2019-09-23T13:41:53.790+0530 I CONTROL  [initandlisten]
2019-09-23T13:41:53.791+0530 I CONTROL  [initandlisten] ** WAR
NING: Access control is not enabled for the database.
2019-09-23T13:41:53.792+0530 I CONTROL  [initandlisten] **
      Read and write access to data and configuration is unres
tricted.
2019-09-23T13:41:53.792+0530 I CONTROL  [initandlisten]
MongoDB Enterprise >
```

Figure 5-8. *Connect to mongo instance on port 20001*

Issue the following command in the mongo shell to create a three-member replica set.

```
rs.initiate();   // to initiate replica set
```

Here is the output,

```
> rs.initiate();
{
        "info2" : "no configuration specified. Using a default
                  configuration for the set",
        "me" : "hostname:20001",
        "ok" : 1,
        "operationTime" : Timestamp(1538362864, 1),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538362864, 1),
                "signature" : {
```

```
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

After initiating the replica set, we can add the secondary node by using this command.

```
rs.add("hostname:20002");  // to add secondary
```

Here is the output,

```
myrs:SECONDARY> rs.add("hostname:20002");
{
        "ok" : 1,
        "operationTime" : Timestamp(1538362927, 1),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538362927, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
myrs:PRIMARY>
```

Here, the mongod instance running on port 20001 becomes primary.

We can add another secondary node by using the following command.

```
rs.add("hostname:20003");    // to add secondary
```

Here is the output,

```
myrs:PRIMARY>  rs.add("hostname:20003");
{
        "ok" : 1,
        "operationTime" : Timestamp(1538362931, 1),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538362931, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
myrs:PRIMARY>
```

Now, you can check the status of the replica set by issuing the following command.

```
rs.status()
```

Next, create a collection named employee using the primary replica node.

```
 db.employee.insert({_id:10001,name:'Subhashini'});
 db.employee.insert({_id:10002,name:'Shobana'});
```

Here is the output,

```
myrs:PRIMARY> db.employee.insert({_
id:10001,name:'Subhashini'});
WriteResult({ "nInserted" : 1 })
myrs:PRIMARY>  db.employee.insert({_id:10002,name:'Shobana'});
WriteResult({ "nInserted" : 1 })
myrs:PRIMARY>
```

Now, connect to the mongo shell running on port 20002 by issuing this command.

```
mongo hostname:20002
```

Here is the output,

```
myrs:SECONDARY>
```

Now, issue the following command to find all employees as shown here.

```
myrs:SECONDARY> db.employee.find()
Error: error: {
        "operationTime" : Timestamp(1538364766, 1),
        "ok" : 0,
        "errmsg" : "not master and slaveOk=false",
        "code" : 13435,
        "codeName" : "NotMasterNoSlaveOk",
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538364766, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
myrs:SECONDARY>
```

The output `Error: error: { ..... }` shows that we are getting an error message because we are trying to read data from the secondary node.

Issue the following command to perform a read operation from a secondary node.

```
myrs:SECONDARY> rs.slaveOk()

myrs:SECONDARY> db.employee.find()
{ "_id" : 10001, "name" : "Subhashini" }
{ "_id" : 10002, "name" : "Shobana" }
```

Next, try to perform a write operation from a secondary node.

```
myrs:SECONDARY> db.employee.insert({_id:10003,name:"Arunaa MS"})
WriteCommandError({
        "operationTime" : Timestamp(1538364966, 1),
        "ok" : 0,
        "errmsg" : "not master",
        "code" : 10107,
        "codeName" : "NotMaster",
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538364966, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
})
myrs:SECONDARY>
```

In the preceding output, `WriteCommandError { ..... }` shows that we are getting an error message because we can't perform write operations in a secondary node. We can perform write operations only in the primary node.

All the secondary nodes replicate the primary's log and apply their operations to ensure the secondary's data set reflects the primary's data set, as shown in Figure 5-9.

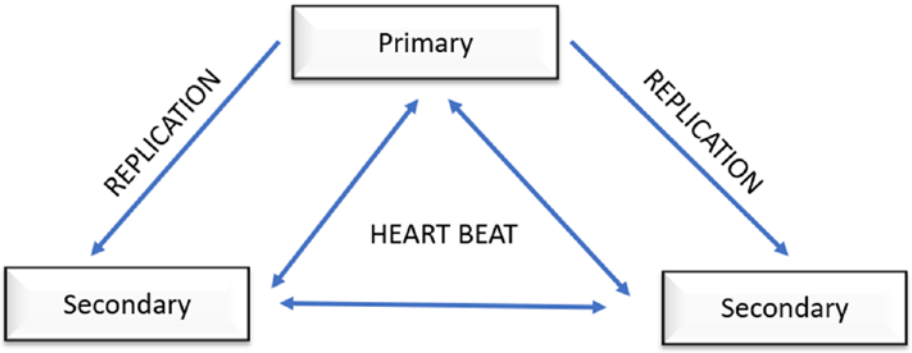


Figure 5-9. *Replication strategy*

Step 2: Auto Failover—High Availability

Kill the primary running on port 20001 and press Enter in the `mongo` shell running on ports 20002 and 20003 (Figure 5-10). Any one of the secondary nodes can become primary now.

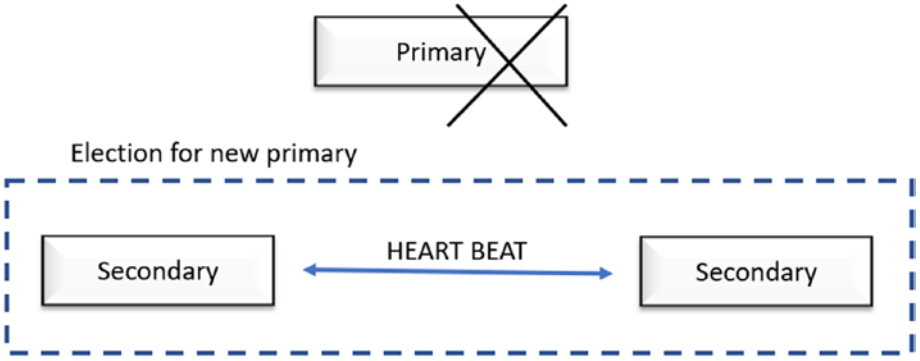


Figure 5-10. *Primary failover*

When a primary does not communicate with other members in the replica set for a configured period (10 seconds by default), an eligible secondary node calls for an election to nominate itself as the new primary and resume its normal operation (Figure 5-11).

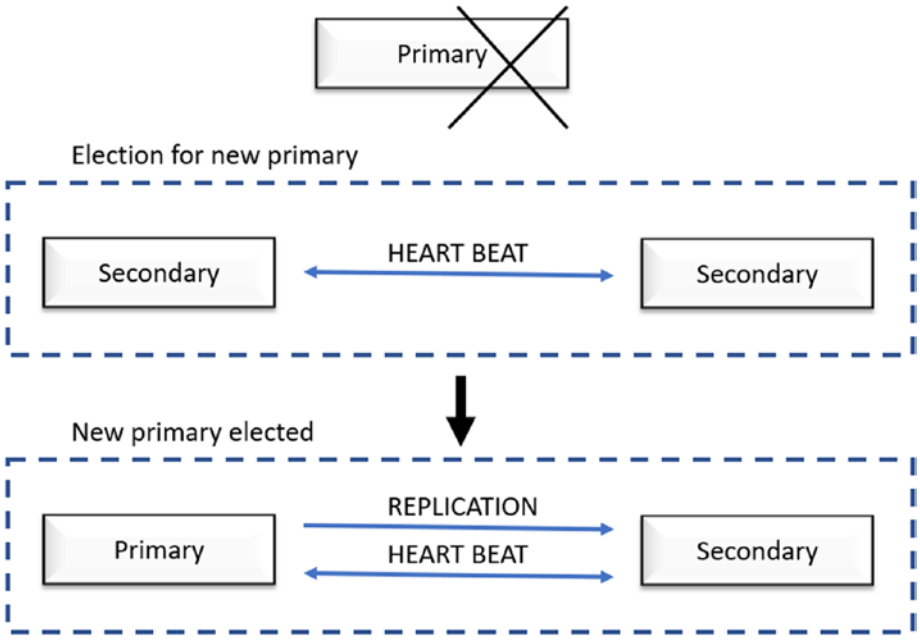


Figure 5-11. *Primary failover and new primary election*

To enable free monitoring, run the following command.

```
db.enableFreeMonitoring()
```

To permanently disable this reminder, run the following command.

```
db.disableFreeMonitoring()
```

To check the status of free monitoring, use this command.

```
db.getFreeMonitoringStatus()
```

Sharding

Sharding is a method for distributing data across multiple machines. There are two methods for addressing system growth: vertical and horizontal scaling.

- *Vertical scaling:* We need to increase the capacity of a single server such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space.
- *Horizontal scaling:* We need to divide the data set and distribute the workload across the servers by adding additional servers to increase the capacity as required.

MongoDB supports horizontal scaling through sharding. A MongoDB sharded cluster consists of the following components:

1. *Shard:* Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.
2. `mongos`*:* The `mongos` acts as a query router, providing an interface between client applications and the sharded cluster.
3. *Config servers:* Config servers store metadata and configuration settings for the cluster.

Recipe 5-2. Sharding

In this recipe, we are going to discuss how to create sharding to distribute data across servers.

Problem

You want to create sharding to distribute data across servers.

Solution

The solution is a group of mongod instances.

How It Works

Let's follow the steps in this section to set up a sharding.

Step 1: Sharded Cluster

First, create data directories for three shards as shown here.

shard1

```
md c:\shard_data\shard1\data1
md c:\shard_data\shard1\data2
md c:\shard_data\shard1\data3
```

shard2

```
md c:\shard_data\shard2\data1
md c:\shard_data\shard2\data2
md c:\shard_data\shard2\data3
```

shard3

```
md c:\shard_data\shard3\data1
md c:\shard_data\shard3\data2
md c:\shard_data\shard3\data3
```

Next, start the shards as shown here.

shard1

```
start mongod.exe --shardsvr --port 26017 --dbpath "c:\shard_
data\shard1\data1" --replSet shard1_replset
start mongod.exe --shardsvr --port 26117 --dbpath "c:\shard_
data\shard1\data2" --replSet shard1_replset
start mongod.exe --shardsvr --port 26217 --dbpath "c:\shard_
data\shard1\data3" --replSet shard1_replset
```

Figure 5-12 shows the execution of commands.

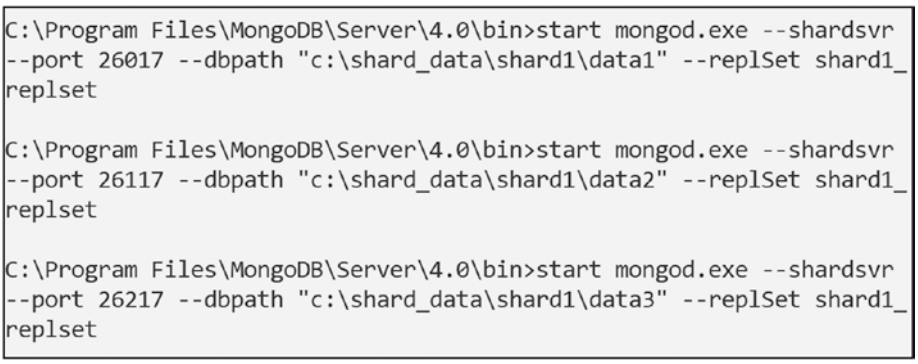


Figure 5-12. *Starting the shard1 server*

shard2

```
start mongod.exe --shardsvr --port 28017 --dbpath "c:\shard_
data\shard2\data1" --replSet shard2_replset
start mongod.exe --shardsvr --port 28117 --dbpath "c:\shard_
data\shard2\data2" --replSet shard2_replset
start mongod.exe --shardsvr --port 28217 --dbpath "c:\shard_
data\shard2\data3" --replSet shard2_replset
```

Figure 5-13 shows the execution of commands.

```
C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --shardsvr
--port 28017 --dbpath "c:\shard_data\shard2\data1" --replSet shard2_
replset

C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --shardsvr
--port 28117 --dbpath "c:\shard_data\shard2\data2" --replSet shard2_
replset

C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --shardsvr
--port 28217 --dbpath "c:\shard_data\shard2\data3" --replSet shard2_
replset
```

Figure 5-13. *Starting the shard2 server*

shard3

```
start mongod.exe --shardsvr --port 29017 --dbpath "c:\shard_
data\shard3\data1" --replSet shard3_replset
start mongod.exe --shardsvr --port 29117 --dbpath "c:\shard_
data\shard3\data2" --replSet shard3_replset
start mongod.exe --shardsvr --port 29217 --dbpath "c:\shard_
data\shard3\data3" --replSet shard3_replset
```

Figure 5-14 shows the execution of commands.

```
C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --shardsvr
--port 29017 --dbpath "c:\shard_data\shard3\data1" --replSet shard3_
replset

C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --shardsvr
--port 29117 --dbpath "c:\shard_data\shard3\data2" --replSet shard3_
replset

C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --shardsvr
--port 29217 --dbpath "c:\shard_data\shard3\data3" --replSet shard3_
replset
```

Figure 5-14. *Starting the shard3 server*

Now, connect to one of the shard servers to enable a replica set as shown here and in Figure 5-15.

```
mongo.exe hostname:26017

C:\Program Files\MongoDB\Server\4.0\bin> mongo.exe
localhost:26017
```

```
C:\Program Files\MongoDB\Server\4.0\bin>mongo.exe localhost:26017
MongoDB shell version v4.0.11
connecting to: mongodb://localhost:26017/test?gssapiServiceName=mong
odb
```

Figure 5-15. *Connect the* `shard1` *server*

Next initiate the replica by using the following command in `mongo` shell as shown here.

```
MongoDB Enterprise > rs.initiate(
{
_id: "shard1_replset",
members: [
{ _id : 0, host:"hostname:26017" },
{ _id : 1, host:"hostname:26117" },
{ _id : 2, host:"hostname:26217" }]}
)
```

Figure 5-16 is the snapshot for reference.

```
MongoDB Enterprise > rs.initiate(
... {
... _id: "shard1_replset",
... members: [
... { _id : 0, host:"hostname:26017" },
... { _id : 1, host:"hostname:26117" },
... { _id : 2, host:"hostname:26217" }]}
... )
```

Figure 5-16. *Initiating the replica on the* `shard1` *server*

Now connect to another shard and initiate the replica.

```
C:\Program Files\MongoDB\Server\4.0\bin> mongo.exe
hostname:28017
MongoDB Enterprise >  rs.initiate(
{
_id: "shard2_replset",
members: [
{ _id : 0, host:"hostname:28017" },
{ _id : 1, host:"hostname:28117" },
{ _id : 2, host:"hostname:28217" }
]
}
)
```

Now connect to the third shard and initiate the replica,.

```
C:\Program Files\MongoDB\Server\4.0\bin> mongo.exe
hostname:29017

MongoDB Enterprise >  rs.initiate(
```

```
{
_id: "shard3_replset",
members: [
{ _id : 0, host:"hostname:29017" },
{ _id : 1, host:"hostname:29117" },
{ _id : 2, host:"hostname:29217" }
]
}
)
```

Now, start the config servers by using the commands that follow. Create data directories for the config servers.

```
md c:\shard_data\config_server1\data1
md c:\shard_data\config_server1\data2
md c:\shard_data\config_server1\data3
```

Start the config server with a replica set.

```
start mongod.exe --configsvr --port 47017 --dbpath "c:\shard_
data\config_server1\data1" --replSet configserver1_replset

start mongod.exe --configsvr --port 47117 --dbpath "c:\shard_
data\config_server1\data2" --replSet configserver1_replset

start mongod.exe --configsvr --port 47217 --dbpath "c:\shard_
data\config_server1\data3" --replSet configserver1_replset
```

Figure 5-17 is a snapshot for reference to start config servers.

```
C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --configsvr
 --port 47017 --dbpath "c:\shard_data\config_server1\data1" --replSe
t configserver1_replset

C:\Program Files\MongoDB\Server\4.0\bin>
C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --configsvr
 --port 47117 --dbpath "c:\shard_data\config_server1\data2" --replSe
t configserver1_replset

C:\Program Files\MongoDB\Server\4.0\bin>
C:\Program Files\MongoDB\Server\4.0\bin>start mongod.exe --configsvr
 --port 47217 --dbpath "c:\shard_data\config_server1\data3" --replSe
t configserver1_replset
```

Figure 5-17. *Starting the config servers*

Now we can connect to config servers to enable the replica set.

```
C:\Program Files\MongoDB\Server\4.0\bin> mongo.exe
hostname:47017
```

Then initiate the replica set.

```
MongoDB Enterprise >  rs.initiate(
{
_id: "configserver1_replset",
configsvr: true,
members: [
{ _id : 0, host : "hostname:47017" },
{ _id : 1, host : "hostname:47117" },
{ _id : 2, host : "hostname:47217" }
]
}
)
```

Start mongos as shown here.

```
start mongos.exe --configdb configserver1_replset/hostname:
47017,hostname:47117,hostname:47217 --port 1000
```

Now it is time to perform sharding. Here, we are going to use one query router, one config server, and three shards.

Connect to the query router as shown here.

```
mongo.exe localhost:1000
```

Here is the output,

```
2018-10-01T12:47:08.929+0530 I CONTROL  [main]
mongos>
```

Next, add the three shard servers to the config server as shown here.

```
sh.addShard("shard1_replset/localhost:26017,localhost:26117,loc
alhost:26217")
```

Here is the output for the first shard:

```
mongos> sh.addShard("shard1_replset/localhost:26017,localhost:2
6117,localhost:26217")
{
        "shardAdded" : "shard1_replset",
        "ok" : 1,
        "operationTime" : Timestamp(1538378773, 7),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538378773, 7),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                AAAAAAAA="),
```

```
                        "keyId" : NumberLong(0)
                }
        }
}
mongos>

sh.addShard("shard2_replset/localhost:28017,localhost:28117,loc
alhost:28217")
```

Here is the output for the second shard.

```
mongos> sh.addShard("shard2_replset/localhost:28017,localhost:2
8117,localhost:28217")
{
        "shardAdded" : "shard2_replset",
        "ok" : 1,
        "operationTime" : Timestamp(1538378886, 4),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538378886, 6),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
mongos>

sh.addShard("shard3_replset/localhost:29017,localhost:29117,
localhost:29217")
```

Here is the output for the third shard.

```
mongos> sh.addShard("shard3_replset/localhost:29017,localhost:
29117,localhost:29217")
{
        "shardAdded" : "shard3_replset",
        "ok" : 1,
        "operationTime" : Timestamp(1538378938, 4),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538378938, 4),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

Issue the following command to check the status of the sharding.

```
mongos> sh.status()
```

Now, to enable sharding for a database, use this command.

```
mongos> sh.enableSharding("demos")
```

To enable sharding for a collection, use this code.

```
 mongos> sh.shardCollection("demos.users",{"id":1}
```

Here is the output,

```
mongos> sh.shardCollection("demos.users",{"id":1})
{
        "collectionsharded" : "demos.users",
        "collectionUUID" : UUID("0122f602-212e-4c79-8be7-
                          5c7a63676c8b"),
        "ok" : 1,
        "operationTime" : Timestamp(1538379345, 15),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1538379345, 15),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

Next we can create a huge collection using the following commands.

```
mongos> use demos;

for(var i=0;i<10000;i++){db.users.insert({id: Math.random(),
count:i, date: new Date()})}
```

Issue the next command to count the number of users.

```
mongos> db.users.count()
10000
```

Issue the following command to see the distribution of the user collection.

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5bb1c63e9bbb23ade7b2dd95")
  }
  shards:
        {  "_id" : "shard1_replset",  "host" : "shard1_replset/
        localhost:26017,localhost:26117,localhost:26217",
        "state" : 1 }
        {  "_id" : "shard2_replset",  "host" : "shard2_replset/
        localhost:28017,localhost:28117,localhost:28217",
        "state" : 1 }
        {  "_id" : "shard3_replset",  "host" : "shard3_replset/
        localhost:29017,localhost:29117,localhost:29217",
        "state" : 1 }
  active mongoses:
        "4.0.2" : 1
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
```

```
databases:
      {  "_id" : "config",  "primary" :
      "config",  "partitioned" : true }
              config.system.sessions
                      shard key: { "_id" : 1 }
                      unique: false
                      balancing: true
                      chunks:
                              shard1_replset  1
                      { "_id" : { "$minKey" : 1 } } -->>
                      { "_id" : { "$maxKey" : 1 } } on :
                      shard1_replset Timestamp(1, 0)
      {  "_id" : "demos",  "primary" : "shard2_replset",
      "partitioned" : true,  "version" : {  "uuid" : UUID
      ("52ced1e7-6afd-4554-a310-609d62c11450"),  "lastMod" :
      1 } }
              demos.users
                      shard key: { "id" : 1 }
                      unique: false
                      balancing: true
                      chunks:
                              shard2_replset  1
                      { "id" : { "$minKey" : 1 } } -->>
                      { "id" : { "$maxKey" : 1 } } on :
                      shard2_replset Timestamp(1, 0)
```

This shows that `user` data is distributed to `shard1` and `shard2`.

Note All these demos are executed in a Windows environment.

Next, to shard a collection using hashed sharding, enable sharding for a database as shown here.

```
sh.enableSharding("sample")
```

Here is the output,

```
mongos> sh.enableSharding("sample")
{
        "ok" : 1,
        "operationTime" : Timestamp(1554705786, 5),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1554705786, 5),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

Finally, use the following command to shard a collection using hashed sharding.

```
sh.shardCollection("sample.users",{"id":"hashed"})
```

Here is the output,

```
mongos> sh.shardCollection("sample.users",{"id":"hashed"})
{
        "collectionsharded" : "sample.users",
        "collectionUUID" : UUID("4fb2750a-ec5d-4ae0-be8f-
                           a8fbea8294ab"),
        "ok" : 1,
        "operationTime" : Timestamp(1554705891, 13),
```

```
        "$clusterTime" : {
                "clusterTime" : Timestamp(1554705891, 25),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

CHAPTER 6

Multidocument Transactions

In Chapter 5, we discussed replica sets and sharding in MongoDB. In this chapter, we are going to discuss multidocument transactions in MongoDB, a new feature introduced in MongoDB 4.0.

Multidocument Transactions in MongoDB

In MongoDB, a write operation on single document is atomic, even if the write operation modifies multiple embedded documents within a single document. When a single write operation (e.g., `db.collection.updateMany()`) modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not.

Starting with version 4.0, MongoDB provides multidocument transactions for replica sets. Multidocument transactions help us to achieve all-or-nothing execution to maintain data integrity.

The multidocument transactions are atomic.

- When a transaction commits, all data changes made in the transaction are saved and visible outside the transaction. The data changes are not visible outside the transaction until the transaction is committed.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_6

- When a transaction aborts, all data changes made in the transaction are discarded without ever becoming visible.

Note Multidocument transactions are available only for a replica set. If we try to use multidocument transactions on a nonreplica set, we would get the error "Transaction numbers are only allowed on a replica set member or mongos," as shown in Figure 6-1.

```
WriteCommandError({
        "ok" : 0,
        "errmsg" : "Transaction numbers are only allowed on a replica set member or mongos",
        "code" : 20,
        "codeName" : "IllegalOperation"
})
```

Figure 6-1. *Error on usage of multidocument transaction on a nonreplica set*

Limitations of Transactions

Transactions do have some limitations, which are given here.

- You can specify CRUD operations only on existing collections. The collections can be in different databases.
- You cannot perform read/write operations on config, admin, and local databases.
- You cannot write to `system.*` collections.
- You cannot create or drop indexes inside a transaction.
- You cannot perform non-CRUD operations inside a transaction.
- You cannot return an operation's query plan (i.e., explain).

Transactions and Sessions

In MongoDB, transactions are associated with sessions. MongoDB's sessions provide a framework that supports consistency and writes that can be retried. MongoDB's sessions are available only for replica sets and shared clusters. A session is required to start the transaction. You cannot run a transaction outside a session, and a session can run only one transaction at a time. A session is essentially a context.

There are three commands that important in working with transactions.

- `session.startTransaction()`: To start a new transaction in the current session.
- `session.commitTransaction()`: To save changes made by the operations in the transaction.
- `session.abortTransaction()`: To abort the transaction without saving it.

Recipe 6-1. Working with Multidocument Transactions

In this recipe, we are going to discuss how to work with multidocument transactions.

Problem

You want to work with multidocument transactions.

Solution

Use `session.startTransaction()`, `session.commitTransaction()`, and `session.abortTransaction()`.

How It Works

Let's follow the steps in this section to work with multidocument transactions.

Step 1: Multidocument Transactions

To work with multidocument transactions, first we create an `employee` collection under the `employee` database as shown here.

```
use employee

db.createCollection("employee")
```

Here is the output,

```
myrs:PRIMARY> use employee
switched to db employee
myrs:PRIMARY> db.createCollection("employee")
{
        "ok" : 1,
        "operationTime" : Timestamp(1552385760, 1),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1552385760, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
```

Next, insert a few documents as shown here.

```
db.employee.insert([{_id:1001, empName:"Subhashini"},{_id:1002,
empName:"Shobana"}])
```

Here is the output,

```
myrs:PRIMARY> db.employee.insert([{_id:1001,
empName:"Subhashini"},{_id:1002, empName:"Shobana"}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
```

Now, create a session as shown here.

```
session = db.getMongo().startSession()
```

Here is the output,

```
myrs:PRIMARY> session = db.getMongo().startSession()
session { "id" : UUID("55d56ef2-cab1-40c0-8d01-c2f75b5696b5") }
```

Next, start the transaction and insert a few documents as shown here.

```
session.startTransaction()
session.getDatabase("employee").employee.insert([{_
id:1003,empName:"Taanushree"},{_id:1004, empName:"Aruna M S"}])
```

Here is the output,

```
myrs:PRIMARY> session.startTransaction()
myrs:PRIMARY> session.getDatabase("employee").employee.
insert([{_id:1003,empName:"Taanushree"},{_id:1004,
empName:"Aruna M S"}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
```

Now, we try to read the collection from inside and outside of the transactions. First, we will read the collection from inside the transaction.

```
session.getDatabase("employee").employee.find()
```

Here is the output,

```
myrs:PRIMARY> session.getDatabase("employee").employee.find()
{ "_id" : 1001, "empName" : "Subhashini" }
{ "_id" : 1002, "empName" : "Shobana" }
{ "_id" : 1003, "empName" : "Taanushree" }
{ "_id" : 1004, "empName" : "Aruna M S" }
```

We can see the modifications from inside the transaction.

Second, we will try to read the collection from outside of the transaction.

```
db.employee.find()
```

```
myrs:PRIMARY> db.employee.find()
{ "_id" : 1001, "empName" : "Subhashini" }
{ "_id" : 1002, "empName" : "Shobana" }
```

Because the transactions are not committed, we cannot see modifications outside the transaction.

Issue the following command to commit the transaction.

```
session.commitTransaction()
```

Here is the ouput:

```
myrs:PRIMARY> session.commitTransaction()
```

Now, we can see the modifications outside of the transaction also.

```
myrs:PRIMARY> db.employee.find()
{ "_id" : 1001, "empName" : "Subhashini" }
{ "_id" : 1002, "empName" : "Shobana" }
{ "_id" : 1003, "empName" : "Taanushree" }
{ "_id" : 1004, "empName" : "Aruna M S" }
```

Recipe 6-2. Isolation Test Between Two Concurrent Transactions

In this recipe, we are going to discuss how to perform an isolation test between two concurrent transactions.

Problem

You want to perform an isolation test between two concurrent transactions.

Solution

Use `session.startTransaction()`, `session.commitTransaction()` and `session.abortTransaction()`.

How It Works

Let's follow the steps in this section to perform an isolation test between two concurrent transactions.

Step 1: Isolation Test Between Two Concurrent Transactions

Create a first connection as shown here.

```
var session1 = db.getMongo().startSession()
session1.startTransaction()
session1.getDatabase("employee").employee.update({_id:1003},
{$set:{designation: "TL" }})
```

Here is the output,

```
myrs:PRIMARY> var session1 = db.getMongo().startSession()
myrs:PRIMARY> session1.startTransaction()
myrs:PRIMARY> session1.getDatabase("employee").employee.
update({_id:1003},{$set:{designation: "TL" }})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Next, read the collection as shown here.

```
session1.getDatabase("employee").employee.find()
```

Here is the output,

```
myrs:PRIMARY> session1.getDatabase("employee").employee.find()
{ "_id" : 1001, "empName" : "Subhashini" }
{ "_id" : 1002, "empName" : "Shobana" }
{ "_id" : 1003, "empName" : "Taanushree", "designation" : "TL" }
{ "_id" : 1004, "empName" : "Aruna M S" }
```

Now, create a second connection and update the documents as shown here.

```
var session2 = db.getMongo().startSession()
session2.startTransaction()
session2.getDatabase("employee").employee.update({_id:{$in:
[1001,1004]}},{$set:{designation:"SE"}},{multi:"true"})
```

Here is the output,

```
myrs:PRIMARY> var session2 = db.getMongo().startSession()
myrs:PRIMARY> session2.startTransaction()
myrs:PRIMARY> session2.getDatabase("employee").employee.
update({_id:{$in:[1001,1004]}},{$set:{designation:"SE"}},{multi
:"true"})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

Next, read the collection as shown here.

```
session2.getDatabase("employee").employee.find()
```

Here is the output,

```
myrs:PRIMARY> session2.getDatabase("employee").employee.find()
{ "_id" : 1001, "empName" : "Subhashini", "designation" : "SE" }
{ "_id" : 1002, "empName" : "Shobana" }
{ "_id" : 1003, "empName" : "Taanushree" }
{ "_id" : 1004, "empName" : "Aruna M S", "designation" : "SE" }
```

Here, the transactions are isolated, and each transaction shows the modification that it has made itself.

Recipe 6-3. Transactions with Write Conflicts

In this recipe, we are going to discuss write conflicts with transactions.

Problem

You want to see the error message for write conflicts that occurs when two transactions try to modify the same document.

Solution

Use `session.startTransaction()`, `session.commitTransaction()` and `session.abortTransaction()`.

How It Works

Let's follow the steps in this section to manage write conflicts between two concurrent transactions.

Step 1: Transactions with Write Conflicts

Create a first connection as shown here.

```
var session1 = db.getMongo().startSession()
session1.startTransaction()
session1.getDatabase("employee").employee.update({empName:
"Subhashini"},{$set:{empName: "Subha" }})
```

Here is the output,

```
myrs:PRIMARY> var session1 = db.getMongo().startSession()
myrs:PRIMARY> session1.startTransaction()
myrs:PRIMARY> session1.getDatabase("employee").employee.update(
{empName:"Subhashini"},{$set:{empName: "Subha" }})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
myrs:PRIMARY>
```

Now, create a second connection and try to update the same document as shown here.

```
var session2 = db.getMongo().startSession()
session2.startTransaction()
session2.getDatabase("employee").employee.update({empName:
"Subhashini"},{$set:{empName: "Subha" }})
```

Here is the output,

```
myrs:PRIMARY> var session2 = db.getMongo().startSession()
myrs:PRIMARY> session2.startTransaction()
myrs:PRIMARY> session2.getDatabase("employee").employee.update(
{empName:"Subhashini"},{$set:{empName: "Subha" }})
WriteCommandError({
        "errorLabels" : [
                "TransientTransactionError"
```

```
        ],
        "operationTime" : Timestamp(1552405529, 1),
        "ok" : 0,
        "errmsg" : "WriteConflict",
        "code" : 112,
        "codeName" : "WriteConflict",
        "$clusterTime" : {
                "clusterTime" : Timestamp(1552405529, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                 AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
})
```

Here, MongoDB detects a write conflict immediately, even though the transactions are not yet committed.

Recipe 6-4. Discarding Data Changes with `abortTransaction`

In this recipe, we are going to discuss how to discard data changes with `abortTransaction`.

Problem

You want to discard data changes with `abortTransaction`.

Solution

Use `session.startTransaction()`, `session.commitTransaction()`, and `session.abortTransaction()`.

How It Works

Let's follow the steps in this section to discard data changes using the `abortTransaction` method.

Step 1: Discard Data Changes

Create a `student` collection under the `student` database as shown here.

```
use student;
db.createCollection("student")
```

Here is the output,

```
myrs:PRIMARY> use student;
switched to db student
myrs:PRIMARY> db.createCollection("student")
{
        "ok" : 1,
        "operationTime" : Timestamp(1552406937, 1),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1552406937, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAA
                                AAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
myrs:PRIMARY>
```

Next, insert a few documents as shown here.

```
db.student.insert({_id:1001,name:"subhashini"})
db.student.insert({_id:1002,name:"shobana"})
```

Start a transaction and insert a document.

```
var session1 = db.getMongo().startSession()
session1.startTransaction()
session1.getDatabase("student").student.insert({_id:1003,
name:"Taanushree"})
```

Here is the output,

```
myrs:PRIMARY> var session1 = db.getMongo().startSession()
myrs:PRIMARY> session1.startTransaction()
myrs:PRIMARY> session1.getDatabase("student").student.insert
({_id:1003,name:"Taanushree"})
WriteResult({ "nInserted" : 1 })
myrs:PRIMARY>
```

Now, find the document from the collection and session.

```
db.student.find()
```

Here is the output,

```
myrs:PRIMARY> db.student.find()
{ "_id" : 1001, "name" : "subhashini" }
{ "_id" : 1002, "name" : "shobana" }
myrs:PRIMARY>session1.getDatabase("student").student.find()
myrs:PRIMARY> session1.getDatabase("student").student.find()
{ "_id" : 1001, "name" : "subhashini" }
{ "_id" : 1002, "name" : "shobana" }
{ "_id" : 1003, "name" : "Taanushree" }
myrs:PRIMARY>
```

Now, you can discard the data changes with `abortTransaction` as shown here.

```
session1.abortTransaction()
```

Here is the output,

```
myrs:PRIMARY> session1.abortTransaction()
```

Now, you can find the collection as shown here.

```
db.student.find()
myrs:PRIMARY> db.student.find()
{ "_id" : 1001, "name" : "subhashini" }
{ "_id" : 1002, "name" : "shobana" }
```

Here, the data changes are discarded.

Note Multidocument transactions are only available for deployments that use WiredTiger storage engine.

CHAPTER 7

MongoDB Monitoring and Backup

In Chapter 6, we discussed transactions in MongoDB. In this chapter, we are going to discuss the following topics:

- MongoDB monitoring tools.
- Backup and restore with MongoDB.

MongoDB Monitoring

MongoDB provides various tools to monitor the database. These MongoDB monitoring tools help us to understand what is happening with a database at various levels.

Log File

The first basic tool is the log file.

Recipe 7-1. Working with MongoDB Log Files

In this recipe, we are going to discuss MongoDB log files.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_7

Problem

You want to see the MongoDB log file and set the log level.

Solution

The MongoDB log file is present in the following installation path:

```
c:\Program Files\MongoDB\Server\4.0\log>
```

Use the `db.setLogLevel` function to set the log level.

How It Works

Let's follow the steps in this section to see the log file in `mongo` shell.

Step 1: To Display the Log File Content and Set the Log Level

Type the following command in the `mongo` shell.

```
show logs
```

Here is the output,

```
> show logs
global
startupWarnings
```

You can see the log entries by typing the next command.

```
show log global
```

We cannot filter the log messages in the console.

MongoDB has several logging levels, which can be set using the `db.setLogLevel` function.

The syntax for db.setLogLevel is

```
db.setLogLevel(<level>, <component>)
```

Here, `level` indicates the log verbosity level, which ranges from 0 to 5. The default log level is 0, which includes informational messages. Levels 1 to 5 increase the verbosity level to include debug messages. The second argument, `component`, is optional. This refers to the name of the component for which you want to set the verbosity level. The component name corresponds to the `<name>` from the corresponding `systemLog.component.<name>.verbosity` setting:

```
accessControl
command
control
ftdc
geo
index
network
query
replication
recovery
sharding
storage
storage.journal
transaction
write
```

To set the log level to 2 on the topic `query`, issue the following command.

```
db.setLogLevel(2,"query")
```

Here is the output,

```
> db.setLogLevel(2,"query")
{
        "was" : {
                "verbosity" : 0,
                "accessControl" : {
                        "verbosity" : -1
                },
                "command" : {
                        "verbosity" : -1
                },
                "control" : {
                        "verbosity" : -1
                },
                "executor" : {
                        "verbosity" : -1
                },
                "geo" : {
                        "verbosity" : -1
                },
                "index" : {
                        "verbosity" : -1
                },
                "network" : {
                        "verbosity" : -1,
                        "asio" : {
                                "verbosity" : -1
                        },
                        "bridge" : {
                                "verbosity" : -1
                        }
                },
```

```
"query" : {
        "verbosity" : -1
},
"replication" : {
        "verbosity" : -1,
        "heartbeats" : {
                "verbosity" : -1
        },
        "rollback" : {
                "verbosity" : -1
        }
},
"sharding" : {
        "verbosity" : -1
},
"storage" : {
        "verbosity" : -1,
        "recovery" : {
                "verbosity" : -1
        },
        "journal" : {
                "verbosity" : -1
        }
},
"write" : {
        "verbosity" : -1
},
"ftdc" : {
        "verbosity" : -1
},
```

```
                "tracking" : {
                        "verbosity" : -1
                },
                "transaction" : {
                        "verbosity" : -1
                }
        },
        "ok" : 1
}
```

MongoDB echoes back the previous configuration before the new setting was applied. At the top, you can see the default level, which indicates that any components that are not set will log at this level. Here, it will override only the query component. The -1 verbosity indicates that it inherits the default level from its parent.

To see the current level, issue the following command.

```
db. getLogComponents
```

Here is the output,

```
> db.getLogComponents()
{
        "verbosity" : 0,
        "accessControl" : {
                "verbosity" : -1
        },
        "command" : {
                "verbosity" : -1
        },
        "control" : {
                "verbosity" : -1
        },
```

```
"executor" : {
        "verbosity" : -1
},
"geo" : {
        "verbosity" : -1
},
"index" : {
        "verbosity" : -1
},
"network" : {
        "verbosity" : -1,
        "asio" : {
                "verbosity" : -1
        },
        "bridge" : {
                "verbosity" : -1
        }
},
"query" : {
        "verbosity" : 2
},
"replication" : {
        "verbosity" : -1,
        "heartbeats" : {
                "verbosity" : -1
        },
        "rollback" : {
                "verbosity" : -1
        }
},
```

```
        "sharding" : {
                "verbosity" : -1
        },
        "storage" : {
                "verbosity" : -1,
                "recovery" : {
                        "verbosity" : -1
                },
                "journal" : {
                        "verbosity" : -1
                }
        },
        "write" : {
                "verbosity" : -1
        },
        "ftdc" : {
                "verbosity" : -1
        },
        "tracking" : {
                "verbosity" : -1
        },
        "transaction" : {
                "verbosity" : -1
        }
}
>
```

Here, the query level is set to 2.

Now, issue the following query, which we'll use to add an entry to the log file:

```
> db.demos.find()
>
```

This query will not return any results because there is no collection named demos in the test database.

Now, issue this command to see the log file.

```
> show log global
```

Here are the last few lines of log file output,

```
2019-02-17T16:08:43.681+0530 D QUERY    [conn1]
Collection test.demos does not exist. Using EOF plan: query: {}
sort: {} projection: {}
2019-02-17T16:09:03.805+0530 D QUERY    [conn1] Collection
test.demos does not exist. Using EOF plan: query: {} sort: {}
projection: {}
2019-02-17T16:09:31.633+0530 D QUERY    [LogicalSessionCache
Refresh] Using idhack: { _id: { id: UUID("9d7def64-8b49-4c85-
9392-01f7ab88e019"), uid: BinData(0, E3B0C44298FC1C149AFBF4C8
996FB92427AE41E4649B934CA495991B7852B855) } }
```

You can see some additional information about the query.

Now set the query log level as -1:

```
> db.setLogLevel(-1,"query")
```

Then, issue the following query and observe the details in the log file as explained further later.

```
> db.demos.find({y:1})
```

Now check the log file by using the following command.

```
> show log global
```

Now check the end of the log file: We cannot see the extra logging information for this find query.

Use db.setLogLevel(0) to turn off the logging level.

If we do not specify the option as `query` or `write` or anything, it will be applied for all options.

The log level 5 is very detailed; levels 1 to 3 are more useful options and these are recommended unless you need all of the detailed information provided by the highest log level of 5.

Also, setting the appropriate log level helps us to improve the performance of queries by identifying the query plans logged in the log file and also by identifying the slow operations by checking the time taken to execute a query.

Note Refer to Recipe 7-10 to learn more about MongoDB query plans.

MongoDB Performance

It is mandatory to analyze the performance of the database when we develop new applications with MongoDB. Performance degradation could happen due to hardware availability issues, number of open database connections, and database access strategies.

Performance issues indicate the database is operating at full capacity and abnormal traffic load due to an increase in the number of connections.

The database profiler can help us to understand the various operations performed on the database that cause performance degradation.

Database Profiler

The database profiler is used to collect information about the commands that are executed against a running `mongod` instance. The database profiler writes all the collected data to the `system.profile` capped collection in the `admin` database.

MongoDB also has a Performance Advisor tool that automatically monitors slow queries and suggests new indexes to improve query performance. A query is considered to be slow if it takes more than 100 milliseconds to execute.

Note The Performance Advisor tool is available in the MongoDB Atlas cluster. MongoDB Atlas is the global cloud database service for modern applications.

The profiler is turned off by default. It can be enabled on a per-database or per-instance basis at any one of the profiling levels mentioned in Table 7-1.

Table 7-1. *Profiling Levels*

Level	Description
0	The profiler is off and does not collect any data. This is the default profiler level.
1	The profiler collects data for operations that take longer than the value of the `slowms` parameter.
2	The profiler collects data for all operations.

Recipe 7-2. Working with Database Profiler

In this recipe, we are going to discuss how to enable database profiler and how to set the profiling level.

Problem

You want to enable database profiling with a profile level.

Solution

Use the `db.setProfilingLevel()` helper in the `mongo` shell.

How It Works

Let's follow the steps in this section to enable the database profiler.

Step 1: Enable and Configure Database Profiler

To enable the database profiler to collect all database information use the following command.

```
db.setProfilingLevel(2)
```

Here is the output,

```
> db.setProfilingLevel(2)
{ "was" : 0, "slowms" : 100, "sampleRate" : 1, "ok" : 1 }
```

`was` is the key/value pair indicating the previous level of profiling.

You can specify the threshold for slow operations using the next command.

```
db.setProfilingLevel(1, { slowms: 30 })
```

Here is the output,

```
> db.setProfilingLevel(1, { slowms: 30 })
{ "was" : 2, "slowms" : 100, "sampleRate" : 1, "ok" : 1 }
```

Here, the slow operation threshold is 30 milliseconds.

To profile a random sample of slow operations, use the following command.

```
db.setProfilingLevel(1, { sampleRate: 0.53 })
```

That code sets the profiler to sample 53% of all slow operations.

Note The default threshold value for the slow operation is 100 milliseconds.

You can get the profiling level by issuing the following command.

```
db.getProfilingStatus()
```

Here is the output,

```
> db.getProfilingStatus()
{ "was" : 1, "slowms" : 30, "sampleRate" : 1 }
```

The `was` field indicates the current profiling level.

To disable the database profiler, use this command.

```
db.setProfilingLevel(0)
```

Here is the output,

```
> db.setProfilingLevel(0)
{ "was" : 1, "slowms" : 30, "sampleRate" : 1, "ok" : 1 }
```

To enable profiling for an entire `mongod` instance, pass the following option to the `mongod` instance at the time of starting it.

```
mongod --profile 1 --slowms 20 --slowOpSampleRate 0.5
```

This command sets the profiling level to 1, the slow operation threshold to 20 milliseconds, and it profiles only 50% of the slow operations.

Recipe 7-3. View Database Profiler

In this recipe, we are going to discuss how to view database profiler.

Problem

You want to view database profiler information.

Solution

Use `db.system.profile.find()` in the `mongo` shell.

How It Works

Let's follow the steps in this section to view database profiler information.

Step 1: Enable and Configure Database Profiler

The database profiler logs information in the `system.profile` collection. You need to query `system.collection` to view profiling information.

Create an `employee` collection in the `example` database as shown here.

```
> use example;
switched to db example
> db.setProfilingLevel(2)
{ "was" : 0, "slowms" : 100, "sampleRate" : 1, "ok" : 1 }
> show collections
> db.employee.insert({_id:1001,name:"Subhashini"})
WriteResult({ "nInserted" : 1 })
> db.employee.insert({_id:1001,name:"Shobana"})
WriteResult({
        "nInserted" : 0,
        "writeError" : {
```

```
                "code" : 11000,
                "errmsg" : "E11000 duplicate key error
                           collection: example.employee index:
                           _id_ dup key: { : 1001.0 }"
        }
})
> db.employee.find()
{ "_id" : 1001, "name" : "Subhashini" }
```

The profiler stores details in their respective keys as follows:

- `op`: Component type (i.e., query or command).
- `ts`: Timestamp.
- `ns`: Collection details where the query is executed.

These options can be used to filter or sort as in the queries that follow.

To return operations performed on the `employee` collection, issue the following command.

```
> db.system.profile.find( { ns : 'example.employee' }
).pretty()
```

Here is the output,

```
{
        "op" : "insert",
        "ns" : "example.employee",
        "command" : {
                "insert" : "employee",
                "ordered" : true,
                "lsid" : {
                        "id" : UUID("93b33e8d-f561-46a1-8310-
                               0f62d31442f1")
                },
```

```
                "$db" : "example"
        },
        "ninserted" : 1,
        "keysInserted" : 1,
        "numYield" : 0,
        "locks" : {
                "Global" : {
                        "acquireCount" : {
                                "r" : NumberLong(3),
                                "w" : NumberLong(3)
                        }
                },
                "Database" : {
                        "acquireCount" : {
                                "w" : NumberLong(2),
                                "W" : NumberLong(1)
                        }
                },
                "Collection" : {
                        "acquireCount" : {
                                "w" : NumberLong(2)
                        }
                }
        },
        "responseLength" : 45,
        "protocol" : "op_msg",
        "millis" : 21,
        "ts" : ISODate("2019-01-14T06:08:38.838Z"),
        "client" : "127.0.0.1",
        "appName" : "MongoDB Shell",
        "allUsers" : [ ],
        "user" : ""
}
```

```
{
        "op" : "insert",
        "ns" : "example.employee",
        "command" : {
                "insert" : "employee",
                "ordered" : true,
                "lsid" : {
                        "id" : UUID("93b33e8d-f561-46a1-8310-
                                0f62d31442f1")
                },
                "$db" : "example"
        },
        "ninserted" : 0,
        "keysInserted" : 0,
        "numYield" : 0,
        "locks" : {
                "Global" : {
                        "acquireCount" : {
                                "r" : NumberLong(1),
                                "w" : NumberLong(1)
                        }
                },
                "Database" : {
                        "acquireCount" : {
                                "w" : NumberLong(1)
                        }
                },
                "Collection" : {
                        "acquireCount" : {
                                "w" : NumberLong(1)
                        }
```

```
                }
        },
        "responseLength" : 194,
        "protocol" : "op_msg",
        "millis" : 2,
        "ts" : ISODate("2019-01-14T06:08:50.297Z"),
        "client" : "127.0.0.1",
        "appName" : "MongoDB Shell",
        "allUsers" : [ ],
        "user" : ""
}
{
        "op" : "query",
        "ns" : "example.employee",
        "command" : {
                "find" : "employee",
                "filter" : {

                },
                "lsid" : {
                        "id" : UUID("93b33e8d-f561-46a1-8310-
                                0f62d31442f1")
                },
                "$db" : "example"
        },
        "keysExamined" : 0,
        "docsExamined" : 1,
        "cursorExhausted" : true,
        "numYield" : 0,
        "nreturned" : 1,
        "locks" : {
                "Global" : {
```

```
                "acquireCount" : {
                        "r" : NumberLong(1)
                }
        },
        "Database" : {
                "acquireCount" : {
                        "r" : NumberLong(1)
                }
        },
        "Collection" : {
                "acquireCount" : {
                        "r" : NumberLong(1)
                }
        }
},
"responseLength" : 147,
"protocol" : "op_msg",
"millis" : 0,
"planSummary" : "COLLSCAN",
"execStats" : {
        "stage" : "COLLSCAN",
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 0,
        "works" : 3,
        "advanced" : 1,
        "needTime" : 1,
        "needYield" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "invalidates" : 0,
```

```
                "direction" : "forward",
                "docsExamined" : 1
        },
        "ts" : ISODate("2019-01-14T06:22:07.873Z"),
        "client" : "127.0.0.1",
        "appName" : "MongoDB Shell",
        "allUsers" : [ ],
        "user" : ""
}
>
```

To return the most recent five log entries, issue the following command.

```
> db.system.profile.find().limit(5).sort( { ts : -1 }
).pretty()
```

Note `ts` specifies the timestamp and the value `-1` specifies to sort in descending order. We can specify `1` for ascending or `-1` for descending order.

To return operations slower than 5 milliseconds, use this syntax.

```
> db.system.profile.find( { millis : { $gt : 5 } } ).pretty()
```

mongostat

`mongostat` is a monitoring utility that provides information about a `mongod` instance if you are working on a single `mongod` instance. If you are working on a shared cluster, it shows information about a `mongos` instance.

Recipe 7-4. Working with `mongostat`

In this recipe, we are going to discuss how to use `mongostat` to monitor the MongoDB server.

Problem

You want to see the details of a MongoDB server.

Solution

Use the `mongostat` tool, which is available in the installation directory. Usually the installation directory is `C:\Program Files\MongoDB\Server\4.0\bin` or the custom path specified during your installation process.

Avoid navigating to the directory each time to use the `mongostat` tool and add the directory path to your `PATH` environment variable.

How It Works

Let's follow the steps in this section to view server information using `mongostat`.

Step 1: `mongostat` Command

Issue the `mongostat` command by specifying the hostname and port of MongoDB server.

Here is the output,

```
c:\Program Files\MongoDB\Server\4.0\bin>mongostat --host
localhost:27017

insert query update delete getmore command dirty used flushes
vsize   res qrw arw net_in net_out conn                time
    *0    *0     *0     *0       0   146|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0  11.6k   4.49m    2 Feb 17 16:59:19.131
    *0    *0     *0     *0       0     2|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   163b   63.6k    2 Feb 17 16:59:20.095
    *0    *0     *0     *0       0     2|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   158b   61.7k    2 Feb 17 16:59:21.090
    *0    *0     *0     *0       0     1|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   157b   61.3k    2 Feb 17 16:59:22.090
    *0    *0     *0     *0       0     1|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   157b   61.2k    2 Feb 17 16:59:23.093
    *0    *0     *0     *0       0     2|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   158b   61.4k    2 Feb 17 16:59:24.092
    *0    *0     *0     *0       0     2|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   159b   62.1k    2 Feb 17 16:59:25.080
    *0    *0     *0     *0       0     1|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   157b   61.2k    2 Feb 17 16:59:26.082
    *0    *0     *0     *0       0     1|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   157b   61.0k    2 Feb 17 16:59:27.088
    *0    *0     *0     *0       0     2|0  0.0% 0.0%       0
4.97G 28.0M 0|0 1|0   158b   61.6k    2 Feb 17 16:59:28.083
```

Note 27017 is the default port. We can simply use `mongostat` without any options if we want to connect and get statistics of localhost and default port 27017. To connect a different or remote instance, specify the option `-host` as shown.

You can see that there are multiple columns in the output.

- `insert/query/update/delete`: These columns show the number of insert, query, update, and delete operations per second.
- `getmore`: This column shows the number of times the `getmore` operation is executed in one second.
- `command`: This column shows the number of commands executed on the server in one second.
- `flushes`: This column shows the number of times data was flushed to disk in one second.
- `mapped`: This column shows the amount of memory used by the mongo process against a database. It is the same as the size of the database.
- `vsize` *(virtual size):* This column represents virtual memory allocated to the entire `mongod` process.
- `res` *(resident memory)*: This column represents the physical memory used by MongoDB.
- `faults`: This column shows the number of Linux page faults per second.
- `qr|qw`: This column shows queued-up reads and writes that are waiting for the chance to be executed.
- `ar|aw`: This column shows number of active clients.
- `netIn and netOut`: These columns show the network traffic in and out of the MongoDB server within a given time frame.

- `conn`: This column shows the number of open connections.
- `time`: This column shows the time frame in which operations are performed.

`mongotop`

`mongotop` provides a method to track the amount of time spent by the `mongod` process during reads and writes. `mongotop` provides statistics on a per-collection level.

Recipe 7-5. Working with `mongotop`

In this recipe, we are going to discuss how to use `mongotop` to track the amount of time spent by the `mongod` process during reads and writes.

Problem

You want to see the amount of time spent by the `mongod` process during reads and writes.

Solution

Issue the `mongotop` command in the installation path of MongoDB.

How It Works

Let's follow the steps in this section to view the amount of time spent during reads and writes.

Step 1: mongotop Command

Issue the following command.

```
c:\Program Files\MongoDB\Server\4.0\bin>mongotop
2019-02-17T21:51:25.975+0530    connected to: 127.0.0.1

                    ns    total    read    write    2019-02-
                                                    17T21:
                                                    51:26+
                                                    05:30
     admin.system.roles      0ms     0ms      0ms
   admin.system.version      0ms     0ms      0ms
 config.system.sessions      0ms     0ms      0ms
           emp.employee      0ms     0ms      0ms
        employee.author      0ms     0ms      0ms
       employee.authors      0ms     0ms      0ms
    employee.categories      0ms     0ms      0ms
      employee.employee      0ms     0ms      0ms
employee.employeedetails     0ms     0ms      0ms
          employee.names     0ms     0ms      0ms
```

db.stats()

`db.stats()` provides the statistics of a single database.

Recipe 7-6. Working with db.stats()

In this recipe, we are going to discuss how to get the statistics for a single database.

Problem

You want to see the disk and memory usage estimates for a database.

Solution

Use the `db.stats` command to display the statistics for a database.

How It Works

Let's follow the steps in this section to view the statistics for the database.

Step 1: `db.stats()` Command

Issue the following commands to display the statistics for a database named `employee`.

```
> use employee
switched to db employee
> db.stats()
{
        "db" : "employee",
        "collections" : 12,
        "views" : 0,
        "objects" : 40,
        "avgObjSize" : 90.875,
        "dataSize" : 3635,
        "storageSize" : 401408,
        "numExtents" : 0,
        "indexes" : 12,
        "indexSize" : 212992,
        "fsUsedSize" : 208018800640,
        "fsTotalSize" : 509929914368,
        "ok" : 1
}
>
```

The details such as data size and storage size are given in bytes. To see the information in megabytes, issue the following command.

```
> db.stats(1000000)
{
        "db" : "employee",
        "collections" : 12,
        "views" : 0,
        "objects" : 40,
        "avgObjSize" : 90.875,
        "dataSize" : 0.003635,
        "storageSize" : 0.401408,
        "numExtents" : 0,
        "indexes" : 12,
        "indexSize" : 0.212992,
        "fsUsedSize" : 208020.402176,
        "fsTotalSize" : 509929.914368,
        "ok" : 1
}
>
```

db.serverStatus()

`db.serverStatus()` returns a document that provides statistics for the state of a database process.

Recipe 7-7. Working with db.serverStatus()

In this recipe, we are going to discuss the `db.serverStatus()` command.

Problem

You want to see the memory status of a database.

Solution

Use the `db.serverStatus()` command.

How It Works

Let's follow the steps in this section to view the statistics for the memory status of a database.

Step 1: `db.serverStatus()` Command

Issue following command to view the memory statistics for a database.

```
> db.serverStatus().mem
{
        "bits" : 64,
        "resident" : 85,
        "virtual" : 5089,
        "supported" : true,
        "mapped" : 0,
        "mappedWithJournal" : 0
}
```

Backup and Restore with MongoDB Tools

MongoDB provides `mongodump` and `mongorestore` utilities to work with BSON data dumps. These utilities are useful for creating backups for small deployments. To create resilient and nondisruptive backups for large deployments, we can use file system or block-level disk snapshot methods.

We should use a system-level tool to create a copy of the device file system that holds MongoDB files. The file system snaphots or the disk-level snapshot backup method require additional system configuration outside of MongoDB, so we do not cover this in depth in this book.

When deploying MongoDB in production, we should have a backup and failover strategy for capturing and restoring backups in the case of data loss events.

Recipe 7-8. Working with `mongodump`

In this recipe, we are going to discuss how to back up data using `mongodump`.

Problem

You want to back up data using `mongodump`.

Solution

Use the `mongodump` command.

How It Works

Let's follow the steps in this section to back up data.

The `mongodump` utility makes a backup by connecting to a running `mongod` or `mongos` instance.

When you run the `mongodump` command without specifying any arguments, it connects to the `mongodb` running on localhost on port 27017 and creates a database backup named `dump` in the current directory.

Let us discuss how to issue the mongodump command to create a backup.

1. Ensure mongod is running.
2. Open a command prompt with administrator privileges, navigate to the mongodb installation folder, and type mongodump as shown here.

   ```
   c:\Program Files\MongoDB\Server\4.0\bin>mongodump
   ```

3. You can see the dump folder in the current directory as shown in Figure 7-1.

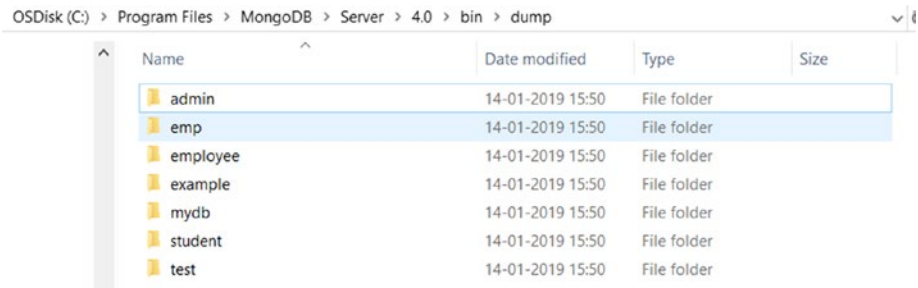


Figure 7-1. *MongoDB installation folder*

You can specify the output directory by issuing the following command.

```
mkdir "c:\Program Files\MongoDB\Server\4.0\dump"
mongodump --out "c:\Program Files\MongoDB\Server"\4.0\dump
```

To create a backup of a collection, issue the following command.

```
mongodump --collection employee --db example --out "c:\Program
Files\MongoDB\Server"\4.0\backup
```

Here is the output,

```
c:\Program Files\MongoDB\Server\4.0\bin>mongodump --collection
employee --db example --out "c:\Program Files\MongoDB\
Server"\4.0\backup
```

```
2019-01-14T18:36:28.460+0530    writing example.employee to
2019-01-14T18:36:28.465+0530    done dumping example.employee
(1 document)
```

You can make a backup from a remote host by specifying `host` and `port` arguments as shown here.

```
mongodump --host sample.net --port 5017 --username user
--password "pass"
```

Automatic and Regular Backup Scheduling Using `mongodump`

It is good practice to perform a regular backup of all databases in case of any hard failover. Follow these steps to perform a regular backup at a specific interval.

1. We can create scripts to create the backup directory for the current date and time and export the database to the same directory.

   For Windows, create a `bat` (batch script) file with the commands shown in Figure 7-2.

```
set DIRECTORY=backup_%date:~-4,4%%date:~-10,2%%date:~7,2%

mkdir %DIRECTORY%

mongodump -h <your_database_host> -d <your_database_name> -u <username> -
p <password> -o %DIRECTORY%
```

Figure 7-2. *Creating a batch script file in Windows*

For Unix/Linux machines, create a shell script file with the commands shown in Figure 7-3.

```
DIRECTORY= backup_`date +%m%d%y`

mkdir $DIRECTORY

mongodump -h <your_database_host> -d <your_database_name> -u <username> -
p <password> -o $DIRECTORY
```

Figure 7-3. *Creating a shell script file in Unix/Linux*

2. Schedule the script file to execute at a certain interval. Use task scheduler or any other scheduler for Windows, and use the CRON schedule for Unix/Linux.

Recipe 7-9. Working with `mongorestore`

In this recipe, we are going to discuss how to restore data using `mongorestore`.

Problem

You want to restore data using `mongorestore`.

Solution

Use the `mongorestore` command.

How It Works

Let's follow the steps in this section to restore data.

Step 1: Restore Data Using `mongorestore`

To restore data, open a command prompt and navigate to the `mongodb` installation folder, then issue the following command.

```
c:\Program Files\MongoDB\Server\4.0\bin>mongorestore dump/
```

Here, `mongorestore` imports the data present in the `dump` directory to the running `mongod` instance. By default, `mongorestore` looks for a database backup in the `dump/` directory. We can connect to a different port of the active `mongod` instance and a different backup path by using this command.

```
mongorestore --port <port number> <path to the backup>
```

Step 2: Restore Data Using `mongorestore` to Remote Instances

By default, `mongorestore` connects to running instance on localhost and default port number 27017. To restore to a different host and different port, we can specify the same using the `--host` and `--port` options as shown here.

```
mongorestore --host <example.net> --port <myportNumber>
--username myuser --password 'mypass' /backup/mongodumpfile
```

Specify the `username` and `password` using options `--user` and `--password` only if the remote `mongod` instance requires authentication.

Recipe 7-10. Working with mongodb Query Plans

In this recipe, we are going to discuss query plans to understand the MongoDB query execution details with query planner and query optimizer.

Problem

You want to understand the query execution plan.

Solution

Use the `cursor.explain()` command.

How It Works

Let's follow the steps in this section to understand the query plan.

Step 1: Using `explain()` to Understand the Query Plan

To see the query plan during the MongoDB command execution, use `cursor.explain()`. Let's assume the `employee` collection is available in the `authors` database with the data shown here.

```
MongoDB Enterprise > use authors
switched to db authors
MongoDB Enterprise > db.employee.find()
{ "_id" : ObjectId("5d50ed688dcf280c50fde439"), "empId" : 1,
"empName" : "John", "state" : "KA", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43a"), "empId" : 2,
"empName" : "Smith", "state" : "CA", "country" : "US" }
```

```
{ "_id" : ObjectId("5d50ed688dcf280c50fde43b"), "empId" : 3,
"empName" : "James", "state" : "FL", "country" : "US" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43d"), "empId" : 5,
"empName" : "Joshi", "state" : "HYD", "country" : "India" }
```

To get the query plan while executing the command `db.employee.find()`, use `.explain()` as shown here.

```
MongoDB Enterprise > db.employee.find().explain()
```

Here is the output,

```
MongoDB Enterprise > db.employee.find().explain()
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "authors.employee",
                "indexFilterSet" : false,
                "parsedQuery" : {

                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
                "host" : "DESKTOP-MEISTBV",
                "port" : 27017,
                "version" : "4.0.11",
```

```
                "gitVersion" : "417d1a712e9f040d54beca8e4943edc
                                e218e9a8c"
        },
        "ok" : 1
}
```

Observe in this output that there is no `parsedQuery` in the query planner, and the winning plan says that it involves the complete collection scan.

Now, let's try to add a filter to the same query and observe the query plan.

```
MongoDB Enterprise > db.employee.find({country:"India"})
{ "_id" : ObjectId("5d50ed688dcf280c50fde439"), "empId" : 1,
"empName" : "John", "state" : "KA", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43c"), "empId" : 4,
"empName" : "Josh", "state" : "TN", "country" : "India" }
{ "_id" : ObjectId("5d50ed688dcf280c50fde43d"), "empId" : 5,
"empName" : "Joshi", "state" : "HYD", "country" : "India" }
```

We have added the filter condition to retrieve only `country:India`. Now check the query plan for the same query with the filter condition.

```
MongoDB Enterprise> db.employee.find({country:"India"}).explain()
```

Here is the output,

```
MongoDB Enterprise > db.employee.find({country:"India"}).
explain()
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "authors.employee",
                "indexFilterSet" : false,
                "parsedQuery" : {
```

```
                        "country" : {
                                "$eq" : "India"
                        }
                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "country" : {
                                        "$eq" : "India"
                                }
                        },
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
                "host" : "DESKTOP-MEISTBV",
                "port" : 27017,
                "version" : "4.0.11",
                "gitVersion" : "417d1a712e9f040d54beca8e4943e
                                dce218e9a8c"
        },
        "ok" : 1
}
```

Observe in the preceding output that there is a parsed query added by the query planner as

```
"parsedQuery" : { "country" : {"$eq" : "India" } }
```

Step 2: Understanding Different Modes in `explain()`

The `explain()` method accepts queryPlanner, executionStats, or allPlansExecution as the operating mode. The default mode is `queryPlanner` if we don't specify otherwise.

Use the following command to use `explain()` with any of these parameters as a mode.

```
db.employee.find().explain("queryPlanner")
```

Here is the output,

```
MongoDB Enterprise > db.employee.find().explain("queryPlanner")
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "authors.employee",
                "indexFilterSet" : false,
                "parsedQuery" : {

                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
                "host" : "DESKTOP-MEISTBV",
                "port" : 27017,
                "version" : "4.0.11",
```

```
                "gitVersion" : "417d1a712e9f040d54beca8e4943ed
                                ce218e9a8c"
        },
        "ok" : 1
}
```

Now use executionStats as a operation mode to the explain() method as shown here.

```
db.employee.find().explain("executionStats")
```

Here is the output,

```
MongoDB Enterprise > db.employee.find().
explain("executionStats")
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "authors.employee",
                "indexFilterSet" : false,
                "parsedQuery" : {

                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 5,
                "executionTimeMillis" : 0,
                "totalKeysExamined" : 0,
```

```
                "totalDocsExamined" : 5,
                "executionStages" : {
                        "stage" : "COLLSCAN",
                        "nReturned" : 5,
                        "executionTimeMillisEstimate" : 0,
                        "works" : 7,
                        "advanced" : 5,
                        "needTime" : 1,
                        "needYield" : 0,
                        "saveState" : 0,
                        "restoreState" : 0,
                        "isEOF" : 1,
                        "invalidates" : 0,
                        "direction" : "forward",
                        "docsExamined" : 5
                }
        },
        "serverInfo" : {
                "host" : "DESKTOP-MEISTBV",
                "port" : 27017,
                "version" : "4.0.11",
                "gitVersion" : "417d1a712e9f040d54beca8e4943ed
                                ce218e9a8c"
        },
        "ok" : 1
}
MongoDB Enterprise >
```

In this output, observe the following:

- `"executionSuccess" : true`: This shows whether the query is successfully executed or not.
- `"nReturned" : 5`: This is the number of rows returned.
- `"executionTimeMillis" : 0`: This is the execution time in milliseconds.

If the `executionTimeMillis` value is more, then it could be considered a slow query and we must rewrite the query to optimize the executiuon time by adding the required filters to retrieve only the required data. We could also consider using an index on keys for better performance.

CHAPTER 8

MongoDB Security

In Chapter 7, we discussed monitoring and backup methods in MongoDB. In this chapter, we are going to discuss the security features of MongoDB.

Authentication and Authorization

Authentication is the process of verifying the identity of the user, whereas authorization or access control determines the verified user's access to resources and operations.

MongoDB supports the following various authentication mechanisms for verifying the identity of the user.

- *SCRAM:* This is the default authentication mechanism for MongoDB. It verifies the user against their username, password, and authentication database.
- *x.509 certificates:* This mechanism uses x.509 certificates for authentication. MongoDB supports a number of authentication mechanisms that clients can use to verify their identity. These mechanisms allow MongoDB to integrate into an existing authentication system.

In addition to this, MongoDB Enterprise Edition also supports Lightweight Directory Access Protocol (LDAP) proxy authentication and Kerberos authentication.

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1_8

Note Replica sets and shared clusters require internal authentication between members when access control is enabled. Refer to Recipe 8-2 to understand the authentication process for replica sets.

Access Control

MongoDB enforces authentication to identify users and their permissions to access resources only if access control is enabled. The upcoming recipes explain the procedure to create various roles and the mechanisms to enable access control.

When you install MongoDB, it has no users. First, therefore, we need to create the admin user, which can be used to create other users.

Roles

In MongoDB, roles grant users access to MongoDB resources. MongoDB has certain built-in roles, listed in Figure 8-1, which can be assigned to any user to access resources. We can also create user-defined roles.

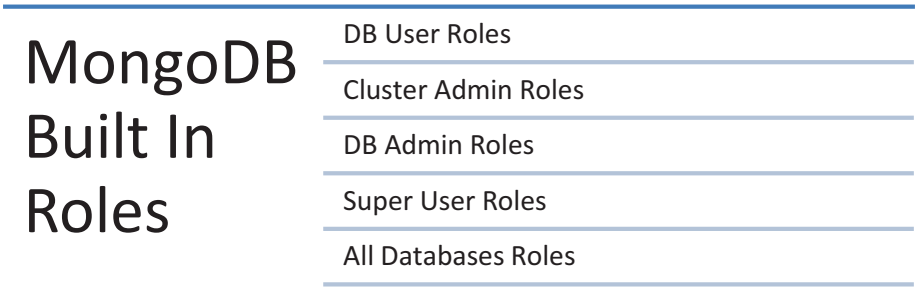


Figure 8-1. *MongoDB built-in roles*

First, let's look at the commonly used built-in roles.

MongoDB User Roles

read

The read role gives the holder access to read all of the user-created collections.

readwrite

The readwrite role gives access to read all of the user-created collections and also the ability to modify the nonsystem (i.e., user-created) collections.

MongoDB Admin Roles

dbAdmin

The dbAdmin role could help us to perform administrative tasks such as schema-related tasks, indexing, and collecting statistics. We cannot use this role to grant privileges for users and others' role management.

dbOwner

We can perform any administrative tasks on a database using the dbOwner role. This role also includes the other role previliges such as the readWrite, dbAdmin, and userAdmin roles.

userAdmin

We can create and modify roles and users on the current active database.

Cluster Admin Roles

clusterAdmin

The clusterAdmin role provides cluster management access. This role also includes other role privileges, such as clusterManager, clusterMonitor, and hostManager roles.

clusterManager

This role helps us to manage and monitor user actions on the mongodb cluster.

Recipe 8-1. Creating a Superuser and Authenticating a User

In this recipe, we are going to discuss how to create a superuser and authenticate a user.

Problem

You want to create a superuser and authenticate a user.

Solution

Here are the commands:

```
db.createUser()
mongo --username username --password password
--authenticationDatabase databasename
db.auth()
```

How It Works

Let's follow the steps in this section to create a superuser.

Step 1: Create a Superuser

Connect to a mongod instance and type the following command.

```
> use admin
> db.createUser({user:"superUserAdmin",pwd:"1234",roles:
[{role:    "userAdminAnyDatabase",db:"admin"}]})
```

Here is the output,

```
> use admin
switched to db admin
> db.createUser({user:"superUserAdmin",pwd:"1234",roles:[{role:
"userAdminAnyDatabase",db:"admin"}]})
Successfully added user: {
        "user" : "superUserAdmin",
        "roles" : [
                {
                        "role" : "userAdminAnyDatabase",
                        "db" : "admin"
                }
        ]
}
>
```

To authenticate using the mongo shell, do with either of the following.

- Use MongoDB command-line authentication options (i.e., `--username`, `--password`, and `–authenticationDatabase`) when connecting to a mongod or mongos instance.
- Connect to a mongod or mongos instance, then run the `db.auth` method against the authentication database.

Step 2: Authenticate a User

Type the following command in the terminal (command prompt) while connecting to a mongo shell.

```
> mongo --username superUserAdmin --password 1234
--authenticationDatabase admin
```

You will be connected to mongoshell. You can also use db.auth() to authenticate a user.

Connect to a mongod instance by issuing the following command.

```
> use admin
> db.auth("superUserAdmin","1234")
```

Here is the output,

```
> use admin
switched to db admin
> db.auth("superUserAdmin","1234")
1
>
```

Step 3: Create a New User with Read and Write Access Control

Type the following command to create a new user with read and write access control for the admin database.

```
> db.createUser(
  {
    user: "user1000",pwd: "abc123",
    roles: [ { role: "readWriteAnyDatabase", db:"admin" } ]
  }
  )
```

Here is the output,

```
> db.createUser(
...   {
...     user: "user1000",
...     pwd: "abc123",
...     roles: [ { role: "readWriteAnyDatabase", db:"admin" } ]
...   }
... )
Successfully added user: {
        "user" : "user1000",
        "roles" : [
                {
                        "role" : "readWriteAnyDatabase",
                        "db" : "admin"
                }
        ]
}
>
```

Now, you can restart your MongoDB with access control enabled as shown here.

```
mongod.exe --auth
```

Recipe 8-2. Authenticating a Server in a Replica Set Using a Key File

In this recipe, we are going to discuss how to authenticate a server in a replica set using a key file.

Problem

You want to authenticate a server in a replica set using a key file.

Solution

Use a replica set and a key file.

How It Works

Let's follow the steps in this section to authenticate a server in a replica set using a key file.

Step 1: Authenticate a Server in a Replica Set Using a Key File

First, we will create a three-member replica set by following the procedure given here.

Create three data directories as shown here.

```
md c:\mongodb\repset\rs1
md c:\mongodb\repset\rs2
md c:\mongodb\repset\rs3
```

Here is the output,

```
c:\>md c:\mongodb\repset\rs1
c:\>md c:\mongodb\repset\rs2
c:\>md c:\mongodb\repset\rs3
```

Next, start three `mongod` instances as shown here.

```
start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs1
--port 20001 --replSet myrs
```

Figure 8-2 shows a mongod instance that is waiting for a connection on port 20001.

```
start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs2
--port 20002 --replSet myrs
```

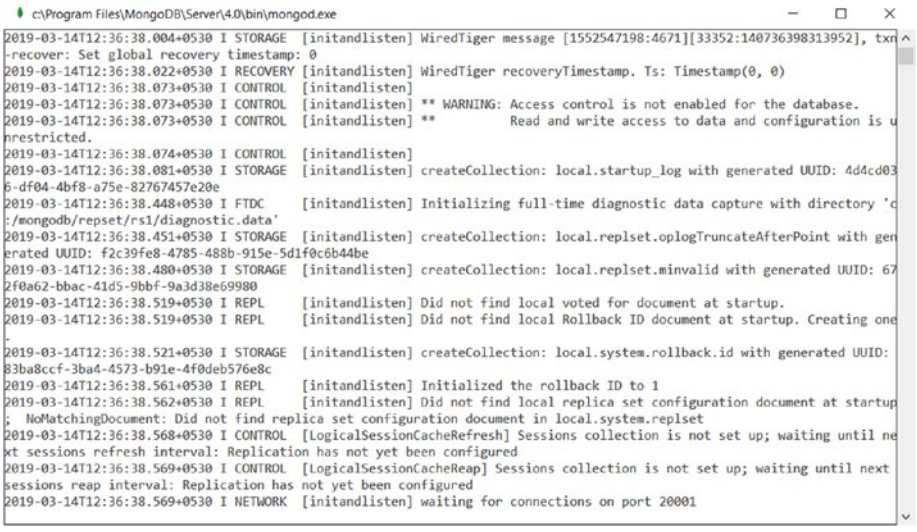


Figure 8-2. *A mongod instance waiting for connection on port 20001*

Figure 8-3 shows a mongod instance that is waiting for connection on port 20002.

```
start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs3
--port 20003 --replSet myrs
```

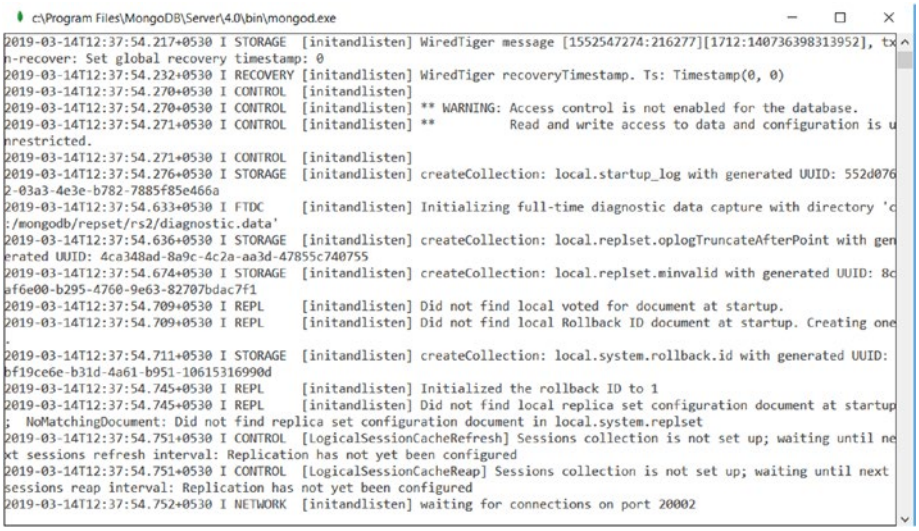


Figure 8-3. *A mongod instance waiting for connection on port 20002*

Figure 8-4 shows a mongod instance that is waiting for connection on port 20003.

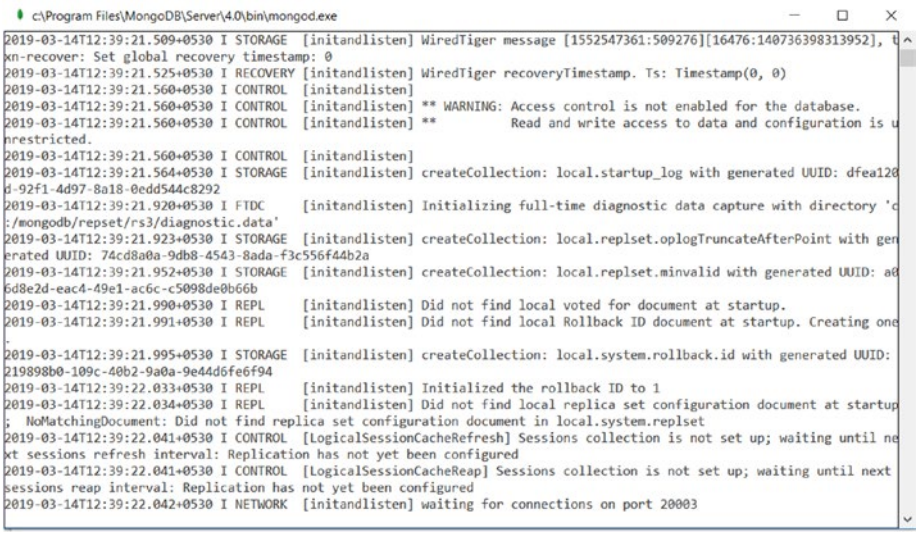


Figure 8-4. *A mongod instance waiting for connection on port 20003*

Issue the following command to connect to a mongod instance running on port 20001.

```
mongo hostname:20001
```

Issue the following commands in the mongo shell to create a three-member replica set.

```
rs.initiate();   // to initiate replica set

rs.add("hostname:20002");  // to add secondary

rs.add("hostname:20003");    // to add secondary
```

Now, create a superuser as shown here.

```
use admin
db.createUser(
  {
    user: "subhashini",
    pwd: "abc123",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

Here is the output,

```
myrs:PRIMARY> use admin
switched to db admin
myrs:PRIMARY> db.createUser(
...   {
...     user: "subhashini",
...     pwd: "abc123",
...     roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
...   }
... )
```

```
Successfully added user: {
        "user" : "subhashini",
        "roles" : [
                {
                        "role" : "userAdminAnyDatabase",
                        "db" : "admin"
                }
        ]
}
myrs:PRIMARY>
```

Next, shut down all the servers. To shut down a `mongod`, connect each `mongod` using a `mongo` shell and issue the `db.shutdownServer()` on the `admin` database as shown here.

```
use admin
db.shutdownServer()
```

Next, generate a key file using the `openssl` method as shown here and copy the key file to each replica member.

```
openssl rand -base64 756 > /home/anyPath/keyfile
chmod 400 /home/anyPath/keyfile
```

Note We can use any method to generate the key file. To use the `openssl` method, you need to install openssl. Go to https://indy.fulgan.com/SSL/ to download and install the openssl package (note that this link might be changed in the future).

Figure 8-5 shows the files in the extracted `openssl` package.

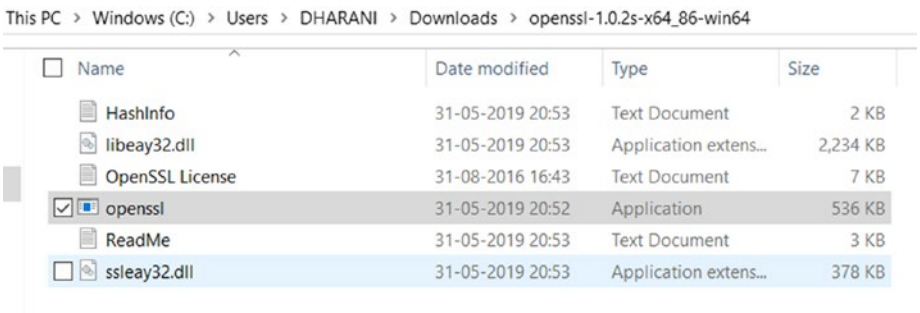


Figure 8-5. *Extracted files of the* `openssl` *package*

Use `openssl.exe` to generate the keys as mentioned earlier. Also, in Windows, the generated key file permissions will not be checked. Only on the Unix platform, the key file should not have group or other permissions.

Now, start the `mongod` instance with access control enabled as shown here.

```
start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs1
--keyFile c:\mongodb\repset\rs1\keyfile --port 20001 --replSet
myrs

start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs2
--keyFile c:\mongodb\repset\rs2\keyfile --port 20002 --replSet
myrs

start mongod --bind_ip hostname --dbpath c:\mongodb\repset\rs3
--keyFile c:\mongodb\repset\rs3\keyfile --port 20003 --replSet
myrs
```

Next, connect to the `mongo` shell and list the databases as shown here.

```
mongo hostname:20001

show dbs;
```

You might see the error message shown here.

```
myrs:PRIMARY> show dbs;
2019-03-14T13:08:48.704+0530 E QUERY    [js] Error:
listDatabases failed:{
        "operationTime" : Timestamp(1552549127, 1),
        "ok" : 0,
        "errmsg" : "command listDatabases requires
                   authentication",
        "code" : 13,
        "codeName" : "Unauthorized",
        "$clusterTime" : {
                "clusterTime" : Timestamp(1552549127, 1),
                "signature" : {
                        "hash" : BinData(0,"OURBFn1xm2E642Ftlh
                                 SIHv7SXoA="),
                        "keyId" : NumberLong("666814134357694
                                  8737")
                }
        }
} :
_getErrorWithCode@src/mongo/shell/utils.js:25:13
Mongo.prototype.getDBs@src/mongo/shell/mongo.js:67:1
shellHelper.show@src/mongo/shell/utils.js:876:19
shellHelper@src/mongo/shell/utils.js:766:15
@(shellhelp2):1:1
myrs:PRIMARY>
```

To resolve the issue, we need to authenticate to list databases as follows.

```
use admin
db.auth("subhashini","abc123")
show dbs;
```

Here is the output,

```
myrs:PRIMARY> use admin
switched to db admin
myrs:PRIMARY> db.auth("subhashini","abc123")
1
myrs:PRIMARY> show dbs;
admin   0.000GB
config  0.000GB
local   0.000GB
myrs:PRIMARY>
```

Recipe 8-3. Modifying Access for the Existing User

In this recipe, we are going to discuss how to modify access for the existing user.

Let us use the `superUserAdmin` user, which we created in Recipe 8-1.

Problem

We want to modify the access for `superUserAdmin` (created earlier).

Solution

Check the granted access (i.e., assigned roles and privileges) of the user, then revoke the permissions that are not needed or grant the new roles and privileges.

How It Works

Let's follow the steps in this section to modify the access of `superUserAdmin`.

Step 1: Connect to the Instance with User `superUserAdmin` and to `admin` Database

Use the following syntax.

```
mongo --username superUserAdmin --password 1234
--authenticationDatabase admin
```

Here is the output,

```
> mongo --username superUserAdmin --password 1234
--authenticationDatabase admin
MongoDB shell version v4.0.11
connecting to: mongodb://127.0.0.1:27017/?authSource=admin&gssa
piServiceName=mongodb
Implicit session: session { "id" : UUID("9f31da4d-ac50-438a-
8dd5-2ebd4fefadbb") }
MongoDB server version: 4.0.11
>
```

Step 2: Identify the User's Existing Roles

```
use <dbName>
db.getUser("<userName">)
```

Use the following commands to get the roles of `superUserAdmin` in the `admin` database.

```
> use admin
```

```
> db.getUser("superUserAdmin")
```

Here is the output,

```
> show dbs;
admin   0.000GB
config  0.000GB
local   0.000GB
> use admin
switched to db admin
> db.getUser("superUserAdmin")
{
        "_id" : "admin.superUserAdmin",
        "userId" : UUID("f109b62e-5138-487b-aa13-f183f977398c"),
        "user" : "superUserAdmin",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "userAdminAnyDatabase",
                        "db" : "admin"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
```

This output clearly shows the user `superUserAdmin` has `"role" : "userAdminAnyDatabase"`.

Note `userAdminAnyDatabase` is a built-in role that provides the ability to perform administration operations as `userAdmin` on all databases except `local` and `config`.

Step 3: Grant an Additional Role to the User

Use the following commands to assign the read role for the `local` database for `superUserAdmin`.

```
> use admin

> db.grantRolesToUser(
    "superUserAdmin",
    [
      { role: "read", db: "local" }
    ]
)
```

Repeat the preceding step to identify the roles and to verify the newly added role to the user `superUserAdmin`.
Here is the output,

```
> use admin
switched to db admin
> db.grantRolesToUser(
...     "superUserAdmin",
...     [
...       { role: "read", db: "local" }
...     ]
... )
> db.getUser("superUserAdmin")
```

```
{
        "_id" : "admin.superUserAdmin",
        "userId" : UUID("f109b62e-5138-487b-aa13-f183f977398c"),
        "user" : "superUserAdmin",
        "db" : "admin",
        "roles" : [
                {
                    "role" : "read", "db" : "local"
                },
                {
                   "role" : "userAdminAnyDatabase","db" : "admin"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1","SCRAM-SHA-256"
        ]
}
```

As you can observe, `"role" : "read"` is added to the `"db" : "local"` for the `superUserAdmin` user.

Step 4: Revoke an Existing Role from the User

Use this step to revoke the `"role":"userAdminAnyDatabase"` from the `superUserAdmin` user.

```
> use admin
> db.revokeRolesFromUser(
    "superUserAdmin",
    [
      { role: "userAdminAnyDatabase", db: "admin" }
    ]
)
```

Now check the existing roles by repeating the earlier step and verify that the role is removed.

```
> db.getUser("superUserAdmin")
```

Here is the output,

```
> db.getUser("superUserAdmin")
{
        "_id" : "admin.superUserAdmin",
        "userId" : UUID("f109b62e-5138-487b-aa13-f183f977398c"),
        "user" : "superUserAdmin",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "read","db" : "local"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
```

You can see that `"role" : "userAdminAnyDatabase"` is removed to the `"db" : "admin"` for the `superUserAdmin` user.

Recipe 8-4. Change the Password for the Existing User

In this recipe, we are going to discuss how to change the password for the `superUserAdmin` user.

Problem

You want to change the password for a specific user.

Solution

Use `changeUserPassword()` with the new password.

How It Works

Let's follow the steps in this section to change the password of a user.

Step 1: Connect to the Instance with User `superUserAdmin` and to the `admin` Database

```
mongo --username superUserAdmin --password 1234
--authenticationDatabase admin
```

Here is the output,

```
> mongo --username superUserAdmin --password 1234
--authenticationDatabase admin
MongoDB shell version v4.0.11
connecting to: mongodb://127.0.0.1:27017/?authSource=admin&
gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("9f31da4d-ac50-438a-
8dd5-2ebd4fefadbb") }
MongoDB server version: 4.0.11
>
```

Step 2: Change the Password

```
> db.changeUserPassword("superUserAdmin", "newpwd@123")
```

Now disconnect the instance and connect with the new password to ensure that the password has been changed.
Here is the output,

```
> exit
bye
C:\Program Files\MongoDB\Server\4.0\bin> mongo --username super
UserAdmin --password newpwd@123 --authenticationDatabase admin
MongoDB shell version v4.0.11
connecting to: mongodb://127.0.0.1:27017/?authSource=admin&
gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("9676d897-719d-4857-
9570-af7e982e10fe") }
MongoDB server version: 4.0.11
>
```

Recipe 8-5. Track the Activity of Users in a Cluster

In this recipe, we are going to discuss how to track the various activities of different users.

Problem

You want to track user activity.

Solution

Use the auditing feature.

Note The auditing feature is available in MongoDB Enterprise Edition only.

How It Works

Let's follow the steps in this section to enable the auditing feature and store the audit information in different output formats like writing audit events to the console, a JSON file, or a BSON file.

We can record the operations shown in Figure 8-6 using the auditing feature

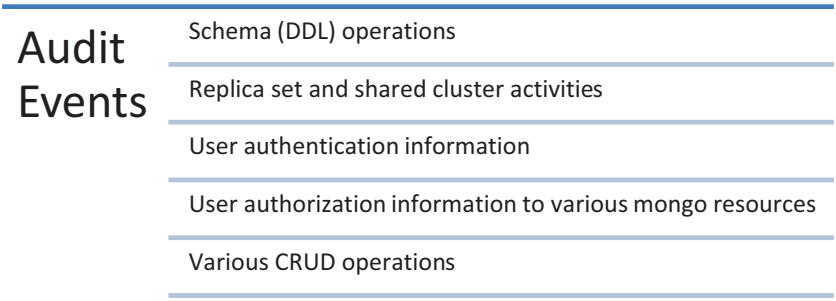


Figure 8-6. *Audit events*

Step 1: Enable and Configure Audit Events

We can use `auditDestination` to enable the auditing feature and specify the output location of audit information.

Use the `syslog` output to print the audit events to syslog.

```
> mongod --dbpath data/testDB --auditDestination syslog
```

Note This `syslog` option is not available for Windows.

Use the `console` output to print the audit events to the console.

```
> mongod --dbpath data/testDB --auditDestination console
```

Use the `auditFormat JSON` output to print the audit events to a separate JSON file in any path.

```
> mongod --dbpath data/testDB --auditDestination file
--auditFormat JSON --auditPath data/testDB/auditLog.json
```

Step 2: Create an Audit Filter

Audit filters help us to enable audit features for only certain operations. Let us say, for example, that we want to create an audit filter to log only the audit events related to authenticatation operations that are performed on the `local` database. Use the following filter.

```
{ atype: "authenticate", "param.db": "test" }
```

Here, `atype` refers to the action type and `param.db` refers to the monitoring database.

Use the following command to set the filter while enabling the audit feature.

```
> mongod --dbpath data/testDB --auth --auditDestination file
--auditFilter '{atype: "authenticate", "param.db": "local"}'
--auditFormat JSON --auditPath data/testDB/auditLog.json
```

Recipe 8-6. Encryption

In this recipe, we are going to discuss how to encrypt the data in MongoDB.

Problem

You want to encrypt the data stored or moved into MongoDB.

Solution

Use encryption at the file level and the whole database level.

Note The auditing feature is available in MongoDB Enterprise Edition only.

How It Works

Let's follow the steps in this section to encrypt the data stored and decrypt while reading.

Step 1: Understanding Different Encryption Options

MongoDB offers encryption at two levels:

- Encrypting data in motion.
- Encrypting data in rest.

Encrypting Data in Motion

MongoDB Enterprise Edition supports Transport Layer Security (TLS) and Secure Sockets Layer (SSL) encryption techniques to secure the data being sent or received over the networks.

Encrypting Data in Rest

MongoDB Enterprise Edition supports a native storage-based symmetric key encryption technique to secure the data available on the storage system. It also provides Transparent Data Encryption (TDE), which is used to encrypt the whole database.

Step 2: Encrypting the Data at Rest Using Local Key Management

Create a 16- or 32-character string base64-encoded key file using any appropriate method. We use the `openssl` method as shown here.

```
> openssl rand -base64 32 > mongo-encryption-file
```

Here is the output,

```
C:\Users\DHARANI\Downloads\openssl-win>
 openssl rand -base64 32 > mongo-encryption-file
```

Now we can see in Figure 8-7 the encryption file created in the same directory.

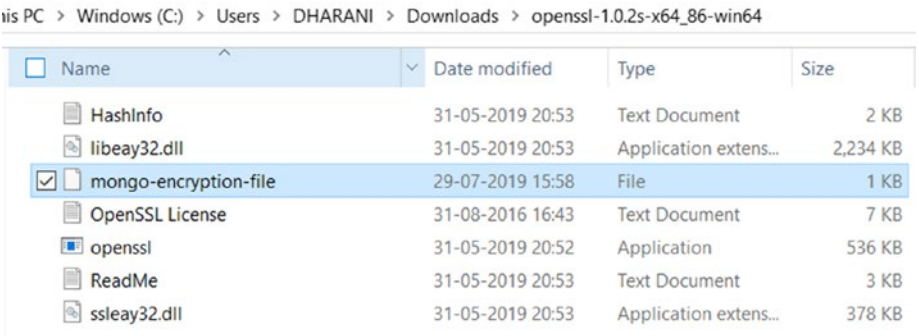


Figure 8-7. *The new encryption file*

Now to use the encryption file to encrypt the data in the storage engine, start MongoDB with the following options.

```
--enableEncryption
--encryptionKeyFile <path to encryption keyfile>
```

```
> mongod --enableEncryption --encryptionKeyFile  " C:\Program
Files\MongoDB\Server\4.0\mongo-encryption-file"
```

The output is shown in Figure 8-8.

```
C:\Program Files\MongoDB\Server\4.0\bin>mongod
--enableEncryption --encryptionKeyFile  "C:\Program
Files\MongoDB\Server\4.0\mongo-encryption-file"
CONTROL  [main] Automatically disabling TLS 1.0, to
force-enable TLS 1.0 specify --sslDisabledProtocols 'none'
CONTROL  [initandlisten] MongoDB starting : pid=13576
port=27017 dbpath=C:\data\db\ 64-bit host=DESKTOP-MEISTBV
STORAGE  [initandlisten] Encryption key manager
initialized with key file: C:\Program
Files\MongoDB\Server\4.0\mongo-encryption-file
CONTROL  [initandlisten] **          server with --bind_ip
127.0.0.1 to disable this warning.
CONTROL  [initandlisten]
STORAGE  [initandlisten] createCollection:
admin.system.version with provided UUID:
1d5bf83b-7b47-4496-9985-1886c70f16cd
NETWORK  [initandlisten] waiting for connections on port
27017
```

Figure 8-8. *The encryption output*

We can check for the following line in the log to ensure the encryption key manager is successfully initialized with the specified key file.

```
2019-07-27T16:07:51.500+0530 I STORAGE  [initandlisten]
Encryption key manager initialized with key file: C:\Program
Files\MongoDB\Server\4.0\mongo-encryption-file
```

Note Encrypting the data using the local key file is not recommended, as the secure management of the local key file is critical.

Index

S. Chellappan and D. Ganesan, *MongoDB Recipes*,
https://doi.org/10.1007/978-1-4842-4891-1

E

F, G

H

I

J, K

L

M

N, O, P, Q

R

S

T, U

V, W

X, Y, Z