

Managing Large Databases



Using MongoDB

Managing Large Databases Using MongoDB

© 2023 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic, or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

Edition 1 – 2023



Onlinevarsity



**LEARN
ANYWHERE
ANYTIME**

Preface

MongoDB is a cross-platform document-oriented database system that uses a flexible schema to store data as JSON-like documents. MongoDB offers high performance, scalability, and availability for a wide range of applications. MongoDB supports various features such as indexing, aggregation, text search, geospatial queries, replication, sharding, and transactions. MongoDB is designed to meet the requirements of modern Web and mobile development, as well as big data and analytics. This module will introduce you to MongoDB which is widely used to create and maintain document-based databases.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team



**MANY
COURSES
ONE
PLATFORM**



Onlinevarsity App for Android devices

Download from **Google Play Store**

Table of Contents

Sessions

- Session 1 – Introduction to MongoDB**
- Session 2 – MongoDB Databases**
- Session 3 – MongoDB Operators**
- Session 4 – Aggregation Pipeline**
- Session 5 – Database Commands**
- Session 6 – MongoDB Shell Methods**
- Session 7 – MongoDB Indexing**
- Session 8 – MongoDB Replication and Sharding**
- Session 9 – Transaction Management**
- Session 10 – MongoDB Tools**
- Session 11 – MongoDB Cloud**
- Session 12 – MongoDB Database Connectivity With Python**
- Appendix – Case Studies**



SESSION 1

INTRODUCTION TO MONGODB

Learning Objectives

In this session, students will learn to:

- Explain various types of databases
- Describe how data is stored in the MongoDB Server
- Explain the installation of MongoDB Server and MongoDB Shell

Databases have been in use for quite some time now. Most of the companies use structured databases, such as SQL Server and Oracle, to store information related to their products, customers, sales, employees, and so on. These databases grow continuously with more and more new information. Companies must invest in the required hardware to store the ever-growing data. As the databases explode, some of the existing database systems become very expensive to upgrade and maintain. In addition, the performance of these databases is affected due to large volume of data.

To overcome the issues that structured databases posed with increasing data, companies are now opting for NoSQL databases. Especially companies that host social media apps or gaming apps, which encounter an ever-increasing customer or player base. MongoDB is one of the popular NoSQL databases.

This session will provide an overview of the types of databases and the prevalence of NoSQL databases. This session also explains the architecture of MongoDB and illustrates the steps to install the MongoDB server and shell.

1.1 Databases

Data refers to any information and a database is a collection of data. An example of a database is a telephone directory or a product catalog. Depending on the way the data is stored, there are different types of databases. These are:

Hierarchical	Similar to a family tree, this type of database stores data in the form of parent-child nodes. Example: Windows Registry
Relational	This type of database stores data in the form of rows and columns that together form a table (relation). Example: MySQL, Oracle
Object-oriented	This type of database stores data in the form of objects similar to the objects used in an object-oriented programming language. Example: PostgreSQL
Network	This database stores data as multiple children and parent nodes that results in complex database structures. Example: RDM Server
NoSQL	This database does not use a relational model and is used for analyzing large sets of distributed data. Example: MongoDB

Before getting into the details of MongoDB, let us understand the difference between relational databases, which are quite popular, and NoSQL databases, which are becoming more and more popular.

Table 1.1 distinguishes the features of relational and NoSQL databases.

Attribute	Relational Database	NoSQL Database
Data storage	It stores data within rows and columns in tables.	It stores data in various formats, such as documents, graphs, or key-value pairs.
Schema	It requires every table in the database to have a defined schema and the schema cannot be altered after it is created.	It does not require data to have a defined schema. It supports dynamic schema, which allows alterations to the storage structure as required.
Scaling	To support increase in data volume and database users, the hardware of the server must be updated with RAM or processors, or a completely new server machine must be installed. Adding more resources to a server is referred to as vertical scaling.	To support increase in data volume and database users, multiple computers or servers with minimal configuration can be installed. Adding more servers that contain same set of databases with different sets of information is referred to as horizontal scaling.
Performance	The information related to a single entity is stored in multiple tables, based on normalization. Therefore, if data is to be retrieved for a single entity, multiple tables must be iterated through. This results in a decrease in performance, especially if the volume of data is large.	The information related to a single entity is stored together in a single location. Therefore, if data is to be retrieved for a single entity, a simple query can be used. This improves the performance of the database.
Consistency	Relational DataBase Management System (RDBMS) follows the Atomicity, Consistency, Isolation, and Durability (ACID) properties.	NoSQL Database does not follow the ACID properties.

Attribute	Relational Database	NoSQL Database
Application	RDBMS is a best fit when data items must be related and stored in a rules-based, consistent manner. Examples include e-commerce transactions and stock tracking.	NoSQL database is a best fit for applications that require huge data volumes and low latency. Examples include online gaming, social media, and shopping apps.

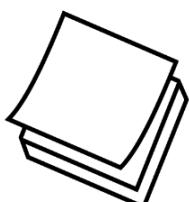
Table 1.1: Relational Databases vs. NoSQL Databases

1.2 Overview of MongoDB

Consider a social media mobile app where a movie star with a million followers has posted his latest movie stills. Several thousands of his followers are commenting on the movie stills and the movie star is also responding to these comments. Comments and responses are a mix of text messages, animations, pictures, and emojis. If this social media app is placed on RDBMS, there would be one table each to store:

- the details of the movie star,
- the details of the followers,
- the posts put up by the movie star, and
- the comments and responses.

All these tables will be related to each other based on one or more columns. In this case, the time taken for reading/writing such large, changing, and varied data to the server will be really long. Consequently, the user experience of the mobile app would not be very good. However, a NoSQL database would perfectly fit in such a scenario because all the data can be stored together. This data storage will speed up data retrieval leading to good user experience. One example of a NoSQL database is MongoDB.

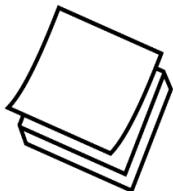


Depending on how data is stored in the database, there are four types of NoSQL databases:

- Document databases: Store data as JSON, BSON, or XML documents
- Key-value databases: Store data with a label or attribute and the value
- Wide-column databases: Store data in columns

- Graph databases: Store each data element a node and relationship between the data elements as links or relationships
-

MongoDB is a NoSQL database program that stores information in formats similar to JavaScript Object Notation (JSON) called the Binary JavaScript Object Notation (BSON). MongoDB is an open-source software which works with large sets of distributed data. It provides high performance, high availability, and automatic scaling.



The JSON format is a standard human-readable format used to transmit data in the form of key-value pairs. BSON is used to store and access documents transmitted using JSON format. MongoDB stores documents in BSON format.

1.2.1 MongoDB Architecture

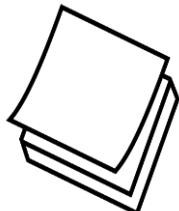
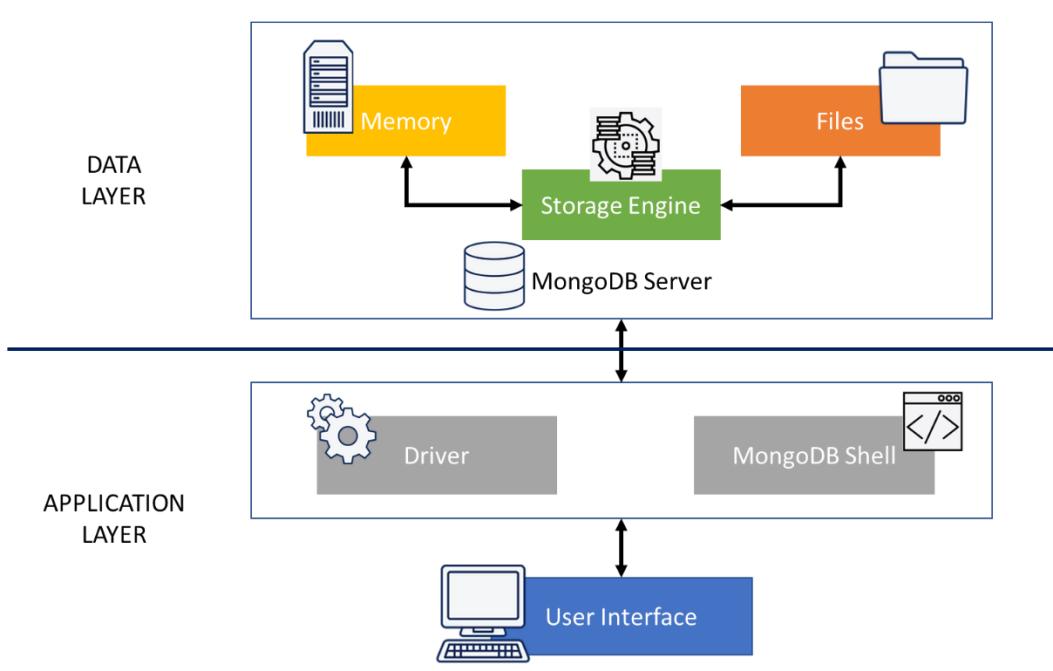
MongoDB operates in two layers: the application layer and the data layer.

The application layer is made up of:

- User interface that the user uses to interact with the database by using a mobile app (Android or iOS) or directly from the Web,
- MongoDB driver that helps to make the connection between the user interface and the MongoDB server, and
- MongoDB Shell which is the command-line interface that users can use to interact with the MongoDB server.

The data layer consists of the MongoDB server that receives the queries from the application layer and sends them on to the storage engine. The storage engine then reads or writes the data to the memory or the file. The storage engine determines how data is stored and the amount of data stored on the disk (file) and memory.

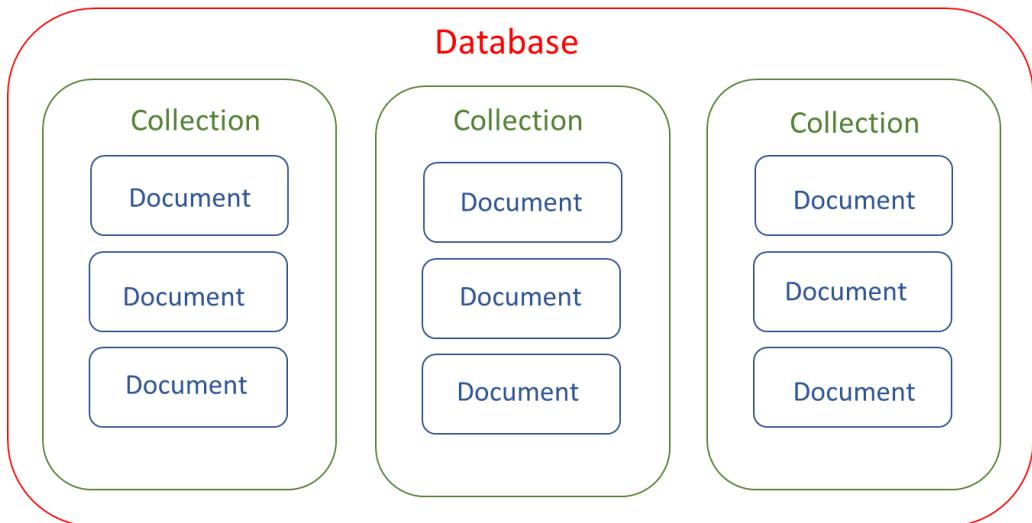
Thus, the vital components of MongoDB that are necessary for its function are:



- MongoDB supports drivers for C, C++, .Net, Go, C#, Python, Node.js, and so on.
- WiredTiger is the default MongoDB storage engine. However, MongoDB also uses other storage engines, such as the in-memory storage engine, MMAPv1 storage engine, and the encrypted storage engine.

1.2.2 Database, Collection, and Document

Data is stored in documents (rows and columns in RDBMS) on the MongoDB Server. Multiple documents together make up a collection (tables in RDBMS). Multiple collections make up a database.



MongoDB stores data in the document as BSON documents. BSON is a binary representation of JSON documents, although BSON supports more data types than JSON. The BSON format will look as follows:

```
{
    field1: value1,
    field2: value2,
    field3: value3,
    ...
    fieldN: valueN
}
```

A sample document with values in the fields will look as follows:

```
{
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Larissa", last: "Smith" },
    birth: new Date('Jun 23, 2015'),
    grade: 6,
    marks: [{sub: "English",
              score: 89},
             {sub: "Math",
              score: 75},
             {sub: "Science",
              score: 92},
             {sub: "Social Studies"
              score: 70}],
}
```

The sample document shows key-value pairs which form the basic unit of data in MongoDB.

1.2.3 MongoDB Editions

MongoDB is available both as a server and as a service on the cloud. As a server, MongoDB is available as Community Server and Enterprise Server. While Community Server is a free edition, Enterprise Server is a commercial edition that is available on subscription. While the core server features are the same across the two editions, the Enterprise edition offers additional operational and management capabilities, an in-memory storage engine, and advanced security features.

As a cloud service, MongoDB is available as MongoDB Atlas. It offers Data as a Service (DaaS) hosting data in the cloud and provides all the features of cloud computing including high availability and scalability.

MongoDB Atlas includes products such as Atlas Database, Atlas Search, Atlas Data Federation, Atlas Charts, Atlas App Services, Atlas Device Sync, Atlas Data Lake, and so on.

With MongoDB Atlas, users can:

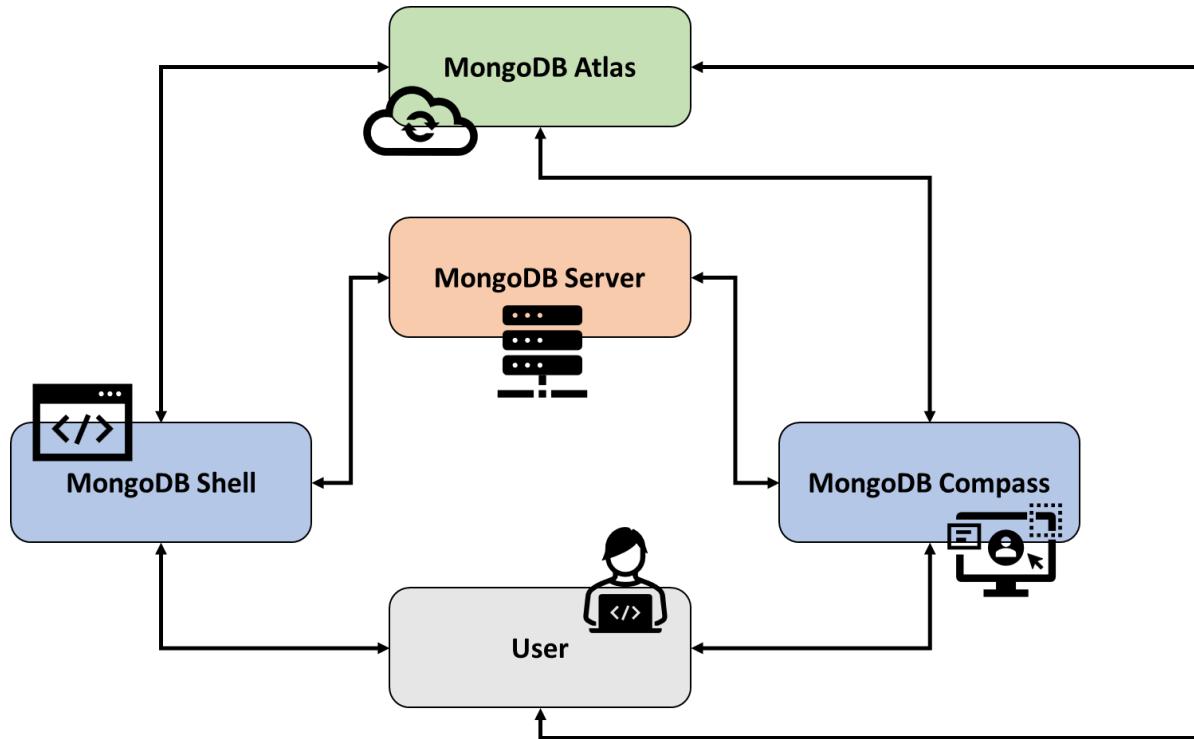
- Save time and cost. MongoDB Atlas includes a fully managed database service that provides access to the database resources as required.
- Use a common API to serve any type of database requests.
- Perform multi-region deployments. Atlas Database is available in more than 80 regions across AWS, Google Cloud, and Azure.
- Focus on building rather than managing. Atlas provides automated infrastructure operations that guarantee availability, scalability, and security compliance.
- Work with data as code. In MongoDB, documents map directly to the objects in the code. As a result, data of any structure can be stored, and the schema can be modified when new features are added to the application.



MongoDB Atlas allows deployment of databases to any cloud provider rather than providing a single option.

The MongoDB Shell is a standalone command-line product that the user can use to connect to the server or the service. MongoDB Compass is Graphical User Interface (GUI) that the users can use to connect to the server or to the service.

Users can also directly access the Atlas platform and work on it without using MongoDB Shell or MongoDB Compass.



1.3 Installing MongoDB Community Server

This course is based on the MongoDB 6.0 Community edition.

To install the MongoDB 6.0 Community edition on Windows systems:

1. Open a browser and navigate to the URL:
<https://www.mongodb.com/try/download/community>
The MongoDB Community Server Download page opens.
2. On this page, scroll down and ensure that:
 - In the **Version** drop-down, **6.0.5(current)** is selected.
 - In the **Platform** drop-down, **Windows** is selected.
 - In the **Package** drop-down, **msi** is selected.
3. Click **Download**.
4. After the download is complete, run the installer.

The MongoDB 6.0.5 2008R2Plus SSL (64-bit) Setup wizard opens as shown in Figure 1.1.

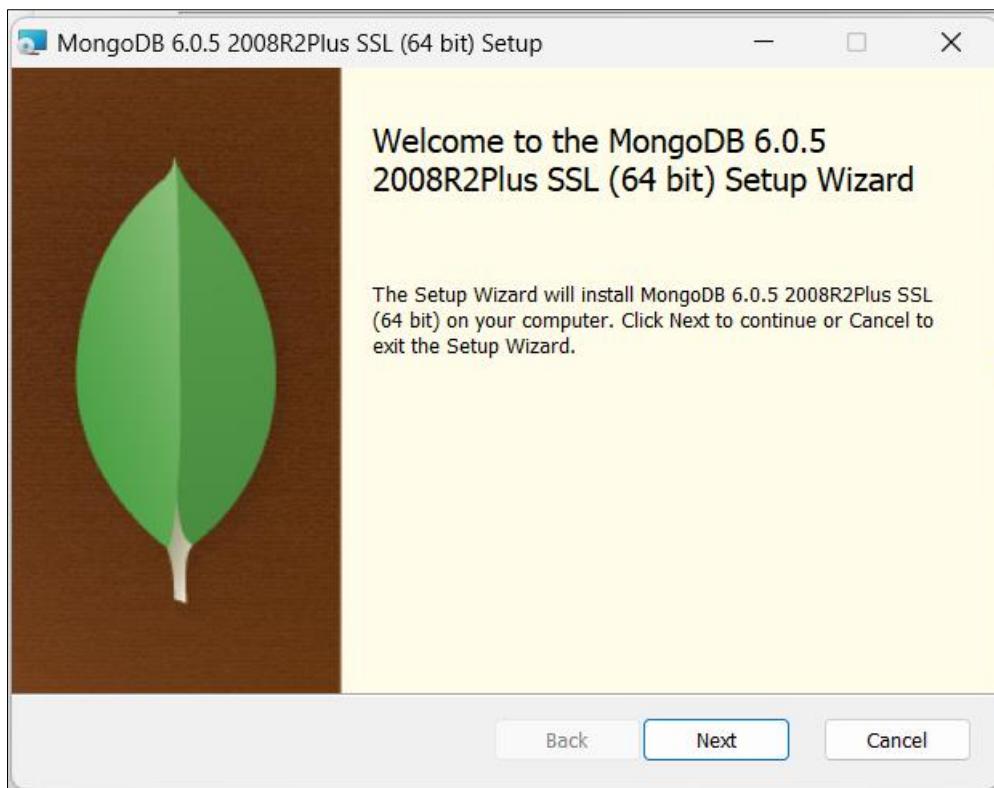


Figure 1.1: Welcome Page of the MongoDB Installation Wizard

5. Click **Next**.

The End-User License Agreement page of the wizard opens as shown in Figure 1.2.

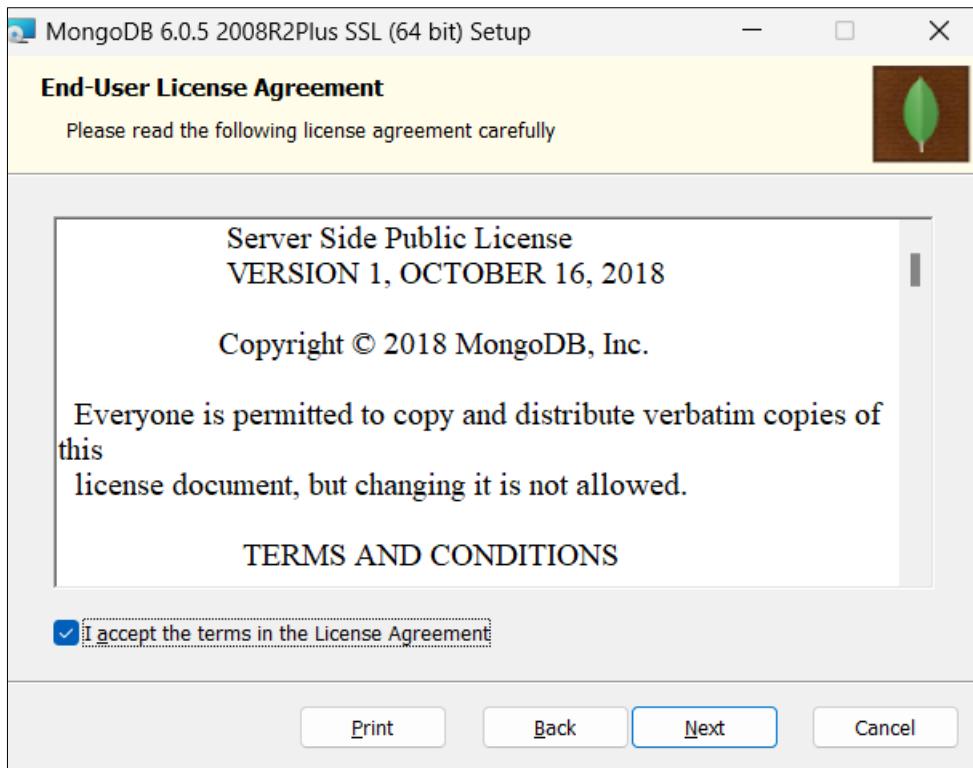


Figure 1.2: End-User License Agreement Page

6. To proceed, select the **I accept the terms in the License Agreement** check box.
7. Click **Next**.

The Choose Setup Type page of the wizard opens as shown in Figure 1.3.

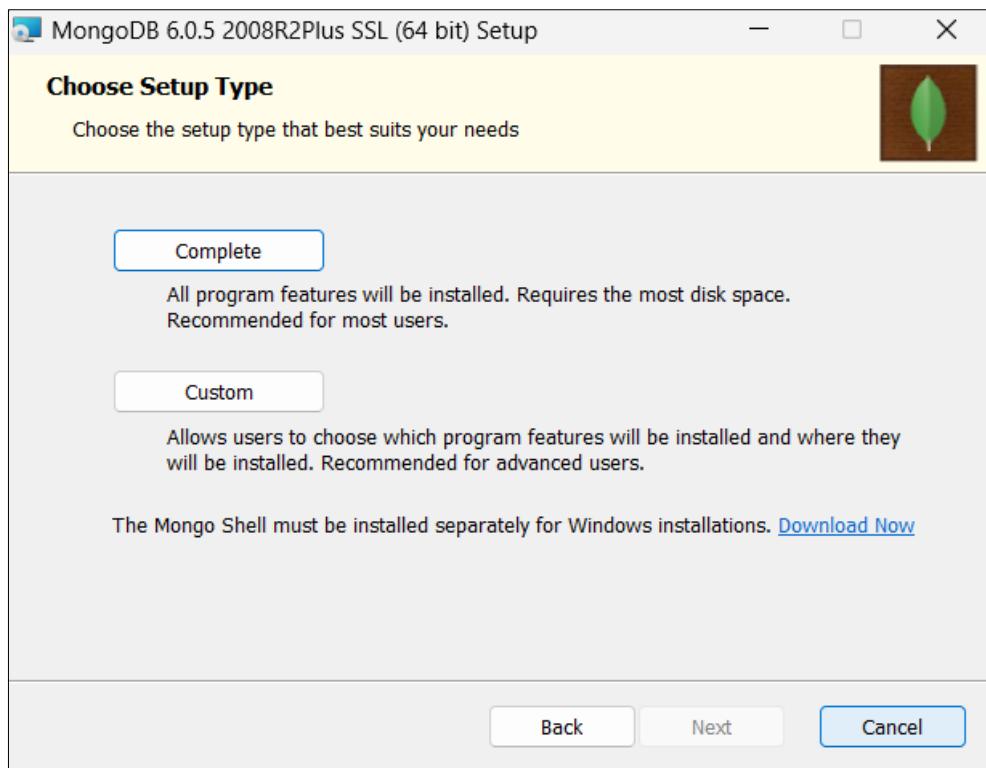


Figure1.3: Choose Setup Type Page

8. Click **Complete**.

The Service Configuration page of the wizard opens as shown in Figure 1.4.

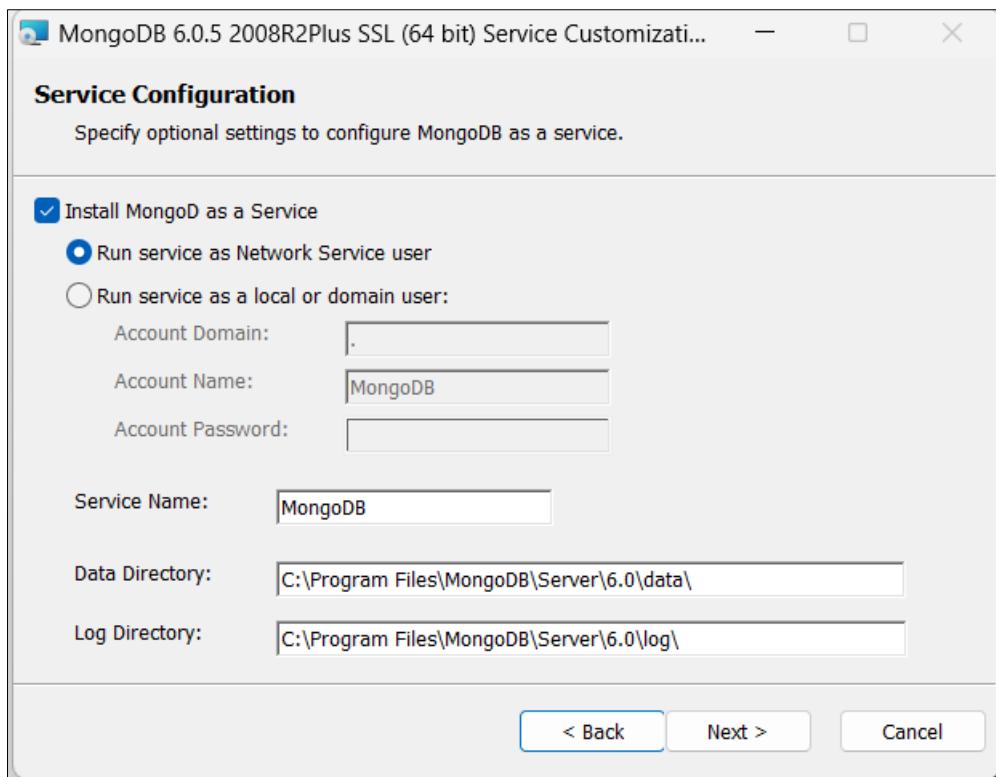


Figure 1.4: Service Configuration Page

9. Click **Next**.

The Install MongoDB Compass page of the wizard opens as shown in Figure 1.5.

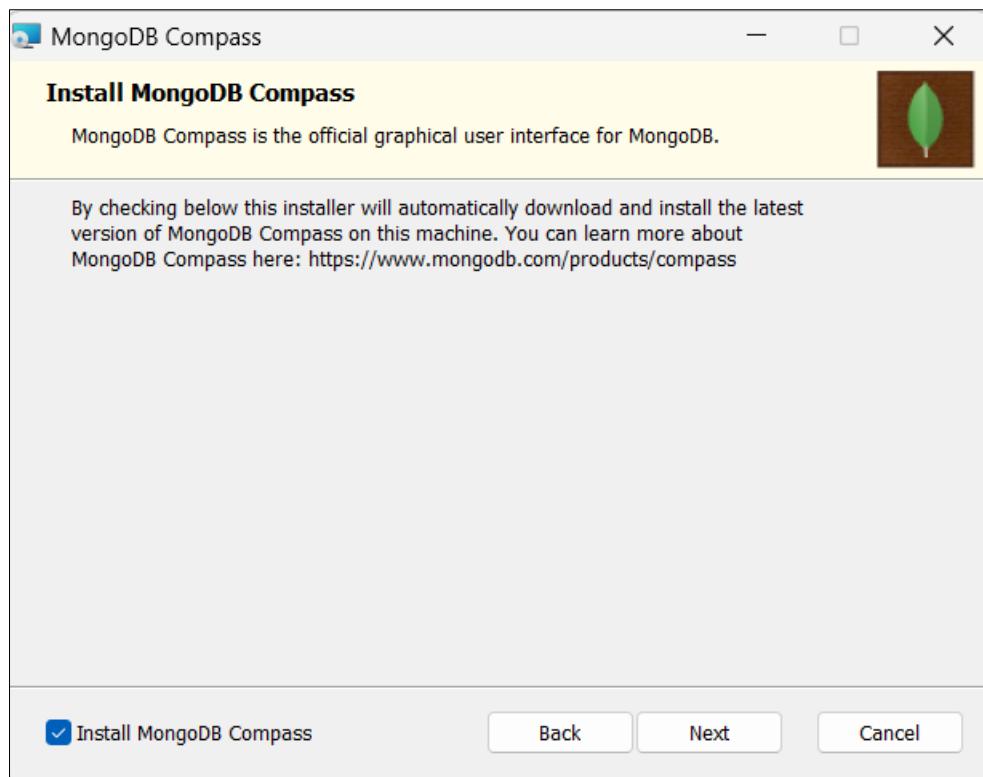


Figure 1.5: Install MongoDB Compass Page

10. Ensure that the **Install MongoDB Compass** check box is selected.
11. Click **Next**.
The Ready to install MongoDB 6.0.5 2008R2Plus SSL (64-bit) page of the wizard opens as shown in Figure 1.6.

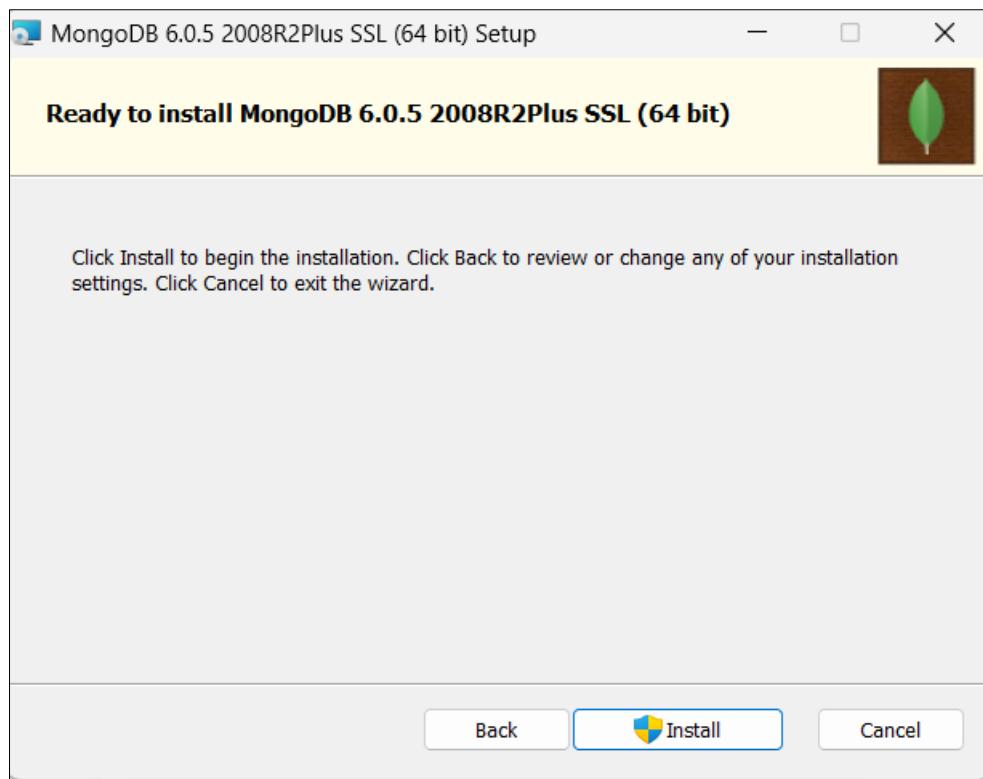


Figure 1.6: Screenshot of the Ready to Install Page

12. Click **Install**.

The Installing MongoDB 6.0.5 2008R2Plus SSL (64-bit) page of the wizard opens as shown in Figure 1.7.

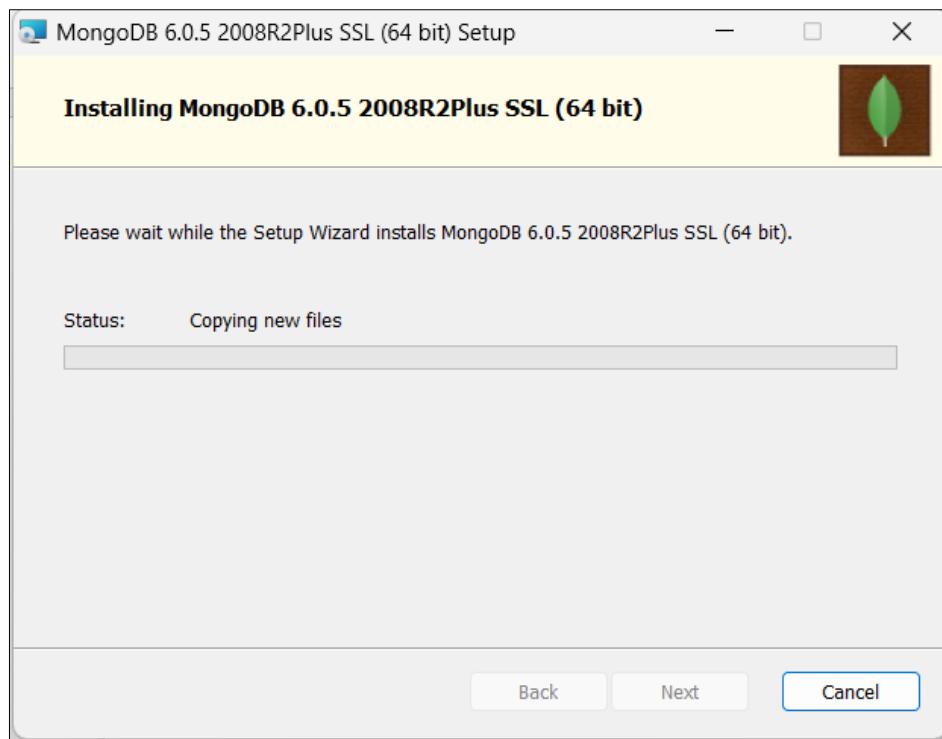


Figure 1.7: Installation in Progress Page

The Completed the MongoDB 6.0.5 2008R2Plus SSL (64-bit) Setup Wizard page of the wizard opens as shown in Figure 1.8.

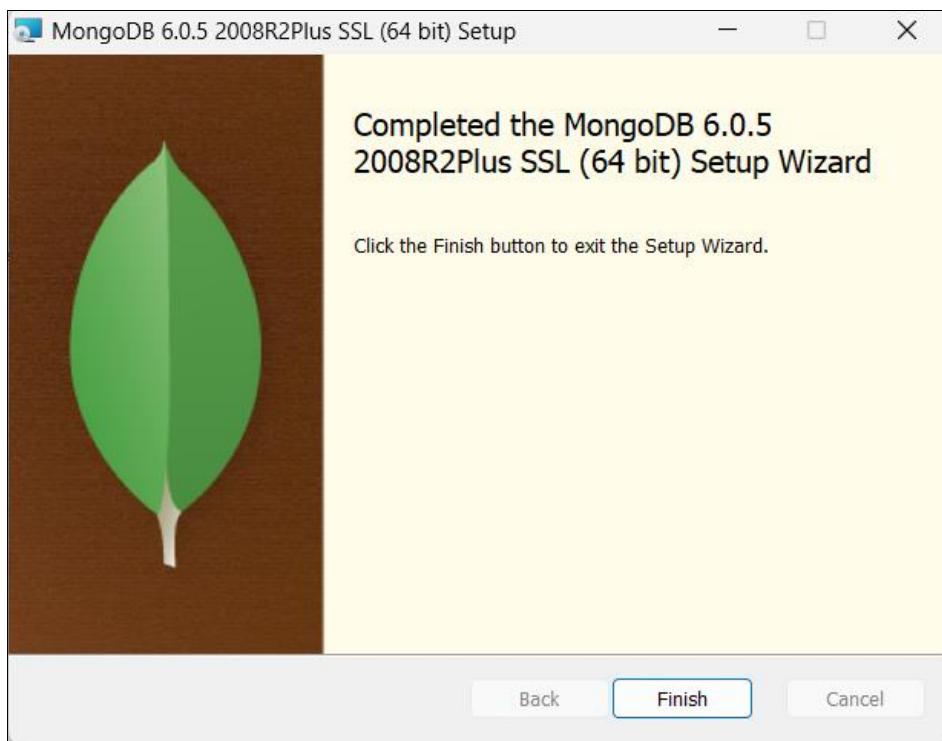


Figure 1.8: Installation Completion Page

13. Click **Finish**.



MongoDB Compass is also installed along with MongoDB Community Server. The MongoDB Compass window opens by default. Close this window.

1.4 Installing MongoDB Shell

After installing the MongoDB Server, install the MongoDB Shell.

To install the MongoDB Shell:

1. Open a browser and navigate to the URL:

<https://www.mongodb.com/try/download/shell>

The MongoDB Shell Download page opens as shown in Figure 1.9.

The screenshot shows a dropdown menu for selecting MongoDB Shell components. The 'Version' dropdown is set to '1.8.2'. The 'Platform' dropdown is set to 'Windows 64-bit (8.1+) (MSI)'. The 'Package' dropdown is set to 'msi'. Below these dropdowns, there is a search bar containing the text 'msi'.

Figure 1.9: Version, Platform, and Package Selection

2. On this page, scroll down and ensure that:
 - In the **Version** drop-down, **1.8.2** is selected.
 - In the **Platform** drop-down, **Windows 64-bit (8.1+) (MSI)** is selected.
 - In the **Package** drop-down, **msi** is selected.
3. Click **Download**.
4. After the download is complete, run the installer.

The MongoDB Shell Setup wizard opens as shown in Figure 1.10.

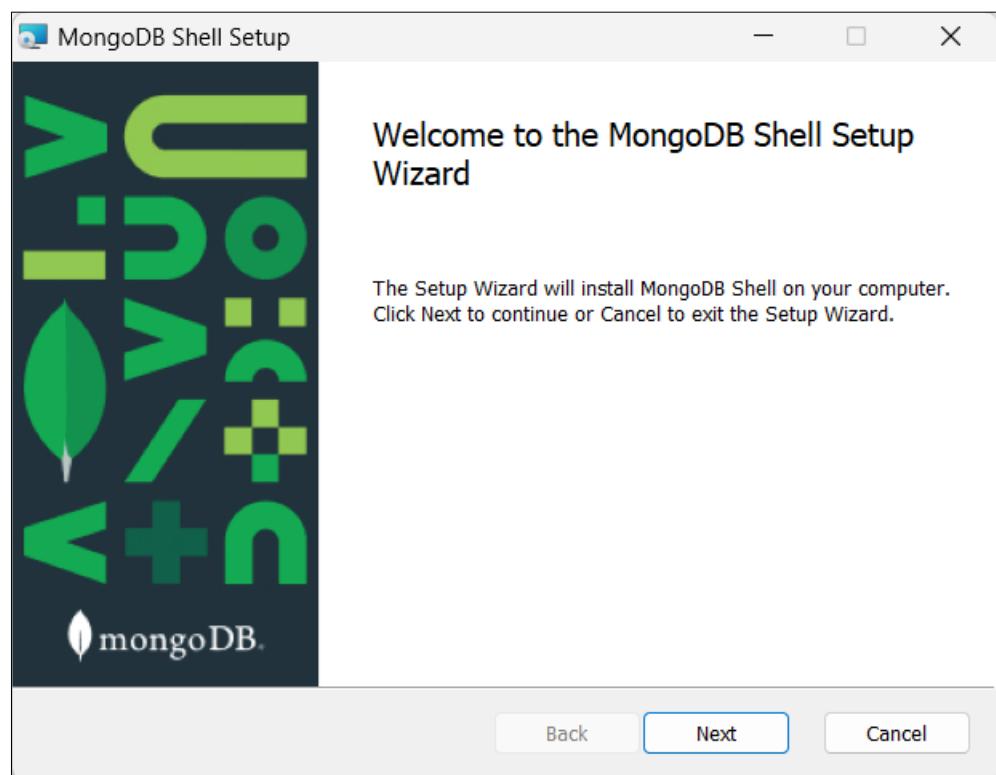


Figure 1.10: Screenshot of MongoDB Shell Setup Wizard

5. Click **Next**.

The Destination Folder page of the wizard opens as shown in Figure 1.11.

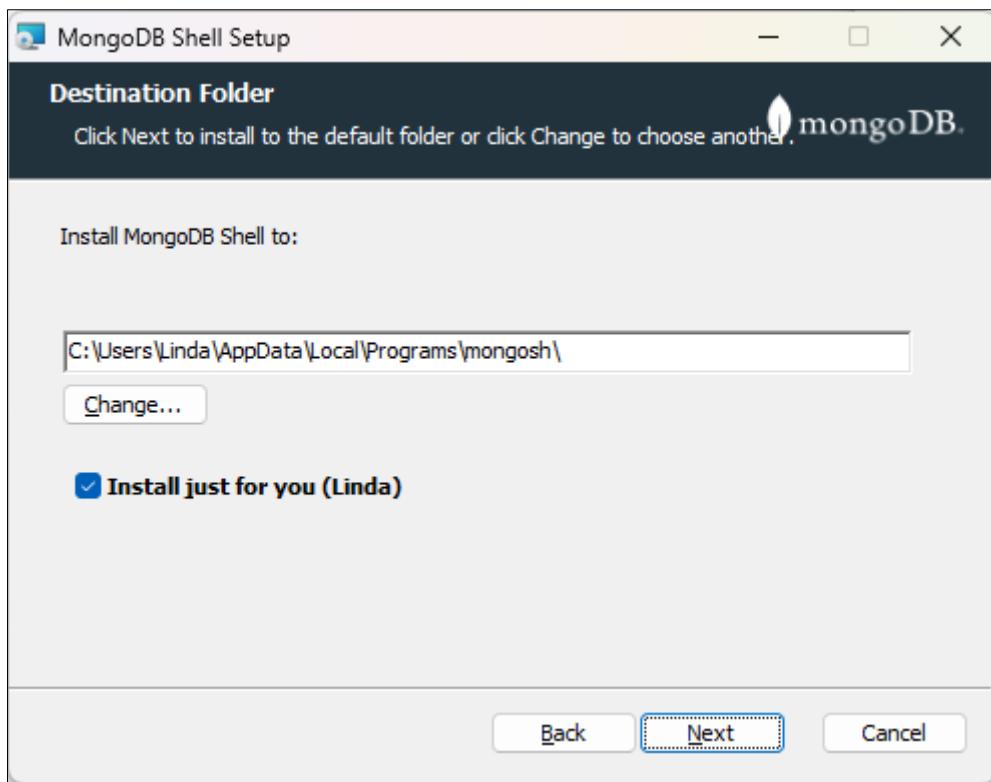


Figure 1.11: Destination Folder Page

6. Check the folder path and click **Next**.

The Ready to install MongoDB Shell page of the wizard opens as shown in Figure 1.12.

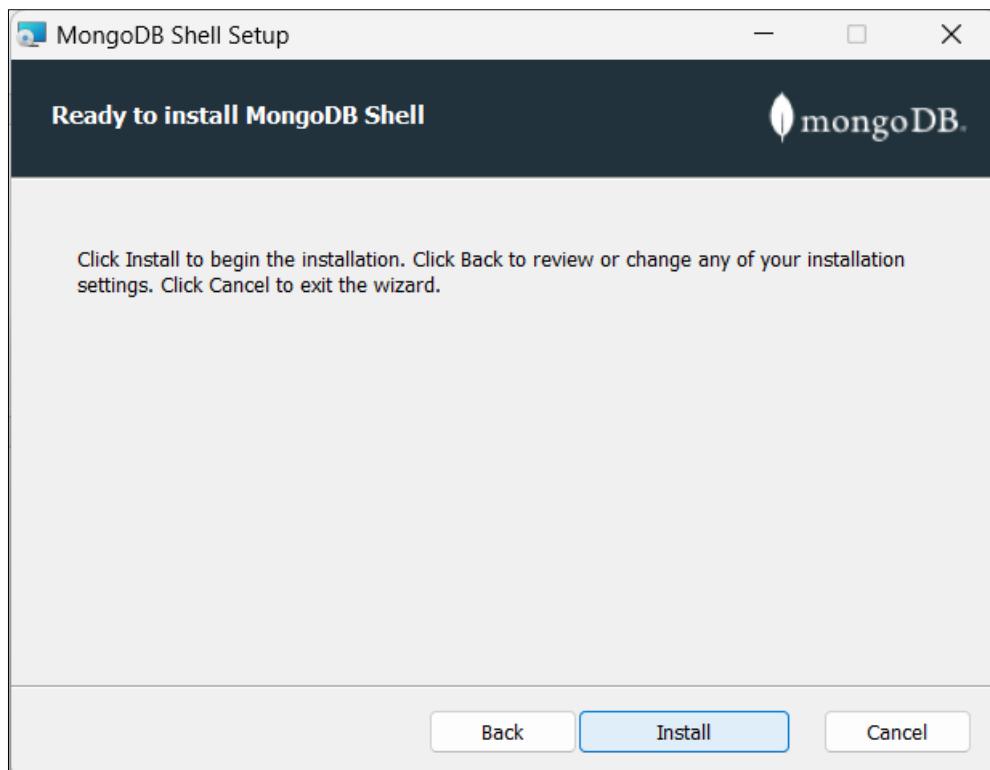


Figure 1.12: Ready to Install MongoDB Shell Page

7. Click **Install**.

The MongoDB Shell Setup page of the wizard opens as shown in Figure 1.13.

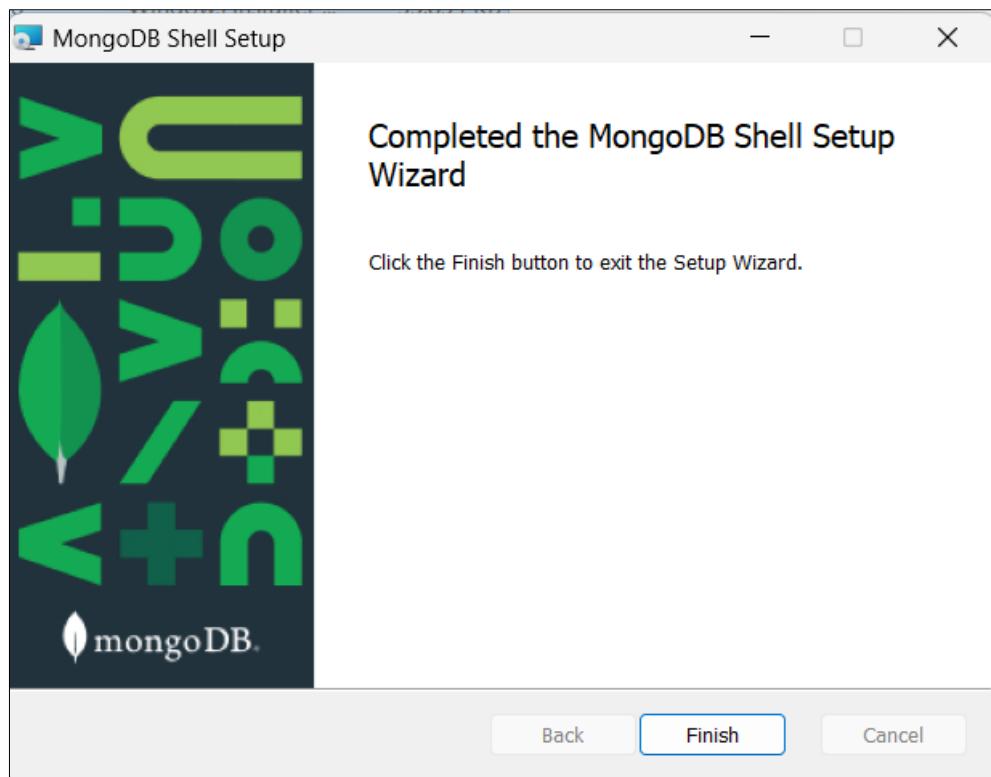


Figure 1.13: Screenshot of the MongoDB Shell Setup Page

8. Click **Finish**.

1.5 Setting up the Environment

To set up the environment:

1. Open a Windows command prompt (cmd.exe) as an **Administrator**.
2. Create the data directory where MongoDB will store all the data, as shown in Figure 1.14.

```
C:\>md "\data\db"  
C:\>
```

Figure 1.14: Creating Directories



MongoDB listens for connections from clients only on port 27017, by default. The default storage path for data is /data/db directory.

To start the **MongoDB** database, run the command:

```
"C:\Program Files\MongoDB\Server\6.0\bin\mongod.exe" --dbpath="c:\data\db"
```

The command is executed as shown in Figure 1.15.

```
C:\>"C:\Program Files\MongoDB\Server\6.0\bin\mongod.exe" --dbpath="c:\data\db"
[{"t": {"$date": "2023-05-01T13:09:48.839+05:30"}, "s": "I", "c": "NETWORK", "id": 4915701, "ctx": "thread1", "msg": "Initialized wire specification", "attr": {"spec": {"incomingExternalClient": {"minWireVersion": 0, "maxWireVersion": 17}, "incomingInternalClient": {"minWireVersion": 0, "maxWireVersion": 17}, "outgoing": {"minWireVersion": 6, "maxWireVersion": 17}, "isInternalClient": true}}}, {"t": {"$date": "2023-05-01T13:09:50.049+05:30"}, "s": "I", "c": "CONTROL", "id": 23285, "ctx": "thread1", "msg": "Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}, {"t": {"$date": "2023-05-01T13:09:50.053+05:30"}, "s": "I", "c": "NETWORK", "id": 4648602, "ctx": "thread1", "msg": "Implicit TCP FastOpen in use."}, {"t": {"$date": "2023-05-01T13:09:50.069+05:30"}, "s": "I", "c": "REPL", "id": 5123008, "ctx": "thread1", "msg": "Successfully registered PrimaryOnlyService", "attr": {"service": "TenantMigrationDonorService", "namespace": "config.tenantMigrationDonors"}}, {"t": {"$date": "2023-05-01T13:09:50.069+05:30"}, "s": "I", "c": "REPL", "id": 5123008, "ctx": "thread1", "msg": "Successfully registered PrimaryOnlyService", "attr": {"service": "TenantMigrationRecipientService", "namespace": "config.tenantMigrationRecipients"}}
```

Figure 1.15: Starting the MongoDB Database

The --dbpath option shows the database directory. If the MongoDB database server is running correctly, the message: **waiting for connections** is displayed as shown in Figure 1.16.

```
[{"t": {"$date": "2023-05-15T17:39:03.517+05:30"}, "s": "I", "c": "NETWORK", "id": 23015, "ctx": "listener", "msg": "Listening on", "attr": {"address": "127.0.0.1"}}, {"t": {"$date": "2023-05-15T17:39:03.517+05:30"}, "s": "I", "c": "NETWORK", "id": 23016, "ctx": "listener", "msg": "Waiting for connections", "attr": {"port": 27017, "ssl": "off"}}]
```

Figure 1.16: Waiting for Connections Message

Now, the MongoDB instance mongod is successfully running.



Add C:\Program Files\MongoDB\Server\6.0\bin to the system path. This allows to run the mongod instance instantly without having to navigate to this folder each time and then run the instance.

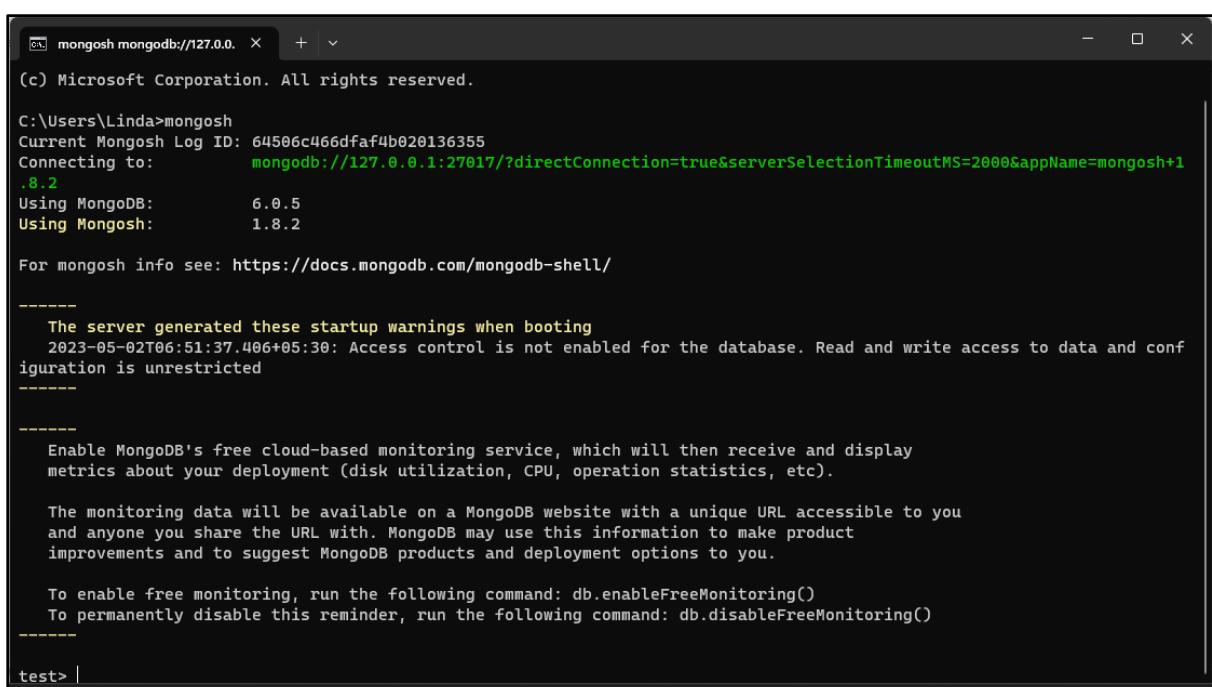
1.6 Connecting MongoDB Server and MongoDB Shell

To connect the MongoDB Shell with MongoDB Server:

1. Open another Windows command prompt/interpreter.
2. To connect the MongoDB Shell to a MongoDB deployment running on **localhost** with the default port 27017, run the command:

```
mongosh
```

The command is executed as shown in Figure 1.17.



A screenshot of a Windows command prompt window titled "mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.2". The window shows the following text:

```
(c) Microsoft Corporation. All rights reserved.

C:\Users\Linda>mongosh
Current Mongosh Log ID: 64506c466dfaf4b020136355
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.2
Using MongoDB:      6.0.5
Using Mongosh:     1.8.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-05-02T06:51:37.406+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

-----
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
-----
```

test> |

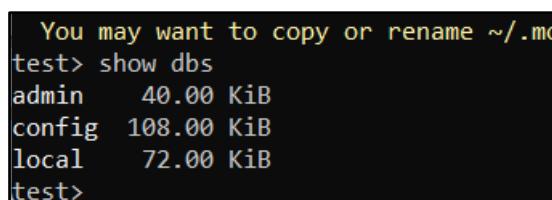
Figure 1.17: Output of the Mongosh Command

MongoDB Shell is now successfully connected to the MongoDB server.

3. To check the connection, run the command:

```
show dbs
```

List of existing databases is displayed as shown in Figure 1.18



```
You may want to copy or rename ~/mc
test> show dbs
admin   40.00 KiB
config  108.00 KiB
local   72.00 KiB
test>
```

Figure 1.18: Output of the show dbs Command

1.7 Summary

- A NoSQL database is a non-relational data that can support huge volumes of unstructured data.
- MongoDB is an open-source NoSQL database that stores data in the BSON format.
- The drivers, shell, and storage engine are vital elements for MongoDB to function.
- MongoDB stores data in the form of databases, collections, and documents.
- MongoDB works in two layers: application layer and data layer.
- Mongo DB Compass is the GUI for interacting with the MongoDB Server.
- MongoDB Atlas is the product MongoDB offers to compute on the cloud.
- To set up the MongoDB environment, first, install the MongoDB Server, then install the MongoDB Shell, and then use mongosh to connect the server to the shell.

Test Your Knowledge

1. Which of the following options is the data storage model of NoSQL databases?
 - a. Document
 - b. Tables with fixed rows and columns
 - c. Key-value
 - d. Graph

2. Which of the following statements are not true about NoSQL databases?
 - a. It has flexible schema.
 - b. It supports Horizontal scaling.
 - c. The data architectures offered by NoSQL databases are more versatile than the data architectures offered by the relational database tables.
 - d. It offers only read scalability.

3. Document database stores data in which of the following format?
 - a. JSON
 - b. BSON
 - c. Google Docs
 - d. XML

4. Which of the following is the default port for MongoDB which listens to the client for connection?
 - a. 27017
 - b. 20717
 - c. 21071
 - d. 20171

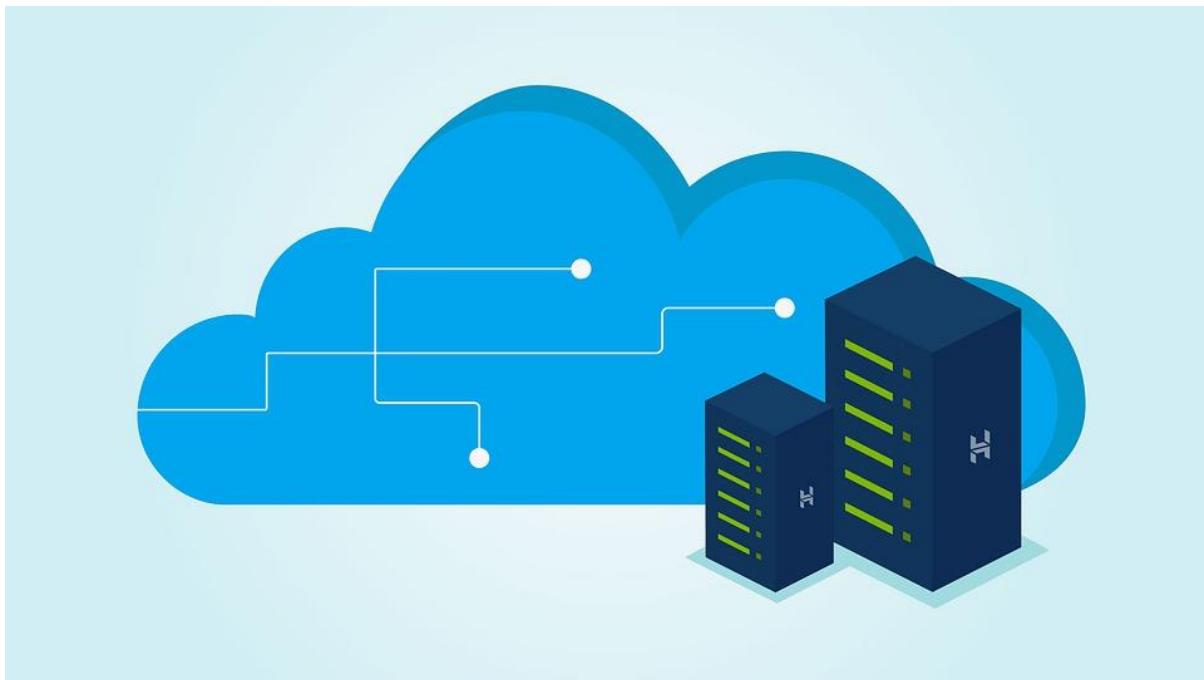
5. Which of the following service does MongoDB cloud Atlas provides?
 - a. Infrastructure as a service
 - b. Platform as a service
 - c. Data as a service
 - d. Function as a service

Answers to Test Your Knowledge

1	a, c, d
2	d
3	a, b, d
4	a
5	c

Try it Yourself

1. Install the **MongoDB 6.0** Community edition on your Windows.
2. Install **Mongo Shell 1.8.2** on your Windows system.
3. Using command prompt, set up the environment for MongoDB to execute `mongod` instance and connect `mongosh` with `mongod` instance. Execute a command to view the databases.



SESSION 2

MONGODB DATABASES

Learning Objectives

In this session, students will learn to:

- Explain the installation of MongoDB Database Tools and set the environment variable for MongoDB Database Tools
- Explain the method to load a sample dataset into MongoDB server
- Describe different datatypes in MongoDB
- Explain the methods to create database and collection
- Describe the ways to insert, query, update, and delete documents, collections, and databases

This session will provide an overview of installing MongoDB Database Tools and setting the environment variables for it. It will also explain how to load a sample database into MongoDB server. This session will explore the methods provided by MongoDB to create databases and collections. It will also explain the methods to insert, query, update, and delete documents, collections, and databases.

2.1 Install MongoDB Database Tools

MongoDB provides a collection of tools that facilitate the users to work with and manage databases. Table 2.1 describes some of the common MongoDB

tools.

Tool Name	Description
mongodump	Exports and creates a binary backup of the contents of a database
mongorestore	Restores backup data from mongodump into a database and loads standard input data into a mongod or mongos instance
mongoexport	Exports and creates a JavaScript Object Notation (JSON) or Comma-Separated Values (CSV) backup of the contents stored in a MongoDB instance
mongoimport	Imports data from CSV, JSON, and Tab-Separated Values (TSV) files into a MongoDB instance
bsondump	Converts BSON(Binary JavaScript Object Notation) files into JSON files
mongotop	Provides data on the amount of time spent by a MongoDB instance on reading and writing
mongofiles	Facilitates the manipulation of files stored in a MongoDB instance
mongostat	Provides statistics such as the number of inserts, deletes, and update queries executed per second and the amount of virtual memory used by the process

Table 2.1: Some Common MongoDB Tools

To install MongoDB database tools for Windows:

1. Open the browser and navigate to the Website
<https://www.mongodb.com/try/download/database-tools>
2. On this page, scroll down:
 - In the **Version** drop-down, ensure that **100.7.0** is selected.
 - In the **Platform** drop-down, ensure that **Windows x86_64** is selected.
 - In the **Package** drop-down, select the **msi** option.
3. Click **Download**.
4. After the download is complete, run the installer.
The **MongoDB Tools 100 Setup** wizard opens as shown in Figure 2.1.



Figure 2.1: MongoDB Tools Setup wizard

5. Click **Next**.
The **End-User License Agreement** page opens.
6. Select the **I accept the terms in the License Agreement** check box as shown in Figure 2.2.

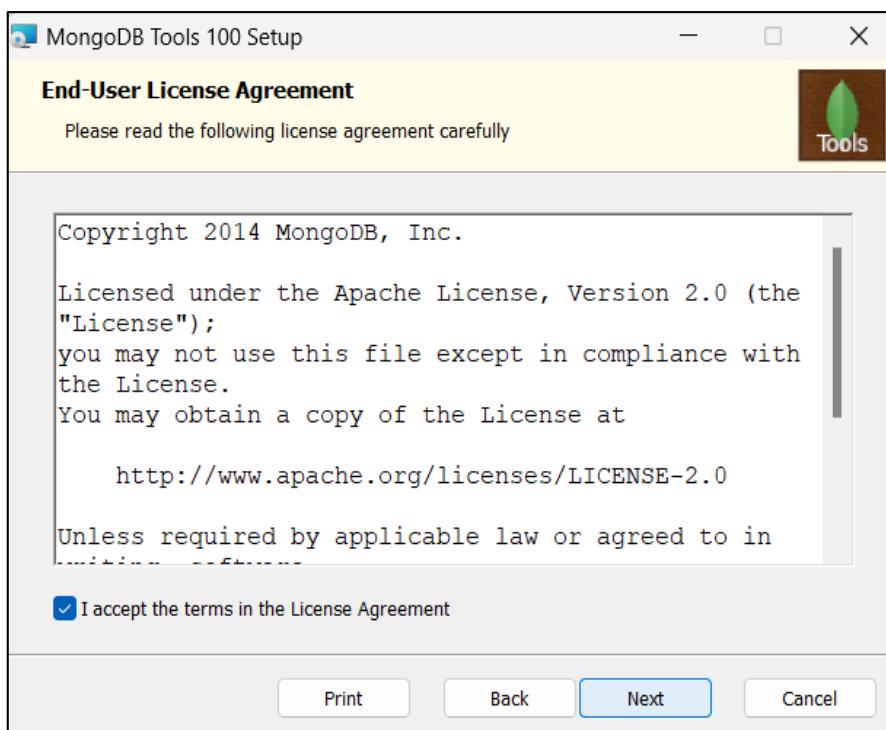


Figure 2.2: End-User License Agreement

7. Click **Next**.

The **Custom Setup** page opens as shown in Figure 2.3.

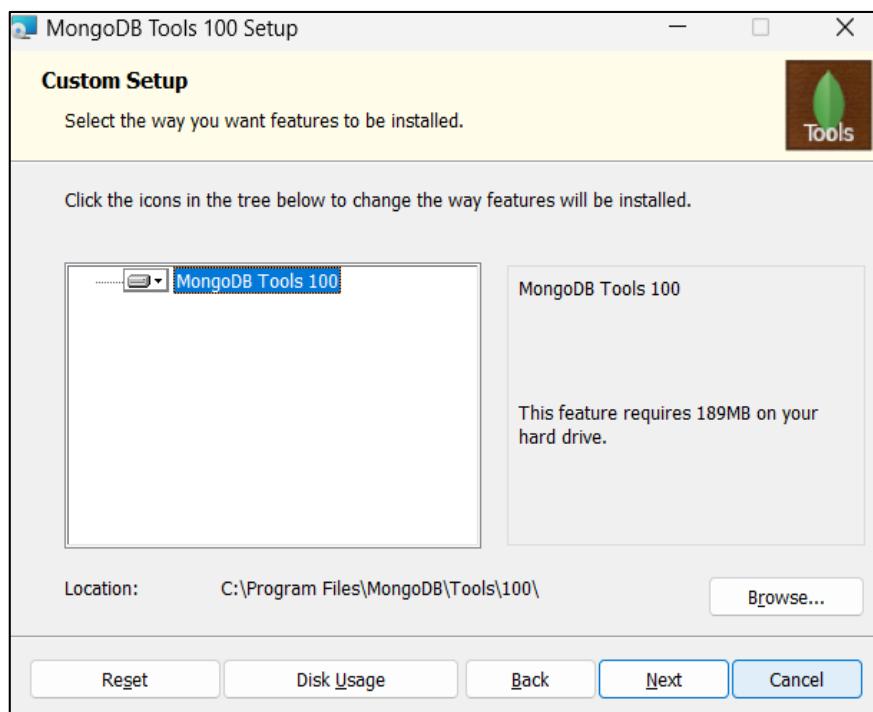


Figure 2.3: Custom Setup Page

8. Click **Next**.

The **Ready to Install MongoDB Tools 100** page opens as shown in Figure 2.4.

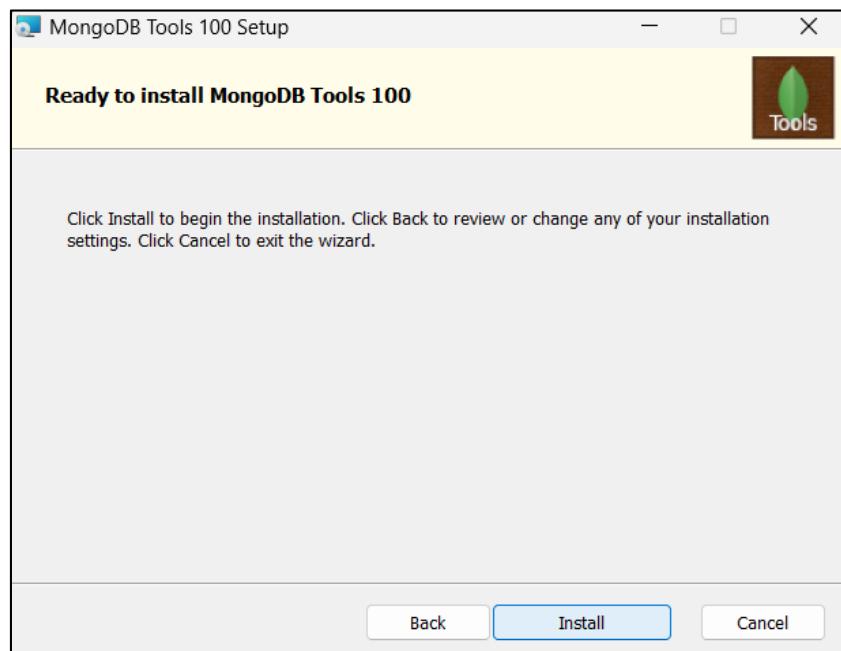


Figure 2.4: Ready to Install MongoDB Tools 100

9. Click **Install**.

The **Completed the MongoDB Tools 100 Setup Wizard** page opens as shown in Figure 2.5.



Figure 2.5: Completed the MongoDB Tools 100 Setup Wizard

10. Click **Finish**.

Now, MongoDB Database Tools are installed. The next step is to set the environment variables for the executable file. To set the environment variable:

1. Open the **Control Panel** in **Windows**.
2. In the **Control Panel** window, select **System and Security** as shown in Figure 2.6.

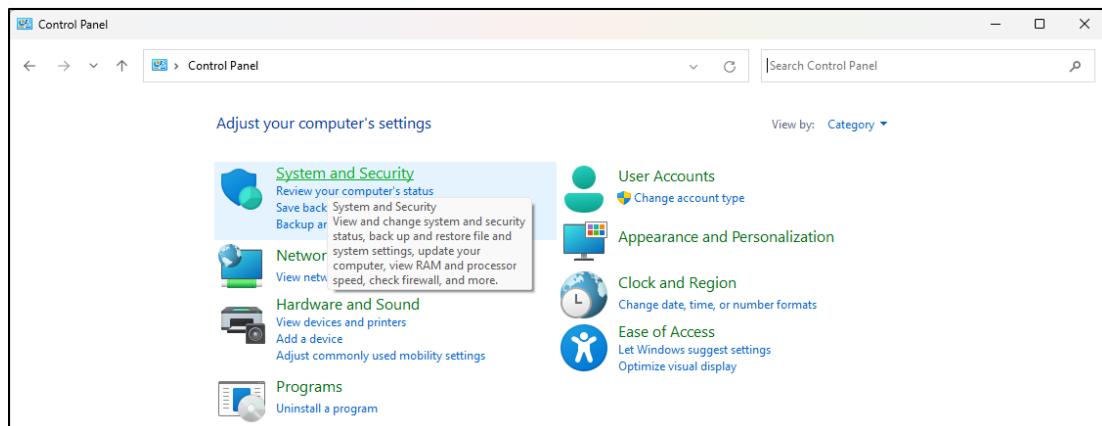


Figure 2.6: Control Panel

3. In the **System and Security** window that opens, click **System**.
4. In the **System → About** window that opens, select **Advanced system settings** as shown in Figure 2.7.

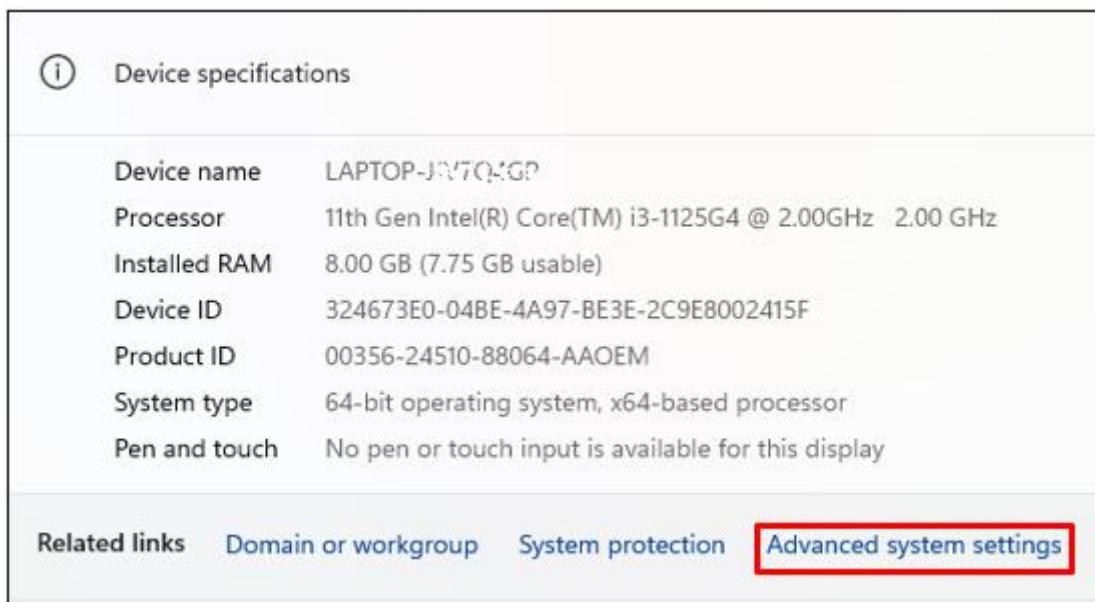


Figure 2.7: Advanced System Settings

- The **System Properties** window opens.
5. In the **System Properties** window, select **Environment Variables** as shown in Figure 2.8.

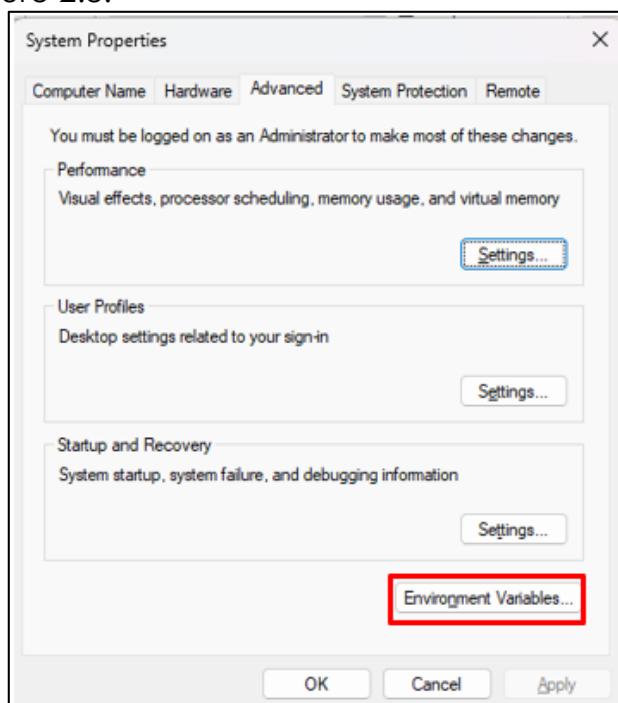


Figure 2.8: System Properties

6. In the **Environment Variables** window that opens, under **System variables** section, select **Path** and click **Edit** as shown in Figure 2.9.

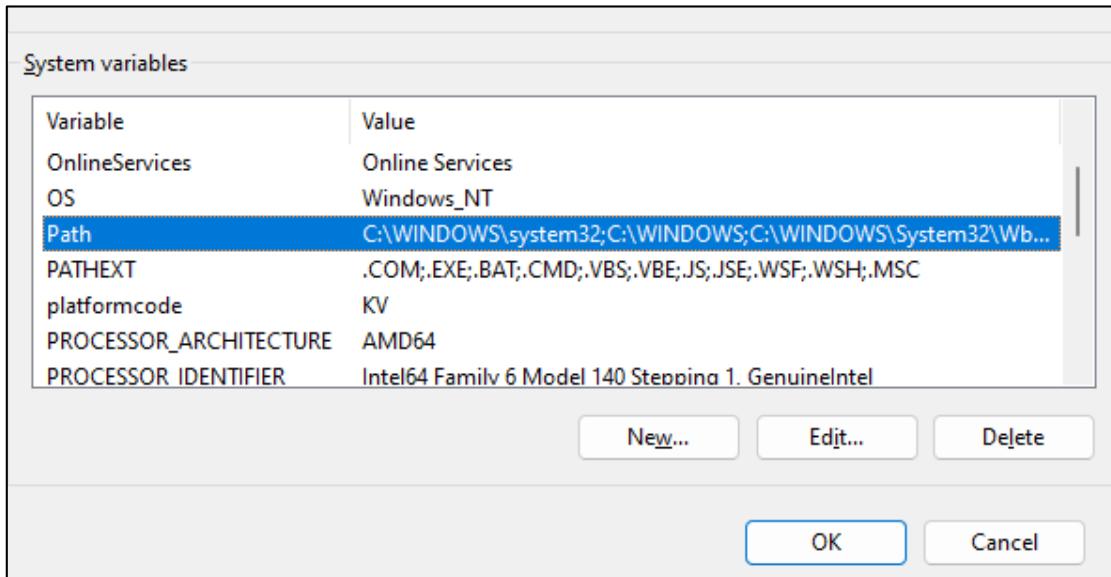


Figure 2.9: Path

7. In the **Edit environment variable** window that opens, click **New**, and then click **Browse** to navigate to the folder where the database tools are installed.
 8. Click **OK** in all the windows opened in Steps 5 through 7. The environment variable is now set for MongoDB tools.

2.2 Load Sample Dataset into MongoDB Server

To load a sample dataset into MongoDB Server using the `mongoimport` tool:

1. Copy the sample dataset named `Sample_dataset` to the C drive.
2. Open the command prompt.
3. To run the `mongod` instance, execute the command:

```
"C:\Program Files\MongoDB\Server\6.0\bin\mongod.exe" --dbpath="c:\data\db"
```

The command is executed as shown in Figure 2.10.

```
C:\>"C:\Program Files\MongoDB\Server\6.0\bin\mongod.exe" --dbpath="c:\data\db"
{"t": {"$date": "2023-05-15T17:39:00.769+05:30"}, "s": "I", "c": "CONTROL", "id": 23285, "ctx": "thread1", "msg": "Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}
{"t": {"$date": "2023-05-15T17:39:02.257+05:30"}, "s": "I", "c": "NETWORK", "id": 4915701, "ctx": "thread1", "msg": "Initialized wire specification", "attr": {"spec": {"incomingExternalClient": {"minWireVersion": 0, "maxWireVersion": 17}, "incomingInternalClient": {"minWireVersion": 0, "maxWireVersion": 17}, "outgoing": {"minWireVersion": 6, "maxWireVersion": 17}, "isInternalClient": true}}}
```

Figure 2.10: Starting the MongoDB Database

The `--dbpath` option shows the database directory. If the MongoDB database server is running correctly, the message waiting for connections is displayed as shown in Figure 2.11.

```
{"t": {"$date": "2023-05-15T17:39:03.517+05:30"}, "s": "I", "c": "NETWORK", "id": 23015, "ctx": "listener", "msg": "Listener is listening on", "attr": {"address": "127.0.0.1"}}, {"t": {"$date": "2023-05-15T17:39:03.517+05:30"}, "s": "I", "c": "NETWORK", "id": 23016, "ctx": "listener", "msg": "Waiting for connections", "attr": {"port": 27017, "ssl": "off"}}
```

Figure 2.11: Waiting for Connections Message

Now, the MongoDB instance `mongod` is successfully running.

4. Open a new command prompt and type the command as:

```
mongoimport --db sample_weatherdata --collection data --jsonArray --file C:\Sample_dataset\sample_weatherdata\data.json
```

The command executes as shown in Figure 2.12.

```
PS C:\Users\linda> mongoimport --db sample_weatherdata --collection data --jsonArray --file C:\Sample_dataset\sample_weatherdata\data.json
2023-05-15T17:54:39.729+0530      connected to: mongodb://localhost/
2023-05-15T17:54:40.578+0530      10000 document(s) imported successfully. 0 document(s) failed to import.
```

Figure 2.12: Import Sample Data

The sample dataset is now imported into the `mongod` instance.

5. To verify this, run the MongoDB shell command as:

```
mongosh
```

The command executes as shown in Figure 2.13.

```
(c) Microsoft Corporation. All rights reserved.  
C:\Users\Linda>mongosh  
Current Mongosh Log ID: 64506c466dfaf4b020136355  
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.2  
Using MongoDB:      6.0.5  
Using Mongosh:      1.8.2  
  
For mongosh info see: https://docs.mongodb.com/mongodb-shell/  
----  
The server generated these startup warnings when booting  
2023-05-02T06:51:37.406+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted  
----  
----  
Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).  
The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.  
To enable free monitoring, run the following command: db.enableFreeMonitoring()  
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()  
----  
test> |
```

Figure 2.13: mongosh Command

6. To view the databases available in this mongod instance, execute the command as:

```
show dbs
```

The command executes as shown in Figure 2.14.

```
test> show dbs  
admin              40.00 KiB  
config             60.00 KiB  
local              72.00 KiB  
sample_weatherdata  2.68 MiB
```

Figure 2.14: show dbs Command

This confirms that the sample data named `sample_weatherdata` is imported into the mongod instance.



The `show dbs` command would display a database only when the database contains at least one document.

2.3 Datatypes in MongoDB

MongoDB stores data internally in BSON format. BSON is an extension of JSON. It supports all the data types that JSON supports. Additionally, it supports other datatypes such as date, timestamp, and binary data. Table 2.2 describes the

datatypes that are available in MongoDB.

Datatype	Description	Example
String	A String is used to store text in MongoDB. BSON strings are of Universal Coded Character Set (UCS) Transformation Format (UTF – 8).	Store the name of a student or employee, designation of an employee, and address
Integer	An Integer is used to store numeric values. MongoDB offers two variations of integer datatype—32-bit signed integer and 64-bit signed integer.	Store numeric values such as marks of a student, salary of an employee, and count of products in inventory
Double	A Double is used to store numeric values that contain 8-byte floating point values.	Store temperature, revenue, and price of an item
Decimal	Decimal is used to store 128-bit floating point decimal values.	Store financial and scientific computations
Boolean	Boolean is used to store true or false as values for the fields.	Store the value of 'Is_married', 'Is_fulltime_employee', and 'Is_passed'
Null	Null is used to store NULL value.	Store the driver's license number of people who do not own a driver's license
Array	Array is a collection of multiple values that are of same type or different types.	Store marks of a student in five subjects
Object	Object is used to store embedded documents.	Store the description of an item which would include the item's shape, color, and dimensions
ObjectID	ObjectID is a unique identifier for each document in a collection. It is a hexadecimal string of 12-bytes that includes: <ul style="list-style-type: none"> • Timestamp value (4-bytes) • Random value(5-bytes with 3-bytes of machine ID and 2-bytes of process ID) • Counter (3-bytes) 	A hexadecimal ID that MongoDB assigns to each new field in MongoDB, for example, '64617a0420cf1016275bd88f'
Date	Stores date as a 64-bit signed integer which is a Universal Time Co-ordinated (UTC)	String date: Mon May 15 2023 01:03:25 GMT-0500 (EST)

Datatype	Description	Example
	<p>datetime format. This BSON date format allows storage of dates before and after Jan 1, 1970, 00:00:00 using the negative and positive sign-bits, respectively.</p> <p>When the user creates a date variable in MongoDB, it returns the date in string format.</p> <p>When the user creates a date variable in MongoDB using the new keyword, MongoDB returns it in the International Standards Organization Date (ISODate) format.</p>	<p>ISODate: 2023-05-15T04:25:48.512Z</p>

Table 2.2: Datatypes in MongoDB

2.4 Working with Databases and Collections

Consider that the user wants to create a database in MongoDB and perform certain operations in it. MongoDB offers a set of commands that facilitates the user to create and drop databases and collections. MongoDB also provides commands that can be used to insert, query, update, and delete documents in a collection.

2.4.1 Create Database

Consider that the user wants to create a database named `Student_detail`. The syntax to create a database is:

```
use DATABASE_NAME
```

To create a database named `Student_detail`:

1. Ensure that the `mongod` instance is running.
2. To run the MongoDB Shell, at the command prompt, execute the command as:

```
mongosh
```

3. To create a database named `Student_detail`, run the command as:

```
use Student_detail
```

The command executes as shown in Figure 2.15.

```
test> use Student_detail
switched to db Student_detail
Student_detail> |
```

Figure 2.15: Create Database

The database named `Student_detail` has been created.

2.4.2 Create Collection

Consider that the user wants to create a collection named `Studentinfo` in the `Student_detail` database. The syntax to create a collection or a view is:

```
db.createCollection(name, options)
```

Table 2.3 describes the parameters of the `createCollection` command.

Parameter	Type	Description
name	String	This parameter specifies the name of the collection to be created. This is a mandatory parameter.
options	Document	<p>This parameter is optional. It includes fields that specify whether the entity to be created must be a:</p> <ul style="list-style-type: none"> • Capped collection: Capped collections are collections of fixed size. If the maximum size of the collection is reached, then the oldest documents are erased to create space for new entries. The size parameter and the maximum number of documents that can be created must be specified for this option. • Timeseries collection: Timeseries collections are collections that store sequences of data points (key-value pairs) over a period along with the time at which the data points were recorded. • Clustered collection: Clustered collections are collections that are stored in the order of a clustered index—a key value that specifies the order of arrangement of the documents in a collection. The parameter includes fields such as key value field, uniqueness of the clustered index, and name of the index. • View: View is a virtual collection. If a view must be created, the name of the source collection or view must be specified. <p>This parameter also includes a field named validator that allows the users to create collections with user-defined validation rules.</p>

Table 2.3: Parameters of `createCollection()` Method

To create a collection named `Studentinfo` in the database named `Student_detail`:

1. To switch to the `Student_detail` database, execute the command as:

```
use Student_detail
```

2. To create a collection named `Studentinfo`, execute the command

as:

```
db.createCollection("Studentinfo")
```

The command executes as shown in Figure 2.16.

```
Student_detail> db.createCollection("Studentinfo")
{ ok: 1 }
Student_detail> |
```

Figure 2.16: Create Collection

The Studentinfo collection is created.

3. To view the collection inside the Student_detail database, execute the command as:

```
show collections
```

The command is executed as shown in Figure 2.17.

```
Student_detail> show collections
Studentinfo
Student_detail> |
```

Figure 2.17: Show Collections

2.4.3 Insert Document

Documents can be inserted into a collection in a database. MongoDB provides options to insert a single document or multiple documents into a collection.

Insert a Single Document

The syntax to insert a single document into a collection is:

```
db.collection.insertOne()
```

Consider that the user wants to insert a document into the Studentname collection in the Student_detail database.

To perform this task, execute the commands as:

```
db.createCollection("Studentname")
db.Studentname.insertOne({ "Name": "Richard"})
```

The command executes as shown in Figure 2.18.

```
Student_detail> db.createCollection("Studentname")
{ ok: 1 }
Student_detail> db.Studentname.insertOne({ "Name": "Richard"})
{
  acknowledged: true,
  insertedId: ObjectId("646219f520cf1016275bd893")
}
```

Figure 2.18: Insert Single Document

A document is inserted into the Studentname collection.

The insertedID value in the output is the unique identifier generated by MongoDB for each document. This identifier is of the ObjectId datatype.



If the user tries to insert a document into a collection that does not exist in the database, then MongoDB automatically creates the collection.

Insert Multiple Documents

The syntax to insert multiple documents in a collection is:

```
db.collection.insertMany()
```

Consider that the user wants to insert three documents into the Studentmarks collection in the Student_detail database.

To perform this task, execute the command as:

```
db.Studentmarks.insertMany([
  { Name: "Robert", Age: 16, Subject1: 89, Subject2: 78},
  { Name: "Oliver", Age: 17, Subject1: 78, Subject2: 85} ,
  { Name: "Henry", Age: 15, Subject1: 90, Subject2: 93} ])
```

The command executes as shown in Figure 2.19.

```
Student_detail> db.Studentmarks.insertMany([
... { Name: "Robert", Age: 16, Subject1: 89, Subject2: 78},
... { Name: "Oliver", Age: 17, Subject1: 78, Subject2: 85} ,
... { Name: "Henry", Age: 15, Subject1: 90, Subject2: 93} ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("64608b0820cf1016275bd88c"),
    '1': ObjectId("64608b0820cf1016275bd88d"),
    '2': ObjectId("64608b0820cf1016275bd88e")
  }
}
```

Figure 2.19: Insert Many Documents

2.4.4 Query Document

MongoDB provides commands to retrieve the documents from a collection. The syntax to retrieve all the documents in a collection is:

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

Consider that the user wants to retrieve all the documents in the Studentmarks collection. To perform this task, execute the command as:

```
db.Studentmarks.find()
```

The command executes as shown in Figure 2.20.

```
Student_detail> db.Studentmarks.find()
[  
  {  
    _id: ObjectId("64608b0820cf1016275bd88c"),  
    Name: 'Robert',  
    Age: 16,  
    Subject1: 89,  
    Subject2: 78  
  },  
  {  
    _id: ObjectId("64608b0820cf1016275bd88d"),  
    Name: 'Oliver',  
    Age: 17,  
    Subject1: 78,  
    Subject2: 85  
  },  
  {  
    _id: ObjectId("64608b0820cf1016275bd88e"),  
    Name: 'Henry',  
    Age: 15,  
    Subject1: 90,  
    Subject2: 93  
  }  
]
```

Figure 2.20: Retrieve Documents in a Collection

All the documents in the Studentmarks collection are retrieved and displayed.

2.4.5 Update Document

MongoDB provides two different methods to update documents in a collection. These methods can be used to:

- Update single document
- Update multiple documents

Update Single Document

The syntax to update a single document is:

```
db.collection.updateOne(filter, update, options)
```

Table 2.4 describes the parameters of the `updateOne` method.

Parameter	Type	Description
<code>filter</code>	Document	This parameter specifies the selection criteria based on which the update must be performed. If multiple documents satisfy the criteria, then only the first document is updated. This parameter is mandatory.
<code>update</code>	Document	This parameter specifies the change that must be applied to the filtered documents. This parameter is mandatory.
<code>options</code>	Document	This parameter specifies some additional options— <code>upsert</code> , <code>writeconcern</code> , <code>collation</code> , <code>arrayfilters</code> , and <code>hint</code> —for the update method. These options will be explored in detail in later sessions. This parameter is optional.

Table 2.4: Parameters of `updateOne()` Method

Consider that the user wants to update the value of `name` field as "David" where the value in the `Name` field is "Robert". To perform this task, execute the command as:

```
db.Studentmarks.updateOne({ "Name": "Robert" },
{ $set: { "Name": "David" } })
```

The command executes as shown in Figure 2.21.

```
Student_detail> db.Studentmarks.updateOne({ "Name": "Robert" }, { $set: { "Name": "David" } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Figure 2.21: Method to Update Single Document

The document is updated. To view the updated document, execute the command as:

```
db.Studentmarks.find()
```

The command executes as shown in Figure 2.22.

```
Student_detail> db.Studentmarks.find()
[
  {
    _id: ObjectId("64608b0820cf1016275bd88c"),
    Name: 'David',
    Age: 16,
    Subject1: 89,
    Subject2: 78
  },
  {
    _id: ObjectId("64608b0820cf1016275bd88d"),
    Name: 'Oliver',
    Age: 17,
    Subject1: 78,
    Subject2: 85
  },
  {
    _id: ObjectId("64608b0820cf1016275bd88e"),
    Name: 'Henry',
    Age: 15,
    Subject1: 90,
    Subject2: 93
  }
]
```

Figure 2.22: Updated Document

Update Multiple Documents

The syntax to update multiple documents is:

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

Consider that the user wants to update and set the value of Subject2 as 99 in all documents where the value of Subject1 is greater than 80. To perform this task, execute the command as:

```
db.Studentmarks.updateMany({ "Subject1": { $gt: 80 } },
{ $set: { "Subject2": 99 } })
```

The command executes as shown in Figure 2.23.

```
Student_detail> db.Studentmarks.updateMany({ "Subject1":{$gt: 80}}, { $set:{ "Subject2":99}})  
{  
  acknowledged: true,  
  insertedId: null,  
  matchedCount: 2,  
  modifiedCount: 2,  
  upsertedCount: 0  
}
```

Figure 2.23: Method to Update Many Documents

The document is updated. In this command, \$gt is a comparison operator that is used to filter out only those documents with Subject1 greater than 80.

To view the updated documents, execute the command as:

```
db.Studentmarks.find()
```

The command executes as shown in Figure 2.24.

```
Student_detail> db.Studentmarks.find()  
[  
  {  
    _id: ObjectId("64617a0420cf1016275bd88f"),  
    Name: 'Robert',  
    Age: 16,  
    Subject1: 89,  
    Subject2: 99  
  },  
  {  
    _id: ObjectId("64617a0420cf1016275bd890"),  
    Name: 'Oliver',  
    Age: 17,  
    Subject1: 78,  
    Subject2: 85  
  },  
  {  
    _id: ObjectId("64617a0420cf1016275bd891"),  
    Name: 'Henry',  
    Age: 15,  
    Subject1: 90,  
    Subject2: 99  
  }  
]
```

Figure 2.24: Updated Documents

Here, two documents are updated as they satisfy the specified criteria.

2.4.6 Delete Document

MongoDB provides two different methods to delete documents in a collection. These methods can be used to:

- Delete single document
- Delete many documents

Delete Single Document

The syntax to delete single document is:

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

Table 2.5 describes the parameters of the `deleteOne` method.

Parameter	Type	Description
filter	Document	This parameter specifies the selection criteria based on which the delete must be performed. If an empty criterion is mentioned, then the first document returned in the collection is deleted. This parameter is mandatory.
options	Document	This parameter specifies some additional options—writeconcern, collation, and hint—for the delete method. These options will be explored in detail in later sessions. This parameter is optional.

Table 2.5: Parameters of `deleteOne` Method

Consider that the user wants to delete the document where the value of the Name field is "Oliver". To perform this task, execute the command as:

```
db.Studentmarks.deleteOne({ "Name": "Oliver" })
```

The command executes as shown in Figure 2.25.

```
Student_detail> db.Studentmarks.deleteOne({ "Name": "Oliver" })
{ acknowledged: true, deletedCount: 1 }
```

Figure 2.25: Method to Delete Single Document

The document where the value of the name field is "Oliver" is deleted. To verify this, execute the command as:

```
db.Studentmarks.find()
```

The command executes as shown in Figure 2.26.

```
student_detail> db.Studentmarks.find()
[
  {
    _id: ObjectId("64617a0420cf1016275bd88f"),
    Name: 'Robert',
    Age: 16,
    Subject1: 89,
    Subject2: 99
  },
  {
    _id: ObjectId("64617a0420cf1016275bd891"),
    Name: 'Henry',
    Age: 15,
    Subject1: 90,
    Subject2: 99
  }
]
```

Figure 2.26: Output of deleteOne Method

Delete Multiple Documents

The syntax to delete multiple documents is:

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

Table 2.6 describes the parameters of the `deleteMany` method.

Parameter	Type	Description
filter	Document	This parameter specifies the selection criteria based on which the delete must be performed. If an empty criterion is mentioned, then all the documents in the collection are deleted. This parameter is mandatory.
options	Document	This parameter specifies some additional options—writeconcern, collation, and hint—for the delete method. These options will be explored in detail in later sessions. This parameter is optional.

Table 2.6: Parameters of deleteMany Method

Consider that the user wants to delete all the documents where the value in the `Age` field is greater than 14. To perform this task, execute the command as:

```
db.Studentmarks.deleteMany({ "Age": { $gt:14 } })
```

The command executes as shown in Figure 2.27.

```
Student_detail> db.Studentmarks.deleteMany({ "Age": { $gt:14 } })
{ acknowledged: true, deletedCount: 2 }
```

Figure 2.27: Method to Delete Many Documents

Those documents where the value of the `Age` field is greater than 14 are deleted. To view the output, execute the command as:

```
db.Studentmarks.find()
```

The command executes as shown in Figure 2.28.

```
Student_detail> db.Studentmarks.find()
Student_detail> |
```

Figure 2.28: Output of `deleteMany` Method

2.4.7 Drop Collection

MongoDB provides a method to remove a collection from the database. The syntax to drop a collection is:

```
db.collection.drop()
```

Consider that the user wants to drop the collection named `Studentmarks` from the `Student_detail` database. To perform this task, execute the command as:

```
db.Studentmarks.drop()
```

The command executes as shown in Figure 2.29.

```
Student_detail> db.Studentmarks.drop()  
true
```

Figure 2.29: Dropping a Collection

The Studentmarks collection is dropped. In MongoDB, when a collection is dropped, all the indexes associated with it are also dropped.

2.4.8 Drop Database

MongoDB provides a method to drop a database. The syntax to drop a database is:

```
db.dropDatabase()
```

This method drops the current database. Consider that the user wants to drop the Student_detail database. To perform this task, execute the command as:

```
db.dropDatabase()
```

The command is executed as shown in Figure 2.30.

```
Student_detail> db.dropDatabase()  
{ ok: 1, dropped: 'Student_detail' }
```

Figure 2.30: Method to Drop a Database

The Student_detail database is dropped. When a database is dropped, all the collections and data files associated with it are also dropped. Here, the Studentinfo and Studentname collections are also dropped.

2.5 Summary

- MongoDB provides a set of tools that helps users to manage the databases in the MongoDB environment.
- To start using the commands in MongoDB, install MongoDB Database Tools and set the environment variable for it.
- A sample dataset can be loaded into MongoDB server using the mongoimport tool.
- MongoDB stores data in BSON format. It supports various datatypes such as string, integer, double, Boolean, null, array, object, objectID, and date.
- MongoDB provides different commands and methods to create and delete databases and collections. It also provides methods to insert, query, update, and delete documents and collections.

Test Your Knowledge

1. Which of the following file formats can be imported into MongoDB instance using the mongoimport database tool?
 - a. CSV
 - b. JSON
 - c. DBF
 - d. TSV

2. Which of the following commands is used to view the databases on the MongoDB server?
 - a. show databases
 - b. view dbs
 - c. show dbs
 - d. view databases

3. Consider that the user wants to create a database named Employee_detail. Which of the following commands must be used to create the database?
 - a. use Employee_detail
 - b. create Employee_detail
 - c. create database Employee_detail
 - d. use database Employee_detail

4. Which of the following datatypes is used to store embedded documents in MongoDB?
 - a. Array
 - b. Object
 - c. String object
 - d. Array Object

5. Which of the following is the correct syntax to delete a collection from the current database?
 - a. db.collection.drop
 - b. db.drop
 - c. db.collection.delete
 - d. db.delete

Answers to Test Your Knowledge

1	a, b, d
2	c
3	a
4	b
5	a

Try it Yourself

1. Install MongoDB Database Tools and set the environment variable for MongoDB Database Tools.
2. Load the sample dataset `sample_training` into the MongoDB server.
3. Create a database named `Employee` and a collection named `Employee_detail`.
4. Insert the following documents into the `Employee_detail` collection.

```
[ {  
    Emp_name: "Oliver Smith",  
    Age: 23,  
    Designation: "Manager"  
},  
{  
    Emp_name: "David Michael",  
    Age: 32,  
    Designation: "Software Engineer"  
} ]
```
5. Update the designation of "Oliver Smith" to "Accountant".
6. Delete the document in `Employee_detail` collection where the name of the employee is "David Michael".
7. Drop the collection `Employee_detail`.
8. Drop the database `Employee`.



SESSION 3

MONGODB OPERATORS

Learning Objectives

In this session, students will learn to:

- Describe the types of MongoDB operators
- Explain how to use the query and projection operators
- Explain how to modify field and array data using update operators

MongoDB provides various operators which enable users to effectively interact with the database.

This session describes the types of MongoDB operators and their use in general. This session also describes in detail the function of the more commonly used operators with real-life examples.

3.1 Introduction to MongoDB Operators

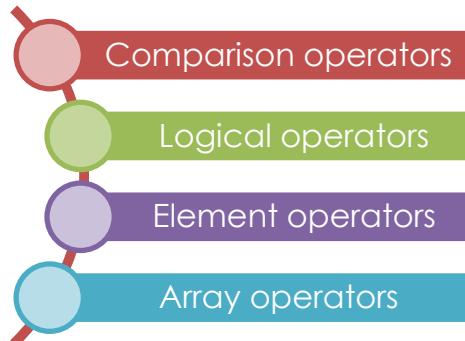
Operators are reserved words or symbols that instruct the compiler or interpreter to perform specific mathematical or logical operations on the dataset as required.

Some of the types of operators in MongoDB are:

- Query operators
- Projection operators
- Update operators
- Miscellaneous operators

3.2 Query Operators

Query operators are used to retrieve data from a database. Different types of query operators are:



3.2.1 Comparison Operators

Comparison operators compare the field's value with the specified value and return the documents that match the value. Table 3.1 describes the comparison operators in MongoDB.

Operator	Description
\$eq	Used to retrieve, update, or delete documents where the field value is equal to a specified value
\$gt	Used to retrieve, update, or delete documents where the field value is greater than a specified value
\$gte	Used to retrieve, update, or delete documents where the field value is greater than or equal to a specified value
\$in	Used to retrieve, update, or delete documents where the field value is any of the values specified in an array
\$lt	Used to retrieve, update, or delete documents where the field value is less than a specified value
\$lte	Used to retrieve, update, or delete documents where the field value is less than or equal to a specified value
\$ne	Used to retrieve, update, or delete documents where the field value is not equal to a specified value
\$nin	Used to retrieve, update, or delete documents where the field value is none of the values specified in an array.

Table 3.1: Comparison Operators in MongoDB

Let us take a look at how to use some of these comparison operators.

\$eq Operator

This operator checks whether the value of a field equals the specified value. The syntax for this operator is:

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

Let us take a look at how to use this operator.

Figure 3.1 shows the documents stored in the `accounts` collection in the sample database named `sample_analytics`.

```
sample_analytics> db.accounts.find()
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238c"),
    account_id: 371138,
    limit: 9000,
    products: [ 'Derivatives', 'InvestmentStock' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238d"),
    account_id: 557378,
    limit: 10000,
    products: [ 'InvestmentStock', 'Commodity', 'Brokerage', 'CurrencyService' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),
    account_id: 198100,
    limit: 10000,
    products: [ 'Derivatives', 'CurrencyService', 'InvestmentStock' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162390"),
    account_id: 278603,
    limit: 10000,
    products: [ 'Commodity', 'InvestmentStock' ]
  }
]
```

Figure 3.1: Documents in the Accounts Collection

Now, consider that the user wants to view the `accounts` which have the `limit` value as 9000. To do so, the user can execute a query that uses the `$eq` operator as:

```
db.accounts.find({limit:{$eq:9000}})
```

Figure 3.2 shows the output of this query.

```
sample_analytics> db.accounts.find({limit:{$eq:9000}})
[{"_id": ObjectId("5ca4bbc7a2dd94ee5816238c"), "account_id": 371138, "limit": 9000, "products": ["Derivatives", "InvestmentStock"]}, {"_id": ObjectId("5ca4bbc7a2dd94ee58162392"), "account_id": 794875, "limit": 9000, "products": ["InvestmentFund", "InvestmentStock"]}, {"_id": ObjectId("5ca4bbc7a2dd94ee581623e0"), "account_id": 161714, "limit": 9000, "products": ["Commodity", "CurrencyService", "Derivatives", "InvestmentFund", "InvestmentStock"]}, {"_id": ObjectId("5ca4bbc7a2dd94ee581623ee")}]
```

Figure 3.2: Output of the Query with \$eq Operator

\$lt Operator

This operator checks whether the value of a field is less than the specified value. The syntax for this operator is:

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

For example, consider that in the `accounts` collection, the user wants to view the documents where the `limit` value is less than 5000. To do so, the user can execute a query that uses the `$lt` operator as:

```
db.accounts.find({limit:{$lt:5000}})
```

Figure 3.3 shows the output of the query.

```
sample_analytics> db.accounts.find({limit:{$lt:5000}})  
[  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee58162661"),  
    account_id: 417993,  
    limit: 3000,  
    products: [ 'InvestmentStock', 'InvestmentFund' ]  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),  
    account_id: 113123,  
    limit: 3000,  
    products: [ 'CurrencyService', 'InvestmentStock' ]  
  }  
]  
sample_analytics>
```

Figure 3.3: Output of the Query with \$lt Operator

\$in Operator

This operator checks whether the value of a field matches any of the values in the specified array. The syntax for this operator is:

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

For example, consider that from the `accounts` collection, the user wants to view only the documents where the product is either a 'Commodity' or an 'InvestmentStock'. To do so, the user can execute a query that uses the `$in` operator as:

```
db.accounts.find({products:  
{$in: ['Commodity', 'InvestmentStock']} })
```

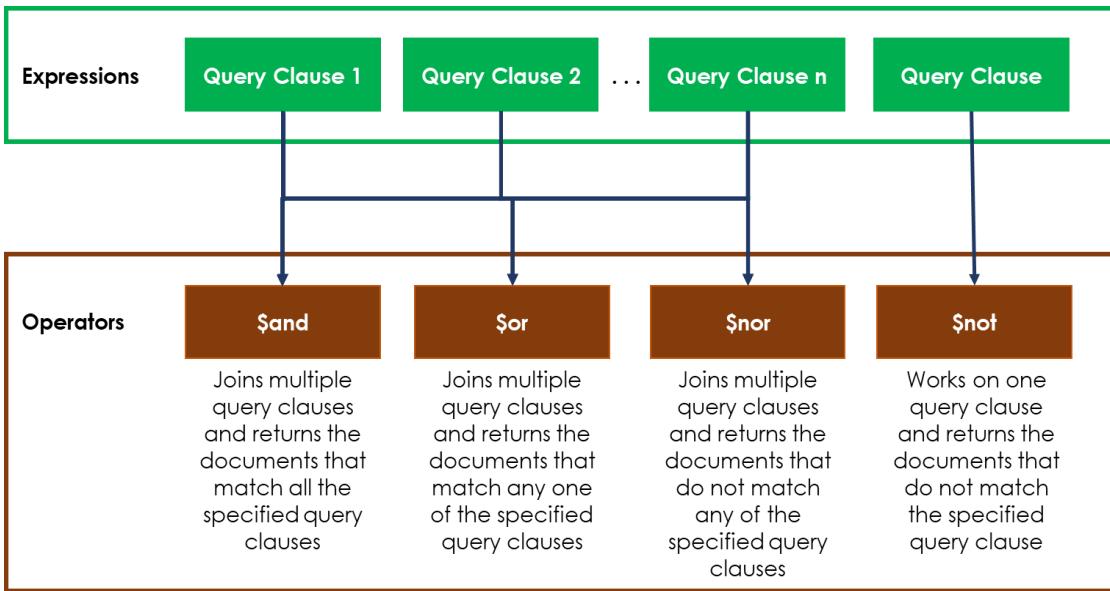
Figure 3.4 shows the output of the query.

```
'sample_analytics> db.accounts.find({products: {$in:[ 'Commodity', 'InvestmentStock']}})'
[  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816238c"),  
    account_id: 371138,  
    limit: 9000,  
    products: [ 'Derivatives', 'InvestmentStock' ]  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816238d"),  
    account_id: 557378,  
    limit: 10000,  
    products: [ 'InvestmentStock', 'Commodity', 'Brokerage', 'CurrencyService' ]  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),  
    account_id: 198100,  
    limit: 10000,  
    products: [ 'Derivatives', 'CurrencyService', 'InvestmentStock' ]  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee58162390"),  
    account_id: 278603,  
    limit: 10000,  
    products: [ 'Commodity', 'InvestmentStock' ]  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee58162392"),  
    account_id: 198100,  
    limit: 10000,  
    products: [ 'Derivatives', 'CurrencyService', 'InvestmentStock' ]  
  }]
```

Figure 3.4: Output of the Query with \$in Operator

3.2.2 Logical Operators

Logical operators evaluate multiple expressions and return a Boolean value (true, false). Four logical operators are: \$and, \$or, \$nor, and \$not:



\$and Operator

This operator, as the name suggests, performs a logical AND operation. This means that the operator lists only the documents that match all the expressions in the specified array.

The syntax for this operator is:

```
{ $and: [ { <expression1> }, { <expression2> } , ... ,
          { <expressionN> } ] }
```

Consider that from the accounts collection, the user wants to view only the documents where the limit is greater than or equal to 5000 and less than 7000. To do so, the user can execute a query that uses the \$and operator as:

```
db.accounts.find({$and: [{limit:{$gte:5000}}, {limit:{$lt:7000}}]})
```

Figure 3.5 shows the output of this query.

```
sample_analytics> db.accounts.find({$and: [{limit:{$gte:5000}}, {limit:{$lt:7000}}]})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816272e"),
    account_id: 170980,
    limit: 5000,
    products: [
      'InvestmentFund',
      'Brokerage',
      'CurrencyService',
      'Derivatives',
      'InvestmentStock'
    ]
  }
]
```

Figure 3.5: Output of the Query with \$and Operator

\$or Operator

The \$or operator performs a logical OR operation. This operator lists documents that satisfy at least one of the expressions in the specified array.

The syntax for this operator is:

```
{ $or: [ { <expression1> }, { <expression2> }, ... , {
  <expressionN> } ] }
```

Consider that from the account collection, the user wants to view all the documents where the limit is less than or equal to 5000 or equal to 7000. To do, the user can execute a query that uses the \$or operator as:

```
db.accounts.find({$or: [{limit:{$lte:5000}}, {limit:{$eq:7000}}] })
```

Figure 3.6 shows the output of the query.

```
sample_analytics> db.accounts.find({$or: [{limit:{$lte:5000}}, {limit:{$eq:7000}}]})  
[  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee58162458"),  
    account_id: 852986,  
    limit: 7000,  
    products: [  
      'Derivatives',  
      'Commodity',  
      'CurrencyService',  
      'InvestmentFund',  
      'InvestmentStock'  
    ]  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816247a"),  
    account_id: 777752,  
    limit: 7000,  
    products: [  
      'CurrencyService',  
      'Brokerage',  
      'Commodity',  
      'InvestmentFund',  
      'InvestmentStock'  
    ]  
  },  
  {
```

Figure 3.6: Output of the Query with \$or Operator

\$nor Operator

This operator performs a logical NOR operation. It lists the documents that do not match any of the expressions in the specified array of expressions.

The syntax for this operator is:

```
{ $nor: [ { <expression1> }, { <expression2> }, ... {  
  <expressionN> } ] }
```

For example, consider that from the `accounts` collection, the user wants to view all the documents where the `limit` is neither 9000 nor 10000. To do so, the users can execute a query that uses the `$nor` operator as:

```
db.accounts.find({$nor: [{limit:9000}, {limit:10000}]})
```

Figure 3.7 shows the output of this query.

```
sample_analytics> db.accounts.find({$nor: [{limit:9000}, {limit:10000}]}  
[  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee58162458"),  
    account_id: 852986,  
    limit: 7000,  
    products: [  
      'Derivatives',  
      'Commodity',  
      'CurrencyService',  
      'InvestmentFund',  
      'InvestmentStock'  
    ]  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816247a"),  
    account_id: 777752,  
    limit: 7000,  
    products: [  
      'CurrencyService',  
      'Brokerage',  
      'Commodity',  
      'InvestmentFund',  
      'InvestmentStock'  
    ]  
  },  
]
```

Figure 3.7: Output of the Query with \$nor Operator

\$not Operator

This operator performs a logical NOT operation. It lists the documents that do not match the specified expression.

The syntax for this operator is:

```
{ field: { $not: { <operator-expression> } } }
```

For example, consider that from the accounts collection, the user wants to view all the documents where limit is not greater than or equal to 5000.

To do so, the user can execute a query that uses the \$not operator as:

```
db.accounts.find( {limit: {$not:{$gte:5000}}})
```

Figure 3.8 shows the output of this query.

```
sample_analytics> db.accounts.find( {limit: {$not:{$gte:5000}}})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162661"),
    account_id: 417993,
    limit: 3000,
    products: [ 'InvestmentStock', 'InvestmentFund' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),
    account_id: 113123,
    limit: 3000,
    products: [ 'CurrencyService', 'InvestmentStock' ]
  }
]
sample_analytics>
```

Figure 3.8: Output of the Query with \$not Operator

3.2.3 Element Operators

Element operators are used to check if a particular field is available in a MongoDB document or if a particular field with a specified type is available in the document. These operators are used with either of the two values: true or false. If these operators are used with the true value, the query will return all the documents that include the specified field or the specified field with the specified datatype. If used with the false value, the query will return all the documents that do not include the specified field or the specified field with the specified datatype.

MongoDB supports two element operators:

- \$exists
- \$type

\$exists Operator

The \$exists operator checks if the specified field is present in a document. If it is used with the true value, \$exists returns the documents that contain the specified field, including documents where the field value is null. If \$exists is used with the false value, \$exists returns only the documents that do not contain the specified field.

The syntax for this operator is:

```
{ field: { $exists: <boolean> } }
```

In this syntax, <Boolean> takes two values: true or false.

Before understanding how this operator works, let us create a database named `Product_detail` and a collection named `product_delivery`. Let us also insert the five documents into the `product_delivery` collection using the query as:

```
db.product_delivery.insertMany(  
  [ { "_id" : 1, address : "2130 Messy Way", zipCode :  
    "90698345", delivered:["Documents","Food","electronics",  
    "Books"],Feedback:[{"product":"xyz","score":6}, {"product":"abc"  
    ,"score":7} ] ,  
    { "_id" : 2, address: "156 Lunar way garden",  
delivered:["Food","electronics"],Feedback:[ {"product":"xyz", "s  
core":5}, {"product":"abc", "score":4} ] ,  
      { "_id" : 3, address : "456 Penny way Palace", zipCode:  
3921412,  
delivered:["electronics"],Feedback:[ {"product":"xyz", "score":8  
}, {"product":"abc", "score":9} ] ,  
      { "_id" : 4, address : "155 Solar Ring  
road",delivered:["Books"],Feedback:[ {"product":"xyz", "score":5  
}, {"product":"abc", "score":2} ] ,  
      { "_id" : 5, address : "134 Julie Garden", zipCode :  
["834847278",  
"1893289032"],delivered:["Food","electronics"],Feedback:[ {"pro  
duct":"xyz", "score":4}, {"product":"abc", "score":8} ] }  
  ] )
```

Figure 3.9 shows the output of this query.

```
Product_detail> db.product_delivery.insertMany(  
... [  
...   { "_id" : 1, address : "2130 Messy Way", zipCode : "90698345", delivered:["Documents","Food","electronics", "books"],Feedback:[{"product":"xyz","score":6}, {"product":"abc","score":7}] },  
...   { "_id" : 2, address: "156 Lunar way garden", delivered:["Food", "electronics"],Feedback:[{"product":"xyz","score":5}, {"product":"abc","score":4}] },  
...   { "_id" : 3, address : "456 Penny way Palace", zipCode: 3921412, delivered:["electronics"],Feedback:[{"product":"xyz","score":8}, {"product":"abc","score":9}] },  
...   { "_id" : 4, address : "155 Solar Ring road",delivered:["Books"],Feedback:[{"product":"xyz","score":5}, {"product":"abc","score":2}] },  
...   { "_id" : 5, address : "134 Julie Garden", zipCode : ["834847278", "1893289032"],delivered:["Food", "electronics"],Feedback:[{"product":"xyz","score":4}, {"product":"abc","score":8}] }  
... ]  
... )  
{  
  acknowledged: true,  
  insertedIds: { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }  
}
```

Figure 3.9: Output of the insertMany Query

Now, consider that from the product_delivery collection, the user wants to view documents that do not have a zip code. To do, so, the user can execute a query that uses the \$exists operator as:

```
db.product_delivery.find({"zipCode":{$exists:false}})
```

Figure 3.10 shows the output of this query.

```
Product_detail> db.product_delivery.find({"zipCode":{$exists:false}})  
[  
  {  
    _id: 2,  
    address: '156 Lunar way garden',  
    delivered: [ 'Food', 'electronics' ],  
    Feedback: [ { product: 'xyz', score: 5 }, { product: 'abc', score: 4 } ]  
  },  
  {  
    _id: 4,  
    address: '155 Solar Ring road',  
    delivered: [ 'Books' ],  
    Feedback: [ { product: 'xyz', score: 5 }, { product: 'abc', score: 2 } ]  
  }  
]
```

Figure 3.10: Output of the Query with \$exists Operator

\$type Operator

This operator returns the documents where the datatype of the specified field in the document matches the specified datatype. The \$type operator is quite useful when dealing with highly unstructured data where the collection is complex and the prediction of the datatypes of any field is difficult.

The syntax for this operator is:

```
{ field: { $type: <BSON type> } }
```

The \$type operator accepts aliases and numbers in place of the datatype. Table 3.2 lists the datatype and its corresponding number and alias.

Datatype	Number	Alias
Double	1	"double"
String	2	"string"
Object	3	"object"
Array	4	"array"
Binary data	5	"binData"
ObjectId	7	"objectId"
Boolean	8	"bool"
Date	9	"date"
Null	10	"null"
Regular Expression	11	"regex"
JavaScript	13	"javascript"
32-bit integer	16	"int"
Timestamp	17	"timestamp"
64-bit integer	18	"long"
Decimal128	19	"decimal"
Min key	-1	"minKey"
Max key	127	"maxKey"

Table 3.2: Datatypes and their Corresponding Numbers and Aliases

Now, consider that from the product_delivery collection, the user wants to view the zipCode fields which have the string type. To do so, the user can execute a query that uses the \$type operator as:

```
db.product_delivery.find( { "zipCode" : { $type : 2}})
```

Figure 3.11 shows the output of the query.

```
Product_detail> db.product_delivery.find( { "zipCode" : { $type : 2}})
[
  {
    _id: 1,
    address: '2130 Messy Way',
    zipCode: '90698345',
    delivered: [ 'Documents', 'Food', 'electronics', 'Books' ],
    Feedback: [ { product: 'xyz', score: 6 }, { product: 'abc', score: 7 } ]
  },
  {
    _id: 5,
    address: '134 Julie Garden',
    zipCode: [ '834847278', '1893289032' ],
    delivered: [ 'Food', 'electronics' ],
    Feedback: [ { product: 'xyz', score: 4 }, { product: 'abc', score: 8 } ]
  }
]
```

Figure 3.11: Output of the Query with \$type Operator

Now, consider that from the `product_delivery` collection, the user wants to view the `zipCode` fields which are of the string type. To do so, the user can execute a query that uses the `$type` operator as:

```
db.product_delivery.find( { "zipCode" : { $type : 16}})
```

Figure 3.12 shows the output of the query.

```
Product_detail> db.product_delivery.find( { "zipCode" : { $type : 16}})
[
  {
    _id: 3,
    address: '456 Penny way Palace',
    zipCode: 3921412,
    delivered: [ 'electronics' ],
    Feedback: [ { product: 'xyz', score: 8 }, { product: 'abc', score: 9 } ]
  }
]
```

Figure 3.12: Output of the Query with \$type Operator

3.2.4 Array Operators

Array operators are used when the user wants to filter data by specifying conditions for array fields.

There are three array operators:

\$all

Used to retrieve, update, or delete documents that have an array field that has all the elements specified in the query

\$elemMatch

Used to retrieve, update, or delete documents that have an array field that has at least one of the elements specified in the query

\$size

Used to retrieve, update, or delete documents that have an array field with the same size as the size specified in the query

\$all Operator

The syntax for this operator is:

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

For example, consider that from the product_delivery collection, the user wants to view the documents that include the value 'Food' and 'electronics' in the delivered field. To do so, the user can execute a query that uses the \$all operator as:

```
db.product_delivery.find({delivered:  
{$all:['Food', 'electronics']} })
```

Figure 3.13 shows the output of the query.

```
Product_detail> db.product_delivery.find({delivered: {$all:['Food', 'electronics']}})
[
  {
    _id: 1,
    address: '2130 Messy Way',
    zipCode: '90698345',
    delivered: [ 'Documents', 'Food', 'electronics', 'Books' ],
    Feedback: [ { product: 'xyz', score: 6 }, { product: 'abc', score: 7 } ]
  },
  {
    _id: 2,
    address: '156 Lunar way garden',
    delivered: [ 'Food', 'electronics' ],
    Feedback: [ { product: 'xyz', score: 5 }, { product: 'abc', score: 4 } ]
  },
  {
    _id: 5,
    address: '134 Julie Garden',
    zipCode: [ '834847278', '1893289032' ],
    delivered: [ 'Food', 'electronics' ],
    Feedback: [ { product: 'xyz', score: 4 }, { product: 'abc', score: 8 } ]
  }
]
```

Figure 3.13: Output of the Query with \$all Operator

\$elemMatch Operator

The syntax for this operator is:

```
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
```

For example, consider that from the `product_delivery` collection, the user wants to view the documents that have the value `product: "xyz"` or `score` greater than `8` in the `Feedback` field. To do so, the user can execute a query that uses the `$elemMatch` operator as:

```
db.product_delivery.find( { Feedback: { $elemMatch:
{ product: "xyz", score: { $gte: 8} } } })
```

Figure 3.14 shows the output of the query.

```
Product_detail> db.product_delivery.find( { Feedback: { $elemMatch: { product: "xyz", score: { $gte: 8} } } })  
[  
  {  
    _id: 3,  
    address: '456 Penny way Palace',  
    zipCode: 3921412,  
    delivered: [ 'electronics' ],  
    Feedback: [ { product: 'xyz', score: 8 }, { product: 'abc', score: 9 } ]  
  }  
]
```

Figure 3.14: Output of the Query with \$elemMatch Operator

\$size Operator

The syntax for this operator is:

```
{ <field>: { $size: n} }
```

For example, consider that from the `product_delivery` collection, the user wants to view the documents that have four products in the `delivered` array. To do so, the user can execute a query that uses the `$size` operator as:

```
db.product_delivery.find({delivered:{$size:4}})
```

Figure 3.15 shows the output of the query.

```
Product_detail> db.product_delivery.find({delivered:{$size:4}})  
[  
  {  
    _id: 1,  
    address: '2130 Messy Way',  
    zipCode: '90698345',  
    delivered: [ 'Documents', 'Food', 'electronics', 'Books' ],  
    Feedback: [ { product: 'xyz', score: 6 }, { product: 'abc', score: 7 } ]  
  }  
]
```

Figure 3.15: Output of the Query with \$size Operator

3.3 Projection Operators

The `find()` operator in MongoDB returns all the data in the collection that matches the specified condition. What if the user wants to view specific fields of data from the collection? This is where project operators come in handy.

Projection operators can be used to specify the fields or elements to display in the query result based on the condition specified in the query.

Four projection operators in MongoDB are:

\$	Used to return the first element in an array that matches the condition specified in the query or the first element in the array, if no query condition is specified
\$elemMatch	Used to return the first element in an array that matches the condition specified by the \$elemMatch operator
\$meta	Used to retrieve the metadata related to a document in a collection
\$slice	Used to specify the number of elements that must be returned by the query

\$ Operator

The syntax for this operator is:

```
db.collection.find( { <array>: <condition> ... },  
                    { "<array>.$": 1 } )
```

To understand how this operator can be used, let us create a database named `student_detail` and switch to that database as shown in Figure 3.16.

```
sample_analytics> use Student_detail  
switched to db Student_detail
```

Figure 3.16: Switching to the `Student_detail` Database

Now, let us create a collection named `Studentmarks` in the database. Next, let us insert documents into the `Studentmarks` collection using the query as:

```
db.Studentmarks.insertMany([
    { name: "Robert", age: 16, sub:1, marks:[89,78,90]}, 
    { name: "Oliver", age: 17, sub:1, marks:[78,85,89]}, 
    { name: "Henry", age: 15, sub:1, marks:[88,89,76]}, 
    { name: "David", age: 16, sub:2, marks:[86,84,66]}])
```

Figure 3.17 shows the output of the query.

```
Student_detail> db.Studentmarks.insertMany([
...     { name: "Robert", age: 16, sub:1, marks:[89,78,90]}, 
...     { name: "Oliver", age: 17, sub:1, marks:[78,85,89]}, 
...     { name: "Henry", age: 15, sub:1, marks:[88,89,76]}, 
...     { name: "David", age: 16, sub:2, marks:[86,84,66]}])
{
    acknowledged: true,
    insertedIds: {
        '0': ObjectId("64525fc59a2bc07eed14823d"),
        '1': ObjectId("64525fc59a2bc07eed14823e"),
        '2': ObjectId("64525fc59a2bc07eed14823f"),
        '3': ObjectId("64525fc59a2bc07eed148240")}
```

Figure 3.17: Output of the `insertMany` Query

Now consider that the user wants to view the first element that is greater than or equal to 85 for the `marks` field, where the value of `sub` is 1. The user also wants to view the `name` field and hide the `_id` field. To do so, the user can execute a query that uses the `$` projection operator as:

```
db.Studentmarks.find( { sub: 1, marks:
{ $gte: 85 }
}, {_id:0, name:1, "marks.$": 1} )
```



In this query, `"marks.$": 1` ensures that the `marks` column is displayed and `_id:0` ensures that the `ID` column is hidden in the query result.

Figure 3.18 shows the output of this query.

```
Student_detail> db.Studentmarks.find( { sub: 1, marks: { $gte: 85 } }, { _id:0, name:1, "marks.$": 1 } )
[
  { name: 'Robert', marks: [ 89 ] },
  { name: 'Oliver', marks: [ 85 ] },
  { name: 'Henry', marks: [ 88 ] }
]
```

Figure 3.18: Output of the Query with \$ Operator

\$slice Operator

The syntax for this operator is:

```
db.collection.find( <query>,
{ <arrayField>: { $slice: <number> } } );
```

For example, consider that the user wants to view the first two elements of the marks field from the document where the name field has the value “Robert”. To do so, the user can execute a query that uses the \$slice operator as:

```
db.Studentmarks.find({ "name": "Robert" },
{ "marks": { $slice: 2 }})
```

Figure 3.19 shows the output of this query.

```
Student_detail> db.Studentmarks.find({ "name": "Robert" }, { "marks": { $slice: 2 } })
[
  {
    _id: ObjectId("64525fc59a2bc07eed14823d"),
    name: 'Robert',
    age: 16,
    sub: 1,
    marks: [ 89, 78 ]
  }
]
Student_detail>
```

Figure 3.19: Output of the Query with \$slice Operator

3.4 Update Operators

Update operators help users modify documents in a collection. There are separate update operators for modifying fields and arrays.

3.4.1 Field Update Operators

There are nine operators in MongoDB that allow updating the values of fields in the documents:

\$currentDate

- Used when the value in a field must be set to the current date

\$inc

- Used when the values in a field must be incremented by a specific number

\$min

- Used when the values in a field must be updated only if they are more than a specific value

\$max

- Used when the values in a field must be updated only if they are less than a specific value

\$mul

- Used when the values in a field must be multiplied by a specific value

\$rename

- Used when a field must be renamed

\$set

- Used when the value of a field in the document must be set

\$unset

- Used when a specific field must be removed from the document

\$setOnInsert

- Used when the value of a field must be set only if a new document is inserted as a result of the update operation

\$currentDate Operator

The syntax for this operator is:

```
{ $currentDate: { <field1>: <typeSpecification1>, ... } }
```

In this syntax, `<typeSpecification>` can be set to `true` if the user wants to input the current date in the `Date` type. If the user wants to insert the current date as a timestamp or a date type, then the user can set `<typeSpecification>` to `timestamp` or `date`, respectively. Note that these values must be used in lower case.



If the field specified in the query does not exist in the document, then this operator will add the field to the document.

For example, consider that the user wants to set the `Exam_Date` field in the `Studentmarks` collection to the current date as a Date. To do so, the user can execute and update query that uses the `$currentDate` operator as:

```
db.Studentmarks.updateOne({"name": "David"},  
{$currentDate: {Exam_Date: true}})
```

Figure 3.20 shows the output of this query.

```
Student_detail> db.Studentmarks.updateOne({"name": "David"}, {$currentDate: {Exam_Date: true}})  
{  
  acknowledged: true,  
  insertedId: null,  
  matchedCount: 1,  
  modifiedCount: 1,  
  upsertedCount: 0  
}
```

Figure 3.20: Output of the Query with `$currentDate` Operator

To view the documents in the updated `Studentmarks` collection, the user can execute the query as:

```
db.Studentmarks.find()
```

Figure 3.21 shows the output of this query.

```
Student_detail> db.Studentmarks.find()
[
  {
    _id: ObjectId("646f510b92732287e845d80d"),
    name: 'Robert',
    age: 16,
    sub: 1,
    marks: [ 89, 78, 90 ]
  },
  {
    _id: ObjectId("646f510b92732287e845d80e"),
    name: 'Oliver',
    age: 17,
    sub: 1,
    marks: [ 78, 85, 89 ]
  },
  {
    _id: ObjectId("646f510b92732287e845d80f"),
    name: 'Henry',
    age: 15,
    sub: 1,
    marks: [ 88, 89, 76 ]
  },
  {
    _id: ObjectId("646f510b92732287e845d810"),
    name: 'David',
    age: 16,
    sub: 2,
    marks: [ 86, 84, 66 ],
    Exam_Date: ISODate("2023-05-25T12:14:19.762Z")
  }
]
```

Figure 3.21: Updated Studentmarks Collection

\$inc Operator

The syntax for this operator is:

```
{ $inc: { <field1>: <amount1>, <field2>:
          <amount2>, ... } }
```

For example, consider that the user wants to increment the value in the age field by 2 where the name is "Henry". To do so, the user can execute a query that uses the \$inc operator as:

```
db.Studentmarks.updateOne( { name: "Henry" },
{ $inc: { age: 2 } })
```

Figure 3.22 shows the output of this query.

```
Student_details> db.Studentmarks.updateOne( { name: "Henry" }, { $inc: { age: 2 } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Figure 3.22: Output of the Query with \$inc Operator

To view the updated document, the user can execute the query as:

```
db.Studentmarks.find({ "name": "Henry" })
```

Figure 3.23 shows the output of this query.

```
Student_details> db.Studentmarks.find({ "name": "Henry" })
[
  {
    _id: ObjectId("646f55e160c2b086a2c39e27"),
    name: 'Henry',
    age: 17,
    sub: 1,
    marks: [ 88, 89, 76 ]
  }
]
```

Figure 3.23: Updated Document

\$set Operator

The syntax for this operator is:

```
{ $set: { <field1>: <value1>, ... } }
```

For example, consider that the user wants to add a new field named `credit` to the `Studentmarks` collection. To do this, the user can execute an update query that uses the `$set` operator as:

```
db.Studentmarks.updateMany( {}, [ { $set: { credit: 0 } } ] )
```

Figure 3.24 shows the output of this query.

```
Student_details> db.Studentmarks.updateMany( {}, [ { $set: { credit:0 } } ] )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 4,
  modifiedCount: 4,
  upsertedCount: 0
}
```

Figure 3.24: Output of the Query with \$set Operator

To view the updated document, the user can execute the query as:

```
db.Studentmarks.find()
```

Figure 3.25 shows the output of this query.

```
Student_details> db.Studentmarks.find()
[
  {
    _id: ObjectId("646f5c0760c2b086a2c39e29"),
    name: 'Robert',
    age: 16,
    sub: 1,
    marks: [ 89, 78, 90 ],
    credit: 0
  },
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2a"),
    name: 'Oliver',
    age: 17,
    sub: 1,
    marks: [ 78, 85, 89 ],
    credit: 0
  },
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2b"),
    name: 'Henry',
    age: 17,
    sub: 1,
    marks: [ 88, 89, 76 ],
    credit: 0
  },
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2c"),
    name: 'David',
    age: 16,
    sub: 2,
    marks: [ 86, 84, 66 ],
    Exam_Date: ISODate("2023-05-25T13:02:05.079Z"),
    credit: 0
  }
]
```

Figure 3.25: Updated Document

\$rename Operator

The syntax for this operator is:

```
{$rename: { <field1>: <newName1>, <field2>:  
          <newName2>, ... } }
```



The new field name must be different from the existing field name.

For example, consider that in the `Studentmarks` collection the user wants to change the name of `credit` field to `unit`. To do this, the user can execute an update query that uses the `$rename` operator as:

```
db.Studentmarks.updateOne( { name: "David" },  
                           { $rename: { 'credit': 'unit' } })
```

Figure 3.26 shows the output of this query.

```
Student_detail> db.Studentmarks.updateOne( { name: "David" }, { $rename: { 'credit': 'unit' } })  
{  
  acknowledged: true,  
  insertedId: null,  
  matchedCount: 1,  
  modifiedCount: 1,  
  upsertedCount: 0  
}
```

Figure 3.26: Output of the Query with \$rename Operator

To view the updated document, the user can execute the query as:

```
db.Studentmarks.find( { "name": "David" } )
```

Figure 3.27 shows the output of this query.

```
Student_details> db.Studentmarks.find({"name": "David"})
[{"_id": ObjectId("646f5c0760c2b086a2c39e2c"),
  "name": "David",
  "age": 16,
  "sub": 2,
  "marks": [ 86, 84, 66 ],
  "Exam_Date": ISODate("2023-05-25T13:02:05.079Z"),
  "unit": 0}
```

Figure 3.27: Updated Document

3.4.2 Array Update Operators

The array update operators supported by MongoDB are:

\$	• Used to update the first element in the specified array that matches the condition specified in the query
\$[]	• Used to update all the elements in the specified array for all the documents that match the condition specified in the query
\$addToSet	• Used to add array elements only if they are not present already
\$pop	• Used when the first or last item has to be removed from an array
\$pull	• Used when all the array elements that match the specified query condition has to be removed
\$push	• Used when elements must be added to an array
\$pullAll	• Used when all the array values that match the value specified in the query has to be removed

\$pop Operator

The syntax for this operator is:

```
{ $pop: { <field>: <-1 | 1>, ... } }
```

In this syntax, `-1` is used to remove the first element and `1` is used to remove the last element in the array.

For example, consider that in the `Studentmarks` collection, the user wants to remove the first element from the `marks` array where `name` is "Oliver". To do so, the user can execute a query that uses the `$pop` operator as:

```
db.Studentmarks.updateOne( { "name": "Oliver" }, { $pop: { marks: -1 } } )
```

Figure 3.28 shows the output of this query.

```
Student_detail> db.Studentmarks.updateOne( { "name": "Oliver" }, { $pop: { marks: -1 } } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Figure 3.28: Output of Query with \$pop Operator

To view the updated document, the user can execute the query as:

```
db.Studentmarks.find( { "name": "Oliver" } )
```

Figure 3.29 shows the output of this query.

```
Student_details> db.Studentmarks.find( { "name": "Oliver" } )
[
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2a"),
    name: 'Oliver',
    age: 17,
    sub: 1,
    marks: [ 85, 89 ],
    credit: 0
  }
]
```

Figure 3.29: Updated Document

\$pull Operator

The syntax for this operator is:

```
{ $pull: { <field1>: <value|condition>,
           <field2>: <value|condition>, ... } }
```

Consider that in the Studentmarks collection, the user wants to remove the specific element from the marks array where name is "Oliver". To do so, the user can execute a query that uses the \$pull operator as:

```
db.Studentmarks.updateOne({ name: "Oliver" },
{ $pull: { marks: { $in: [85] } } })
```

Figure 3.30 shows the output of this query.

```
Student_detail> db.Studentmarks.updateOne({ name: "Oliver" }, { $pull: { marks: { $in: [85] } } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Figure 3.30: Output of Query with \$pull Operator

To view the updated document, the user can execute the query as:

```
db.Studentmarks.find({ "name": "Oliver" })
```

Figure 3.31 shows the output of this query.

```
Student_details> db.Studentmarks.find({ "name": "Oliver" })
[
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2a"),
    name: 'Oliver',
    age: 17,
    sub: 1,
    marks: [ 89 ],
    credit: 0
  }
]
```

Figure 3.31: Updated Document

\$push Operator

The syntax for this operator is:

```
{ $push: { <field1>: <value1>, ... } }
```

For example, consider that in the `Studentmarks` collection, the user wants to add a value to the `marks` array in the document where `name` is "Oliver". To do so, the user can execute a query that uses the `$push` operator as:

```
db.Studentmarks.updateOne( { "name": "Oliver" }, { $push: { marks: 99 } } )
```

Figure 3.32 shows the output of this query.

```
Student_detail> db.Studentmarks.updateOne( { "name": "Oliver" }, { $push: { marks: 99 } } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Figure 3.32: Output of Query with \$push Operator

To view the updated document, the user can execute the query as:

```
db.Studentmarks.find({ "name": "Oliver" })
```

Figure 3.33 shows the output of this query.

```
Student_details> db.Studentmarks.find({ "name": "Oliver" })
[
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2a"),
    name: 'Oliver',
    age: 17,
    sub: 1,
    marks: [ 89, 99 ],
    credit: 0
  }
]
```

Figure 3.33: Updated Document

3.5 Other Operators

Three other operators that are commonly used in MongoDB are:

\$comment

- Used to add comments to a query predicate to explain what the query is intended to do

\$rand

- Used to generate a random floating number between 0 and 1

\$natural

- Used to specify whether the documents must be iterated through in the natural order or the reverse order; takes value 1 for natural order and -1 for reverse order

\$comment Operator

The syntax for this operator is:

```
db.collection.find( { <query>, $comment: <comment> } )
```

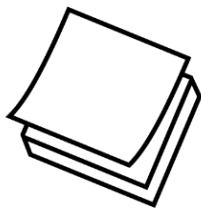
For example, consider that the user wants to add a comment for the document where the name is “Robert”. To do so, the user can execute a query that uses the \$comment operator as:

```
db.Studentmarks.find({ "name": "Robert",  
$comment: "Displaying Robert mark details"})
```

Figure 3.34 shows the output of this query.

```
Student_details> db.Studentmarks.find({ "name": "Robert", $comment: "Displaying Robert mark details"})  
[  
 {  
 _id: ObjectId("646f5c0760c2b086a2c39e29"),  
 name: 'Robert',  
 age: 16,  
 sub: 1,  
 marks: [ 89, 78, 90 ],  
 credit: 0  
 }  
]
```

Figure 3.34: Output of the Query with \$comment Operator



The comment is not visible in Figure 3.34. This is because the comments are stored in the `system.profile` collection. This collection is automatically created in the MongoDB server if database profiling is enabled. To view the comment, user can use the `find()` query on the `system.profile` collection.

\$rand Operator

The syntax for this operator is:

```
{ $rand: {} }
```

For example, consider that in the `Studentmarks` collection, the user updates the field named `unit`. The value in this field will be automatically generated by the `$rand` operator. The value generated should be multiplied by 100 and rounded off to the highest integer. The query to achieve this objective is:

```
db.Studentmarks.updateMany( {}, [ { $set: { unit: { $ceil: { $multiply: [{ $rand: {} }, 100] } } } } ] )
```

The empty update filter matches every document in the collection. Figure 3.35 shows the output of this query:

```
Student_detail> db.Studentmarks.updateMany( {}, [ { $set: { unit: { $ceil: { $multiply: [{ $rand: {} }, 100] } } } } ] )  
{ acknowledged: true,  
  insertedId: null,  
  matchedCount: 4,  
  modifiedCount: 4,  
  upsertedCount: 0  
}
```

Figure 3.35: Output of the Query with \$rand Operator

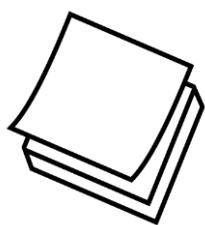
To view the updated documents, the user can execute the query:

```
db.Studentmarks.find()
```

Figure 3.36 shows the output of this query.

```
Student_details> db.Studentmarks.find()
[
  {
    _id: ObjectId("646f5c0760c2b086a2c39e29"),
    name: 'Robert',
    age: 16,
    sub: 1,
    marks: [ 89, 78, 90 ],
    credit: 0,
    unit: 55
  },
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2a"),
    name: 'Oliver',
    age: 17,
    sub: 1,
    marks: [ 89, 99 ],
    credit: 0,
    unit: 50
  },
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2b"),
    name: 'Henry',
    age: 17,
    sub: 1,
    marks: [ 88, 89, 76 ],
    credit: 0,
    unit: 21
  },
  {
    _id: ObjectId("646f5c0760c2b086a2c39e2c"),
    name: 'David',
    age: 16,
    sub: 2,
    marks: [ 86, 84, 66 ],
    Exam_Date: ISODate("2023-05-25T13:02:05.079Z"),
    unit: 37
  }
]
```

Figure 3.36: Output of the `find()` Query



The `unit` field is updated with the random number generated for all the documents in the collection. Note that the `unit` field in the document named "David" is updated with the random number as the `credit` field was renamed to `unit` using the `rename operator`. For all the other records, a new field named `unit` is added and updated with the random number.

3.6 Summary

- MongoDB offers multiple operators to allow users to create complex queries.
- Main types of operators in MongoDB are query, projection, and update operators.
- Query operators allow operations that are based on comparison (both field and array), logic, and existence of a field.
- Projection operators help to specify what fields of a document a query must return.
- Update operators allow modification of both the field and array data.

Test Your Knowledge

1. Which of the following comparison operator will allow you to list only the documents that contain a field whose value matches with any value in the specified array?
 - a. \$in
 - b. \$nin
 - c. \$ne
 - d. \$ni

 2. Consider that `marks` is a collection which consists of a field `score`. Which of the following option will you use to find only the score value of datatype string?
 - a. `db.marks.find({ "Score" : { $type : 1 } })`
 - b. `db.marks.find({ "Score" : { $type : 2 } })`
 - c. `db.marks.find({ "Score" : { $type : 16 } })`
 - d. `db.marks.find({ "Score" : { $type : 3 } })`

 3. Consider that `marks` is a collection that consists of a field `subject_marks` which is an array. Which of the following option will list documents with `subject_marks` field of size 3?
 - a. `db.subject_marks.find({marks:{$size:3}})`
 - b. `db.marks.find({subject_marks:{size:3}})`
 - c. `db.marks.find({"$subject_marks":{$size:3}})`
 - d. `db.marks.find({subject_marks:{$size:3}})`

 4. Which of the following projection operator will project the first element in an array that matches the query condition?
 - a. \$slice
 - b. \$first
 - c. \$
 - d. \$elementfirst

 5. Which of the following syntax allows you to remove the first element in the `marks` array?
 - a. { \$pop: { marks: -1 } }
 - b. { \$pop: { marks: 1 } }
 - c. { \$pop: { marks: 0 } }
 - d. { \$pop: { marks: \$first } }
-
-

Answers to Test Your Knowledge

1	a
2	b
3	d
4	c
5	a

Try it Yourself

1. Create a database named Student and a collection named Stud_mark.
2. Insert the following four documents into the stud_mark collection.

```
[  
    { name: "Adam",  
      gender:"M",  
      subjects:["Java", "C", "Python"],  
      marks:[89,78,90],  
      average:85.6  
    },  
    { name: "Franklin",  
      gender:"M",  
      subjects:["C", "VB", "Python"],  
      marks:[78,85,89],  
      average:84  
    },  
    { name: "Michael",  
      gender:"M",  
      subjects:["Java", "PHP"],  
      marks:[88,89],  
      average:88.5  
    },  
    { name: "Amelia",  
      gender:"F",  
      subjects:["Ruby", "C++"],  
      marks:[86,87],  
      average: 86.5  
    }  
]
```

Using the collection stud_mark perform the following tasks:

3. Find only the documents where the average value is equal to 84.
4. Find only the documents where the average value is greater than 85.
5. Display only the documents where the subjects array contains either Java or C++.
6. View only the documents where the average is greater than or equal to 87 and average is less than or equal to 90.
7. View all the documents where the subjects array has the value Java.
8. Display only the documents where the first element in the marks array is less than 80.

9. Display the details of the student named Adam where the marks array has only the first element and the second element.
10. Add a new date field Date_of_exam which shows the current date only for the student named "Amelia".
11. Increase the average value by 2 for the student named "Franklin".
12. Rename the field Date_of_exam as Examination_date.



SESSION 4

AGGREGATION PIPELINE

Learning Objectives

In this session, students will learn to:

- Explain aggregation pipeline in MongoDB
- Describe the stages in the aggregation pipeline
- Explain the expressions that can be used in the aggregation pipeline

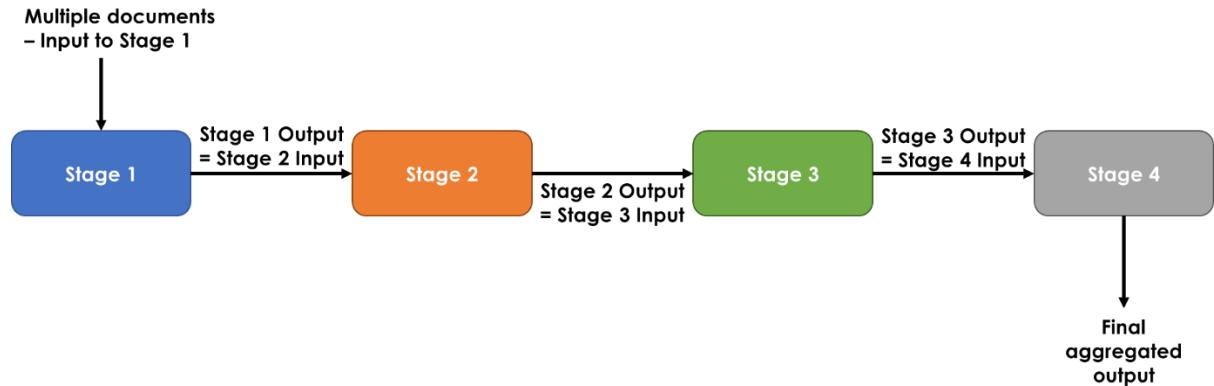
Aggregation in MongoDB helps in grouping multiple documents, applying some processes on them, and returning a single result. This helps in getting useful insights that aid in business decisions. One of the methods of performing data aggregation in MongoDB is the aggregation pipeline.

This session will explain what aggregation pipeline in MongoDB is. It will discuss various stages in the aggregation pipeline. It will also explain the various expressions that can be used with the aggregation pipeline.

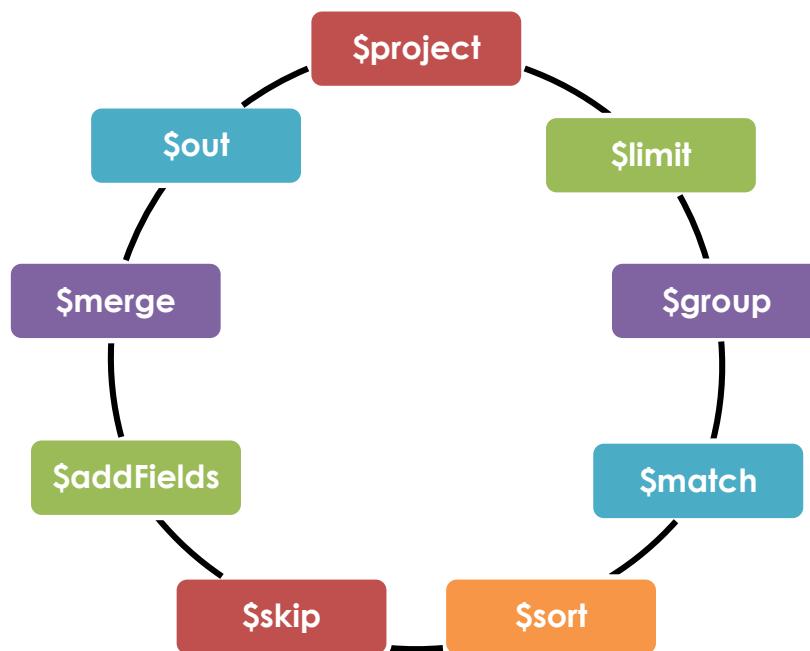
4.1 Aggregation Pipeline Stages

In the context of MongoDB, a pipeline is nothing, but a series of stages involved in data processing. In the pipeline, the first stage takes documents from the database as input, performs one or more calculations to produce a result. This

result is then passed as input to the next stage in the pipeline. This continues until the last stage of pipeline, which finally delivers the desired aggregated output.



The pipeline stages exist in an array in the `db.collection.aggregate` method. The aggregation pipeline can consist of many stages. This session will discuss the most important nine stages of the aggregation pipeline:



4.1.1 \$project Stage

The `$project` stage collects the required documents, manipulates the documents by adding new fields, deleting existing fields, or adding computed fields. It then passes these modified documents as input to the next stage. The format of `$project` is:

```
{$project: {<specification(s)>} }
```

In this format, <specification (s)> can take one of the values as given in Table 4.1.

Value	Description	Example
field: 0 or false	The field is not included in the result.	{_id: 0} The default field, _id is not included in the result.
field: 1 or true	The field is included in the result.	{certificate_number: 1} The certificate_number field is included in the result.
new field: expression	A new field with the specified expression is included in the result.	{grade: 'Grade field'} A new field called grade is included in the result with value as Grade field.

Table 4.1: Values of <specification(s)> in the \$project Stage

For example, consider that the user wants to aggregate data present in the inspections collection from the sample_training database. To view the existing data in the inspections collection, the user runs the query as:

```
db.inspections.find()
```

Figure 4.1 shows the output of this query.

```
{  
  _id: ObjectId("56d61033a378eccde8a83550"),  
  id: '10057-2015-ENFO',  
  certificate_number: 6007104,  
  business_name: 'LD BUSINESS SOLUTIONS',  
  date: 'Feb 25 2015',  
  result: 'Violation Issued',  
  sector: 'Tax Preparers - 891',  
  address: {  
    city: 'NEW YORK',  
    zip: 10030,  
    street: 'FREDERICK DOUGLASS BLVD',  
    number: 2655  
  }  
},  
{  
  _id: ObjectId("56d61033a378eccde8a8354f"),  
  id: '10021-2015-ENFO',  
  certificate_number: 9278806,  
  business_name: 'ATLIXCO DELI GROCERY INC.',  
  date: 'Feb 20 2015',  
  result: 'No Violation Issued',  
  sector: 'Cigarette Retail Dealer - 127',  
  address: {  
    city: 'RIDGEWOOD',  
    zip: 11385,  
    street: 'MENAHAN ST',  
    number: 1712  
}
```

Figure 4.1: Data in the inspections Collection

Now, consider that the user wants to view only the certificate_number, business_name, and address fields. In this case, the user can use the \$project stage of the aggregation pipeline as:

```
db.inspections.aggregate( [ { $project : {  
  certificate_number : 1 , business_name : 1,  
  address:1 } } ] )
```



The _id field is displayed by default. So, the user must exclude this field by setting it to 0.

Figure 4.2 shows the output of the query.

```
sample_training> db.inspections.aggregate([{$project:{_id:0,certificate_number:1,business_name:1,address:1}}])
[
  {
    certificate_number: 6007104,
    business_name: 'LD BUSINESS SOLUTIONS',
    address: {
      city: 'NEW YORK',
      zip: 10030,
      street: 'FREDERICK DOUGLASS BLVD',
      number: 2655
    }
  },
  {
    certificate_number: 5381180,
    business_name: 'ERIC CONSTRUCTION AND DECORATING INC.',
    address: {
      city: 'STATEN ISLAND',
      zip: 10304,
      street: 'TODT HILL RD',
      number: 1233
    }
  },
  {
    certificate_number: 9278914,
    business_name: 'MICHAEL GOMEZ RANGHALL',
    address: {
      city: 'QUEENS VLG',
      zip: 11427,
      street: '214TH ST',
      number: 8823
    }
  }
]
```

Figure 4.2: Result of the \$project Stage

The output returns all the documents with the specified fields.

4.1.2 \$limit Stage

The `$limit` stage helps in limiting the number of documents that are passed to the next stage. The format of the `$limit` stage is:

```
{$limit: {<64-bit positive integer>} }
```

The number of documents that are passed to the next stage is controlled by the number specified in `{<64-bit positive integer>}` .

Consider the `inspections` collection. To limit the number of documents returned to two and display only specific fields, the `$limit` stage can be used with the `$project` stage as:

```
db.inspections.aggregate( [ { $limit : 2
}, { $project : { certificate_number : 1 ,
business_name : 1, address:1 } } ] )
```

Figure 4.3 shows the output of the query.

```
sample_training> db.inspections.aggregate( [ { $limit : 2 },{ $project : { certificate_number : 1 , business_name : 1, address:1 } } ] )
[ {
  _id: ObjectId("56d61033a378eccde8a83550"),
  certificate_number: 6007104,
  business_name: 'LD BUSINESS SOLUTIONS',
  address: {
    city: 'NEW YORK',
    zip: 10030,
    street: 'FREDERICK DOUGLASS BLVD',
    number: 2655
  }
},
{
  _id: ObjectId("56d61033a378eccde8a83553"),
  certificate_number: 5381180,
  business_name: 'ERIC CONSTRUCTION AND DECORATING INC.',
  address: {
    city: 'STATEN ISLAND',
    zip: 10304,
    street: 'TOOT HILL RD',
    number: 1233
  }
]
]
```

Figure 4.3: Result of the \$limit Stage

4.1.3 \$group Stage

The \$group stage helps in grouping the documents based on a field or key or group of fields. The format of the \$group stage is:

```
{$group: {_id: <expression>, //Group Key
<field1>: {<accumulator1>: <expression1>}, ... }}
```

Parameters in this format are described in Table 4.2.

Specification Form	Description
_id: <expression>	Specifies the field based on which the data must be grouped
<field1>	Specifies the name of the field whose values must be used for computation
<accumulator1>	Specifies the function to be applied on the field specified in <field1>

Table 4.2: Parameters of \$group stage

The <accumulator1> parameter can specify one of the functions listed in Table 4.3.

Function	Description
\$avg	Used to calculate the average value of the group
\$bottom	Used to retrieve the bottom element in the group based on the order in which the elements are sorted
\$bottomN	Used to retrieve the bottom N element in the group based on the order in which the elements are sorted
\$top	Used to retrieve the top element of the group based on the order in which the elements are sorted
\$topN	Used to retrieve the top N element in the group based on the order in which the elements are sorted
\$min	Used to retrieve the element with the lowest value in the group
\$minN	Used to retrieve N elements with the lowest value in the group
\$max	Used to retrieve the element with the highest value in the group
\$mergeObjects	Returns a document created by combining the input documents for each group
\$first	Used to retrieve the value from the first document in each group
\$last	Used to retrieve the value from the last document in each group
\$count	Used to know the number of documents in each group
\$sum	Used to calculate the sum of the numerical values of a field in a group

Table 4.3: Accumulator Functions

For example, consider that in the `inspections` collection, the user wants to group the document based on the `address.city` field and find the number of documents in each group. To do so, the user can run the query as:

```
db.inspections.aggregate( [ { $group : { _id : "$address.city", count: { $count: { } } } } ] )
```

Figure 4.4 shows the output of this query.

```
sample_training> db.inspections.aggregate( [ { $group : { _id : "$address.city", count: { $count: { } } } } ] )
[ { _id: 'UNION CITY', count: 4 },
{ _id: 'Elmhurst', count: 5 },
{ _id: 'L.I.C.', count: 1 },
{ _id: 'JACKSON HTS', count: 63 },
{ _id: 'LONG ISLAND CITY', count: 718 },
{ _id: 'FAIR LAWN', count: 1 },
{ _id: 'DOUGLASTON', count: 2 },
{ _id: 'BRONX', count: 12148 },
{ _id: 'WANTAGH', count: 1 },
{ _id: 'MOUNT VERNON', count: 4 },
{ _id: 'WOODHAVEN', count: 353 },
{ _id: 'QUEENS VILLAGE', count: 345 },
{ _id: 'PATERSON', count: 2 },
{ _id: 'ALBERTSON', count: 2 },
{ _id: 'SOUTH PLAINFIELD', count: 1 },
{ _id: 'BUFFALO', count: 1 },
{ _id: 'LITITZ', count: 1 },
{ _id: 'SOUTH OZONE PARK', count: 242 },
{ _id: 'FRANKLIN LAKES', count: 6 },
{ _id: 'GUTTENBERG', count: 2 }
]
```

Figure 4.4: Result of the \$count Accumulator in the \$group Stage

Consider that the user wants to limit the number of documents passed to the \$group stage to 5000. The user wants to group the resulting documents based on the address.city field. After doing so, the user wants to find the number of documents in each group. To do so, the user can use the query as:

```
db.inspections.aggregate( [{$limit:5000}, { $group : { _id : "$address.city", count: { $count: { } } } } ] )
```

Figure 4.5 shows the output of this query.

```
sample_training> db.inspections.aggregate( [{$limit:5000}, { $group : { _id : "$address.city", count: { $count: { } } } } ] )
[ { _id: 'WEEHAWKEN', count: 2 },
{ _id: 'ISELIN', count: 1 },
{ _id: 'FREEHOLD', count: 1 },
{ _id: 'UNION CITY', count: 3 },
{ _id: 'LAURELTON', count: 3 },
{ _id: 'Jackson Heights', count: 1 },
{ _id: 'JERSEY CITY', count: 86 },
{ _id: 'LINDEN', count: 1 },
{ _id: 'Highland', count: 1 },
{ _id: 'EDISON', count: 1 },
{ _id: 'RUTHERFORD', count: 1 },
{ _id: 'Richmond Hill', count: 1 },
{ _id: 'LINDENWOLD', count: 2 },
{ _id: 'Brooklyn', count: 1 },
{ _id: 'S RICHMOND HL', count: 5 },
{ _id: 'TOWNSHIP OF WASHINGTON', count: 1 },
{ _id: 'WEST HEMPSTEAD', count: 4 },
{ _id: 'BERGENFIELD', count: 1 },
{ _id: 'IRVINGTON', count: 1 },
{ _id: 'MASSAPEQUA PARK', count: 10 }
]
```

Figure 4.5: Result of \$group Stage With \$limit Stage

The user can also use the `$group` and `$limit` stage in the reverse order. That is, if the user wants to group all the documents based on the `address.city` field and count the number of documents in each group. If the user wants to view output only for 10 groups, the user can run the query as:

```
db.inspections.aggregate( [ { $group : { _id : "$address.city", count: { $count: { } } } }, {$limit:10 } ] )
db.inspections.aggregate( [ { $group : { _id : "$address.city", count: { $count: { } } } }, {$limit:20 } ]
)
```

Figure 4.6 shows the output of this query.

```
sample_training> db.inspections.aggregate( [ { $group : { _id : "$address.city", count: { $count: { } } } }, {$limit:10 } ] )
[ { _id: 'WADING RIVER', count: 6 },
{ _id: 'NEWTOWN', count: 1 },
{ _id: 'PLAINFIELD', count: 1 },
{ _id: 'GARFIELD', count: 1 },
{ _id: 'FLUSHING', count: 1513 },
{ _id: 'CORONA', count: 1128 },
{ _id: 'FAIRVIEW', count: 2 },
{ _id: 'WARWICK', count: 2 },
{ _id: 'MOUNT SINAI', count: 1 },
{ _id: 'INWOOD', count: 36 }
```

Figure 4.6: Result of `$group` Stage With `$limit` Stage



The output will change depending on the order in which the stages are used.

4.1.4 `$match` Stage

The `$match` stage in the aggregate pipeline helps in filtering documents based on a condition and passing the result to the next stage. The format of `$match` stage is:

```
{ $match: {<expression>} } ... }
```

The `$match` takes documents as input and filters the documents on the specified expression. Consider that in the `inspections` collection, the user wants to hide the `_id` field, view the `business_name` and `address.city` fields for documents where the `address.city` is JERSEY CITY. Also, the user wants to limit this output to 10 documents. To do so, the user can run the query as:

```
db.inspections.aggregate([{$project:{_id:0,"address.city":1,business_name: 1}}, {$match:{"address.city": "JERSEY CITY"}}, {$limit:10}]))
```

Figure 4.7 shows the output of this query.

```
sample_training> db.inspections.aggregate([{$project:{_id:0,"address.city":1,business_name: 1}}, {$match:{"address.city": "JERSEY CITY"}}, {$limit:10}])
[{"business_name": "AYAD YOUSSEF", "address": {"city": "JERSEY CITY"}}, {"business_name": "AZMY KIROLES", "address": {"city": "JERSEY CITY"}}, {"business_name": "MOHAMED AFIFI", "address": {"city": "JERSEY CITY"}}, {"business_name": "MAHMUD A MOHAMED ALI EID", "address": {"city": "JERSEY CITY"}}, {"business_name": "AHMED M EL SAKKAR", "address": {"city": "JERSEY CITY"}}, {"business_name": "EID AYOUB", "address": {"city": "JERSEY CITY"}}, {"business_name": "ROBERTO E. GONZALEZ", "address": {"city": "JERSEY CITY"}}, {"business_name": "WAHIM WAKIM", "address": {"city": "JERSEY CITY"}}, {"business_name": "ATEF ZAKI", "address": {"city": "JERSEY CITY"}}, {"business_name": "HASSAN AHMED", "address": {"city": "JERSEY CITY"}}]
```

Figure 4.7: Result of \$match Stage With \$limit and \$project Stages

4.1.5 \$sort Stage

As the name suggests, the \$sort stage sorts the documents in the ascending or descending order based on the specified field or fields. The sorted list of documents is then passed to the next stage. If the documents are to be sorted on multiple fields, they are followed in the order of left to right. The format of \$sort is:

```
{$$sort:{field1:<sort order>,field2:<sort order>,...}}}
```

Table 4.4 describes the parameters of the \$sort stage format.

Parameter	Description
field1, field2	Specifies the names of the fields based on which the documents must be sorted
sort order	Specifies the order in which the documents must be sorted. It takes the value 1 for ascending order and -1 for descending order

Table 4.4: Parameters of \$sort stage

For example, consider that in the `inspections` collection, the user wants to:

- Use 5000 documents
 - Hide the `_id` field
 - View the `business_name` and `address.city` fields
 - Sort the documents based first on the descending order of the `address.city` field and then on the ascending order of the `business_name` field
 - Output only 10 documents

To do so, the user can run the query as:

```
db.inspections.aggregate([{$project:{_id:0,"address.city":1, business_name: 1}}, {$limit: 5000}, {$sort:{"address.city": -1, "business name": 1}}, {$limit:10}])
```

Figure 4.8 shows the output of this query.

Figure 4.8: Result of \$sort Stage With \$limit and \$project Stages

In the output, the `address.city` field is sorted in descending order. So, all the city names in small letters are displayed first in descending order and then the city names in capital letters are displayed in descending order. Two records having the same city name are sorted in ascending order of the business names as specified.

4.1.6 \$skip Stage

The `$skip` stage is used to remove some of the documents that are passed to the next stage. The format of the `$skip` stage is:

`{$skip:{<64-bit positive integer}}}`

The number of documents that are passed to the next stage is controlled by the number specified in {<64-bit positive integer>} }.

Consider that the user wants to use the same query used for the `$sort` stage. In that query, the user wants to skip the first 10 documents instead of limiting the output to 10 documents. To do so, the user can run the query as:

```
db.inspections.aggregate([{$project:{_id:0,"address.city":1, business_name: 1}}, {$limit: 5000}, {$sort:{"address.city": -1, "business_name": 1}}, {$skip:10}])
```

Figure 4.9 shows the output of this query.

```
sample_training> db.inspections.aggregate([{$project:{_id:0,"address.city":1,business_name: 1}}, {$limit: 5000}, {$sort: {"address.city": -1, "business_name": 1}}, {$skip:10}])
[{"business_name": "PETROLEUM KINGS LLC", "address": { city: "YONKERS" }}, {"business_name": "RAMIREZ ALFREDO", "address": { city: "YONKERS" }}, {"business_name": "YAHYA EL SHAFFEY", "address": { city: "YONKERS" }}, {"business_name": "ABDUS SALAM", "address": { city: "WOODSIDE" }}, {"business_name": "AHERN MAINTENANCE & SUPPLY CORP", "address": { city: "WOODSIDE" }}, {"business_name": "AHRAMI, ABDUL", "address": { city: "WOODSIDE" }}, {"business_name": "AKOTA MEAT MARKET CO. INC.", "address": { city: "WOODSIDE" }}, {"business_name": "ALEM AZZEDDINE", "address": { city: "WOODSIDE" }}, {"business_name": "ASLAM, MUHAMMAD", "address": { city: "WOODSIDE" }}, {"business_name": "BALLOONS PARTY INC.", "address": { city: "WOODSIDE" }}, {"business_name": "BASHIR A QAYMI", "address": { city: "WOODSIDE" }}, {"business_name": "BHULU BHUIYAN", "address": { city: "WOODSIDE" }}}]
```

Figure 4.9: Result of \$skip Stage With \$limit, \$sort, and \$project Stages

In the output, the first documents where the values in the address.city field were in lower letters have been skipped.

4.1.7 \$addFields Stage

As the name suggests, the \$addFields stage is used to add a new field to the documents. It takes the documents passed from the previous stage, adds the new specified field to those documents, and passes the updated documents to the next stage. The format of \$addFields stage is as follows:

```
{ $addFields: {<new field1>:<expression1>... } }
```

In this syntax, <new field1> specifies the name for the new field and <expression1> specifies the value for the new field. The user can add as many fields as required using a single query.

Consider the inspections collection. The user wants to add a new field named address.country and assign the value USA for this new field. Then, the user wants to display the business_name, address.city, and address.country fields for 5 documents. To do so, the user can run the query as:

```
db.inspections.aggregate ([{$addFields: {"address.country": "USA"}}, {$limit:5}, {$project:{_id:0,business_name: 1, "address.city": 1, "address.country": 1}}])
```

Figure 4.10 shows the output of this query.

```
sample_training> db.inspections.aggregate ([{$addFields: {"address.country": "USA"}}, {$limit:5}, {$project:{_id:0,business_name: 1, "address.city": 1, "address.country": 1}}])
[
  {
    business_name: 'LD BUSINESS SOLUTIONS',
    address: { city: 'NEW YORK', country: 'USA' }
  },
  {
    business_name: 'ERIC CONSTRUCTION AND DECORATING INC.',
    address: { city: 'STATEN ISLAND', country: 'USA' }
  },
  {
    business_name: 'MICHAEL GOMEZ RANGHALL',
    address: { city: 'QUEENS VLG', country: 'USA' }
  },
  {
    business_name: 'UNNAMED HOT DOG VENDOR LICENSE NUMBER TA01158',
    address: { city: '', country: 'USA' }
  },
  {
    business_name: 'VYACHESLAV KANDZHANOV',
    address: { city: 'NEW YORK', country: 'USA' }
  }
]
```

Figure 4.10: Result of \$addFields Stage With \$limit, and \$project Stages

4.1.8 \$out Stage

The \$out stage is used to copy the query results to a collection. If the query results are written to an existing document, the \$out stage completely overwrites the query results to the collection without retaining any existing documents. The format for \$out stage is:

```
{ $out: { db: "<output-db>",  
coll: "<output-collection>" } }
```

In this format, db: "<output-db>" specifies the name of the database where the output collection exists and coll: "<output-collection>" specifies the name of the output collection.

For example, if the user wants to push eight documents from the inspections collection with the certificate_number, business_name, and address fields to a new collection named out_inspection1. To do so, the user can run the query as:

```
db.inspections.aggregate( [{$limit:8}, { $project : {  
certificate_number : 1 , business_name : 1,  
address:1 } }, { $out : "out_inspection1" } ] )
```

To view the documents in the out_inspection1 collection, the user can run the query as:

```
db.out_inspection1.find()
```

Figure 4.11 shows the output of this query.

```
sample_training> db.out_inspection1.find()
[  
  {  
    "_id": ObjectId("56d61033a378eccde8a83550"),  
    "certificate_number": 6007104,  
    "business_name": "LD BUSINESS SOLUTIONS",  
    "address": {  
      "city": "NEW YORK",  
      "zip": 10030,  
      "street": "FREDERICK DOUGLASS BLVD",  
      "number": 2655  
    }  
  },  
  {  
    "_id": ObjectId("56d61033a378eccde8a83553"),  
    "certificate_number": 5381180,  
    "business_name": "ERIC CONSTRUCTION AND DECORATING INC.",  
    "address": {  
      "city": "STATEN ISLAND",  
      "zip": 10304,  
      "street": "TODT HILL RD",  
      "number": 1233  
    }  
  },  
  {  
    "_id": ObjectId("56d61033a378eccde8a83551"),  
    "certificate_number": 9278914,  
    "business_name": "MICHAEL GOMEZ RANGHALL",  
    "address": {  
      "city": "QUEENS VLG",  
      "zip": 11427,  
      "street": "214TH ST",  
      "number": 8823  
    }  
  },  
  {  
    "_id": ObjectId("56d61033a378eccde8a83552"),  
    "certificate_number": 9278915,  
    "business_name": "MICHAEL GOMEZ RANGHALL",  
    "address": {  
      "city": "QUEENS VLG",  
      "zip": 11427,  
      "street": "214TH ST",  
      "number": 8823  
    }  
  },  
  {  
    "_id": ObjectId("56d61033a378eccde8a83554"),  
    "certificate_number": 9278916,  
    "business_name": "MICHAEL GOMEZ RANGHALL",  
    "address": {  
      "city": "QUEENS VLG",  
      "zip": 11427,  
      "street": "214TH ST",  
      "number": 8823  
    }  
  },  
  {  
    "_id": ObjectId("56d61033a378eccde8a83555"),  
    "certificate_number": 9278917,  
    "business_name": "MICHAEL GOMEZ RANGHALL",  
    "address": {  
      "city": "QUEENS VLG",  
      "zip": 11427,  
      "street": "214TH ST",  
      "number": 8823  
    }  
  },  
  {  
    "_id": ObjectId("56d61033a378eccde8a83556"),  
    "certificate_number": 9278918,  
    "business_name": "MICHAEL GOMEZ RANGHALL",  
    "address": {  
      "city": "QUEENS VLG",  
      "zip": 11427,  
      "street": "214TH ST",  
      "number": 8823  
    }  
  }]
```

Figure 4.11: Documents in the out_inspection1 Collection

The out_inspection1 collection consists of eight documents and the last two documents have the address.city as NEW YORK.

Consider that the user wants to skip the first 10 documents in the inspections collection. The user then wants to push three documents with the certificate_number, business_name, and address fields where address.city is NEW YORK to the out_inspection1 collection. To do so, the user can run the query as:

```
db.inspections.aggregate( [{$skip:10}, { $match: {  
  "address.city": "NEW YORK" } } ,{ $project : {  
    certificate_number : 1 , business_name : 1, address:1 }  
  },{$limit:3} { $out : "out_inspection" } ] )
```

To view the documents in the `out_inspection1` collection, the user can run the query as:

```
db.out_inspection1.find()
```

Figure 4.12 shows the output of this query.

```
sample_training> db.inspections.aggregate( [{$skip:10}, { $match: { "address.city": "NEW YORK" } } ,{ $project : { certificate_number : 1 , business_name : 1, address:1 } },{$limit: 3}, { $out : "out_inspection1" } ] )  
  
sample_training> db.out_inspection1.find()  
[  
  {  
    _id: ObjectId("56d61033a378eccde8a83567"),  
    certificate_number: 9302188,  
    business_name: 'EAGLE TILE & HOME CENTER, INC.',  
    address: { city: 'NEW YORK', zip: 10029, street: '2ND AVE', number: 2254 }  
  },  
  {  
    _id: ObjectId("56d61033a378eccde8a8356c"),  
    certificate_number: 5389246,  
    business_name: 'NADAL 2 DELI CONVENIENCE, INC.',  
    address: { city: 'NEW YORK', zip: 10031, street: 'BROADWAY', number: 3578 }  
  },  
  {  
    _id: ObjectId("56d61033a378eccde8a83575"),  
    certificate_number: 5393007,  
    business_name: 'BEST FOR LESS CONSTRUCTUIN',  
    address: {  
      city: 'NEW YORK',  
      zip: 10033,  
      street: 'AUDUBON AVE',  
      number: 215  
    }  
  }  
]
```

Figure 4.12: Documents in the `out_inspection1` Collection

Figure 4.12 shows three documents in `out_inspection1` collection instead of the original eight documents. This is because the `$out` stage has removed the earlier documents and replaced them with the documents from the last query.

4.1.9 `$merge` Stage

The `$merge` stage is similar to the `$out` stage and is used to store the output of the pipeline to a new collection in the same or different database. The difference is that `$merge` does not remove the earlier documents. It uses an identifier that uniquely identifies the document and when a match is found, it performs actions such as replacing existing document, keeping existing document, or merging the documents. If a match is not found, it performs actions such as inserting the document or discarding the document. The format of `$merge` is:

```

{ $merge: {
  into: <collection> -or- { db: <db>, coll: <collection> },
  on: <identifier field> -or- [ <identifier field1>, ... ],
  let: <variables>,
  whenMatched: <replace|keepExisting|merge|fail|pipeline>,
  whenNotMatched: <insert|discard|fail>
} }

```

Table 4.5 describes the parameters used in the `$merge` stage.

Parameter	Description
<code>into:</code>	Specifies the collection if the documents are to be pushed into a collection in the same database where the query is being run Otherwise, it specifies the database and collection to which the documents are to be pushed. If the specified collection does not exist, MongoDB creates a new collection.
<code>on:</code>	Specifies an identifier that uniquely identifies the documents It is used to identify if a document in the result matches a document in the output collection.
<code>whenMatched:</code>	Specifies the action to be taken if the identifier of a document in the output collection matches the identifier of a document in the query result It allows replacing the existing document, retaining the existing document, or merging the documents. When merging the documents, the fields that are missing in the output collection will be added. If the fields exist in both the documents, the values from the query results will replace the values in the output collection. If the identifier in the query results and output collection match, the merge operation can also be failed.

Parameter	Description
whenNotMatched:	<p>Specifies the action that must be taken if the identifier of a document in the output collection does not match the identifier of a document in the query result</p> <p>In this case,</p> <ul style="list-style-type: none"> • the document from the query result can be inserted into the output collection • the documents from the query result can be discarded, or • the merge operation can be failed.

Table 4.5: Parameters of \$merge stage

For example, consider that the user wants to push documents from the inspections collection into the out_inspection1 collection where address.city is HAZLET. If a match is found the documents must be replaced in the out_inspection1 collection. If a match is not found, the document must be inserted into the out_inspection1 collection. To do so, the user can run a query as:

```
db.inspections.aggregate( [{$skip:10}, { $match: {
  "address.city": "NEW YORK" }}, { $project : {
    certificate_number : 1 , business_name : 1, address:1 } },
  {$limit:3} { $out : "out_inspection" } ] )
```

To view the documents in the out_inpsection1 collection, the user can run the query as:

```
db.out_inspection1.find()
```

Figure 4.13 shows the output of this query.

```
sample_training> db.inspections.aggregate( [ { $match: { "address.city": "HAZLET" } } , { $pr
sample_training> db.inspections.aggregate( [ { $match: { "address.city": "HAZLET" } } , { $pr
object : { certificate_number : 1 , business_name : 1 , address:1 } } , { $merge : { into: {
db: "sample_training", coll: "out_inspection1" } , on: "_id", whenMatched: "replace", when
whenNotMatched: "insert" } } ] )
```

```
sample_training> db.out_inspection1.find()
[
  {
    _id: ObjectId("56d61033a378eccde8a83567"),
    certificate_number: 9302188,
    business_name: 'EAGLE TILE & HOME CENTER, INC.',
    address: { city: 'NEW YORK', zip: 10029, street: '2ND AVE', number: 2254 }
  },
  {
    _id: ObjectId("56d61033a378eccde8a8356c"),
    certificate_number: 5389246,
    business_name: 'NADAL 2 DELI CONVENIENCE, INC.',
    address: { city: 'NEW YORK', zip: 10031, street: 'BROADWAY', number: 3578 }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83575"),
    certificate_number: 5393007,
    business_name: 'BEST FOR LESS CONSTRUCTUIN',
    address: {
      city: 'NEW YORK',
      zip: 10033,
      street: 'AUDUBON AVE',
      number: 215
    }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83796"),
    certificate_number: 9303370,
    business_name: 'B & B SIDING CONTRACTORS LLC',
    address: { city: 'HAZLET', zip: 7730, street: 'LEITRIM LN', number: 8 }
  }
]
```

Figure 4.13: Documents in the out_inspection1 Collection

Figure 4.13 shows that the out_inspection1 collection now has four documents instead of the original three. This is because the document with address.city as HAZLET did not find a match and so that document was inserted into the collection.

4.2 Aggregation Pipeline Operators

Aggregation pipeline operators are used in various stages of the aggregation pipeline to perform arithmetic or logical calculations.

Different types of aggregation pipeline operators are:



4.2.1 Arithmetic Operators

Arithmetic operators perform arithmetic operations on the given operands and return a single value. Table 4.6 describes some of the arithmetic operators offered by MongoDB.

Name	Description
\$add	Used to add two or more numbers or numbers and one date and return the sum. It can only take one input as date. If date is one of the inputs, then the other inputs are treated as milliseconds for addition purposes and the result returned will be a date.
\$subtract	Used to subtract two specified numbers or dates and return the difference. It accepts two expressions and subtracts the second expression from the first one. If the two values are numbers, it returns the difference in value. If the two values are dates, it returns the difference in milliseconds. If one value is a date and the other is a number, it considers the number as milliseconds and returns the resulting date. In this case, the date should be specified first because it is not meaningful to subtract a date from a number.
\$multiply	Used to multiply two or more numbers and return the product. It accepts any number of argument expressions.

Name	Description
\$divide	Used to divide two numbers. It accepts two expressions as input and divides the value of the first expression by the value of the second expression.
\$ceil	Used to get the smallest integer greater than or equal to the given expression, if the expression resolves to a number. If the expression resolves to a null, this operator returns a null. If the expression evaluates to a value that is Not a Number, it returns NaN.
\$floor	Used to get the largest integer less than or equal to the given expression, if the expression resolves to a number. If the expression resolves to a null, this operator returns a null. If the expression evaluates to a value that is not a number, it returns NaN.
\$abs	Used to evaluate the absolute value of a given expression, if the expression resolves to a number. If the expression resolves to a null, this operator returns a null. If the expression evaluates to a value that is not a number, it returns NaN.
\$pow	Used to find the value of the specified number when raised to the power of the specified exponent.
\$exp	Used to evaluate the value of Euler's number (e) to the specified exponent, if the exponent resolves to a number. If the exponent resolves to a null or a value that is not a number, this operator returns null.
\$ln	Used to evaluate the natural log of a number.
\$log	Used to evaluate the log of a number in the specified base.
\$log10	Used to evaluate the log of a number to the base of 10.
\$mod	Used to retrieve the remainder of division of two numbers. It accepts two expressions and divides the first number by the second to return the result.
\$round	Used to round off a number to a whole integer or to a specified number of decimal places.
\$sqrt	Used to calculate the square root of the specified number.
\$trunc	Used to truncate a number to a whole integer or to a specified number of decimal places.

Table 4.6: Arithmetic Operators

Let us look at an example of one of the arithmetic operators, \$add. The syntax of the \$add operator is:

```
$add: {<expression1>, <expression2>, ...}
```

In this syntax, <expression> can be all valid numbers or a combination of numbers and one date.

Consider that in the sample_analytics database, the user wants to increase the sales limit in the accounts collection for all entries by 2000. The user wants to store this new limit in a field named newLimit. To do so, the user can run a query that uses the \$add operator as:

```
db.accounts.aggregate([ { $project: { account_id: 1, limit: 1, newLimit: { $add: ["$limit", 2000] } } }])
```

Figure 4.14 shows the output of this query.

```
sample_analytics> db.accounts.aggregate([ { $project: { account_id: 1, limit: 1, newLimit: { $add: ["$limit", 2000] } } } ])
[ {
  _id: ObjectId("5ca4bbc7a2dd94ee5816238c"),
  account_id: 371138,
  limit: 9000,
  newLimit: 11000
},
{
  _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
  account_id: 674364,
  limit: 10000,
  newLimit: 12000
},
{
  _id: ObjectId("5ca4bbc7a2dd94ee58162392"),
  account_id: 794875,
  limit: 9000,
  newLimit: 11000
},
{
  _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),
  account_id: 198100,
  limit: 10000,
  newLimit: 12000
},
{
  _id: ObjectId("5ca4bbc7a2dd94ee58162394"),
  account_id: 487188,
  limit: 10000,
  newLimit: 12000
},
```

Figure 4.14: Output of Query With \$add Operator

4.2.2 Array Operators

An array is a collection of values that has a single variable name. Each element in the array is identified by an index, which starts from 0. That is, the first element in the array will have the index as 0. Each value can be referred to by using the array name and its index. MongoDB provides various operators that work on arrays. Table 4.7 describes some of these operators.

Name	Description
\$first	Used to retrieve the first element in the specified array.
\$firstN	Used to retrieve the specified number of elements from an array. The elements are retrieved from the start of the array.
\$last	Used to retrieve the last element in the specified array.
\$lastN	Used to retrieve the specified number of elements from an array. The elements are retrieved from the end of the array.
\$isArray	Used to check if the specified expression is an array. If the expression is an array, it returns true; else, it returns false.
\$concatArray	Used to concatenate multiple arrays. This operator returns a single array.
\$filter	Used to select specific elements from an array. This operator returns an array with the elements that match the specified condition.
\$sortArray	Used to sort elements in an array in the ascending or descending order. For ascending order, it takes the sort direction as 1 and for descending order it takes the sort direction as -1.
\$maxN	Used to retrieve N number of elements with largest values from the array.
\$minN	Used to retrieve N number of elements with smallest values from the array.
\$arrayElemAt	Used to retrieve the element located at the specified index in the array.
\$indexofArray	Used to check if the specified value is present in the specified array. It returns the index of the first occurrence of the value in the array. If the value is not present in the array, it returns the value -1.
\$size	Used to retrieve the count of the elements in an array. It accepts a single expression as argument.

Name	Description
<code>\$slice</code>	Used to retrieve a subset of elements in an array from the start or end of the array or from a specified position in the array.
<code>\$arrayToObject</code>	Used to convert key-value pairs to documents.
<code>\$objectToArray</code>	Used to convert a document to an array key-value pairs that represent the documents.
<code>\$in</code>	Used to check if the specified value is present in the specified array. It returns true if the value is present in the array and false if it is not present in the array.
<code>\$map</code>	Used to apply an expression to the elements in an array. It returns the array with the new values that are arrived at by applying the expression.
<code>\$range</code>	Used to create an array with a sequence of numbers that are generated using the specified start number, end number, and step-size.
<code>\$reduce</code>	Used to apply an expression to each element in an array to arrive at new values. It then combines the values into a single value.
<code>\$reverseArray</code>	Used to reverse the order of the elements in an array.
<code>\$zip</code>	Used to get the transpose of two arrays. The resultant arrays will be formed by first elements of the first and second arrays, second elements of the first and second arrays and so on.

Table 4.7 Array Operators

Let us look at an example of the array operators, `$first` and `$last`. The syntax for these operators is:

`$first/$last:<expression>`

In this syntax, `<expression>` can be any valid expression that resolves to an array.

For example, the `grades` collection in the `sample_training` database has an array named `scores` that provides scores for `exam`, `quiz`, `homework`, and `homework` in that order. Consider that the user wants to retrieve the `exam` score, which is the first element in the `scores` array. To do so, the user can run a query with the `$first` operator as:

```
db.grades.aggregate([{$addFields: {exam_score: {$first: "$scores"} } },
{$project:{student_id:1,class_id:1, exam_score:1}}])
```

Figure 4.15 shows the output of this query.

```
sample_training> db.grades.aggregate([{$addFields: {exam_score: {$first: "$scores"} } },
{$project:{student_id:1,class_id:1, exam_score:1}}])
[ {
  _id: ObjectId("56d5f7eb604eb380b0d8d8d2"),
  student_id: 0,
  class_id: 391,
  exam_score: { type: 'exam', score: 41.25131199553351 }
},
{
  _id: ObjectId("56d5f7eb604eb380b0d8d8d0"),
  student_id: 0,
  class_id: 149,
  exam_score: { type: 'exam', score: 84.72636832669608 }
},
{
  _id: ObjectId("56d5f7eb604eb380b0d8d8d6"),
  student_id: 0,
  class_id: 108,
  exam_score: { type: 'exam', score: 25.926204502143857 }
},
{
  _id: ObjectId("56d5f7eb604eb380b0d8d8d7"),
  student_id: 0,
  class_id: 331,
  exam_score: { type: 'exam', score: 57.44037561654658 }
},
{
  _id: ObjectId("56d5f7eb604eb380b0d8d8d3"),
  student_id: 0,
  class_id: 7,
  exam_score: { type: 'exam', score: 11.182574562228819 }
},
```

Figure 4.15: Output of Query with \$first Operator

Now, consider that the user wants to retrieve the homework score, which is the last element in the `scores` array. To do so, the user can run a query with the `$last` operator as:

```
db.grades.aggregate([{$addFields:
{homework_score: {$last: "$scores"} } },
{$project:{student_id:1,class_id:1,
homework_score:1}}])
```

Figure 4.16 shows the output of this query.

```
sample_training> db.grades.aggregate([{$addFields: {homework_score: {$last: "$scores" }}}],{$project:{student_id:1,class_id:1, homework_score:1}}])  
[  
  {  
    _id: ObjectId("56d5f7eb604eb380b0d8d8d2"),  
    student_id: 0,  
    class_id: 391,  
    homework_score: { type: 'homework', score: 79.77471812670814 }  
  },  
  {  
    _id: ObjectId("56d5f7eb604eb380b0d8d8d0"),  
    student_id: 0,  
    class_id: 149,  
    homework_score: { type: 'homework', score: 80.85669686147487 }  
  },  
  {  
    _id: ObjectId("56d5f7eb604eb380b0d8d8d6"),  
    student_id: 0,  
    class_id: 108,  
    homework_score: { type: 'homework', score: 98.7923690220697 }  
  },  
  {  
    _id: ObjectId("56d5f7eb604eb380b0d8d8d7"),  
    student_id: 0,  
    class_id: 331,  
    homework_score: { type: 'homework', score: 63.127706923208194 }  
  },  
  {  
    _id: ObjectId("56d5f7eb604eb380b0d8d8d3"),  
    student_id: 0,  
    class_id: 7,  
    homework_score: { type: 'homework', score: 16.263573466709346 }  
},  
]
```

Figure 4.16: Output of Query with \$last Operator

4.2.3 Boolean Operators

Boolean operator takes the parameters in the form of expression and resolves them to Boolean values. It then returns a Boolean value as described in Table 4.8.

Name	Description
\$and	Returns true if all the specified expressions resolve to true; else, it returns false. It accepts multiple argument expressions.
\$not	Returns true if the specified expression resolves to false and it returns false if the specified expression resolves to true. It accepts a single argument expression.
\$or	Returns true if any one of the specified expressions evaluates to true. It accepts multiple argument expressions.

Table 4.8: Boolean Operators

Let us look at an example of a Boolean operator, \$and. The syntax for the \$and operators is:

```
{ $and: [ <expression1>, <expression2>, ... ] }
```

In this syntax, <expression> can be any valid expression that resolves to a Boolean value.

For example, the sample_training database has a trips collection, which specifies the tripduration and usertype for each document. Consider that the user wants to set the value of a new field named op_status to true for all the documents where tripduration is 379 and usertype is Subscriber. For all the other documents the value of op_status will be set to false. To do so, the user can run a query with the \$and operator as:

```
db.trips.aggregate([{$project:{tripduration: 1,
usertype: 1}},{$addFields:{op_status:{$and:
[{$eq:["$tripduration",379]},
{$eq:["$usertype","Subscriber"]}]}}},{$limit:4}])
```

Figure 4.17 shows the output of this query.

```
sample_training> db.trips.aggregate([{$project:{tripduration: 1, usertype: 1}},{$addFields:{op_status:{$and:
[{$eq:["$tripduration",379]}, {$eq:["$usertype","Subscriber"]}]}}}},{$limit:4}])
[
  {
    "_id": ObjectId("572bb8222b288919b68abf5d"),
    "tripduration": 923,
    "usertype": "Subscriber",
    "op_status": false
  },
  {
    "_id": ObjectId("572bb8222b288919b68abf5a"),
    "tripduration": 379,
    "usertype": "Subscriber",
    "op_status": true
  },
  {
    "_id": ObjectId("572bb8222b288919b68abf5b"),
    "tripduration": 889,
    "usertype": "Subscriber",
    "op_status": false
  },
  {
    "_id": ObjectId("572bb8222b288919b68abf5c"),
    "tripduration": 589,
    "usertype": "Subscriber",
    "op_status": false
  }
]
```

Figure 4.17: Output of Query with \$and Operator

4.2.4 Comparison Operators

As the name suggests, comparison operators compare two expressions and return a Boolean value. Table 4.9 lists the comparison operators.

Name	Description
\$cmp	Compares two expressions and returns: <ul style="list-style-type: none">• 0 if two expressions are equal• 1 if the first expression is greater than the second one• -1 if the first expression is less than the second one
\$eq	Compares two expressions and returns true if both the expressions are equal, else, it returns false
\$gt	Compares two expressions and returns true if the first expression is greater than the second one, else, it returns false
\$gte	Compares two expressions and returns true if the first expression is greater than or equal to the second one, else, it returns false
\$lt	Compares two expressions and returns true if the first expression is less than the second one, else, it returns false
\$lte	Compares two expressions and returns true if the first expression is less than or equal to the second one, else, it returns false
\$ne	Compares two expressions and returns true if both the expressions are not equal, else, it returns false

Table 4.9: Comparison Operators

Let us take a look at how to use the comparison operators, \$gt and \$lt. The syntax for these operators is:

```
{ $gt/$lt: [ <expression1>, <expression2> ] }
```

For example, the sample_analytics database has an accounts collection, which specifies the limit for each account. Consider that the user wants to retrieve the documents where the limit is greater than 9000 and less than 12000. To do so, the user can run a query with the \$and operator as:

```
db.accounts.aggregate([{$project:{account_id:1,limit:1,limit_status:{$and:[{$gt:["$limit",9000]},{$lt:["$limit",12000]}]}},products:1}])
```

Figure 4.18 shows the output of this query.

```
sample_analytics> db.accounts.aggregate([{$project:{account_id:1,limit:1, limit_status:{$and: [{$gt:["$limit",9000]}, {$lt:["$limit",12000]}]}},products:1}])  
[  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816238c"),  
    account_id: 371138,  
    limit: 9000,  
    products: [ 'Derivatives', 'InvestmentStock' ],  
    limit_status: false  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),  
    account_id: 674364,  
    limit: 10000,  
    products: [ 'InvestmentStock' ],  
    limit_status: true  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee58162392"),  
    account_id: 794875,  
    limit: 9000,  
    products: [ 'InvestmentFund', 'InvestmentStock' ],  
    limit_status: false  
  },  
  {  
    _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),  
    account_id: 198100,  
    limit: 10000,  
    products: [ 'Derivatives', 'CurrencyService', 'InvestmentStock' ],  
    limit_status: true  
  },  
]
```

Figure 4.18: Output of Query With \$gt and \$lt Operators

4.2.5 String Operator

String operators work on strings. Some of the string operators are listed in Table 4.10.

Name	Description
\$concat	Used to combine multiple strings into a single string
\$dateFromString	Used to convert a string into a date object The string that is specified should be a date/time string.
\$dateToString	Used to convert a date object to a date/time string in the specified format
\$indexofBytes	Used to search for an occurrence of a substring in a string When it finds the first matching occurrence, it returns the UTF-8 byte index of that occurrence. If it does not find the substring, it returns -1.
\$indexofCP	Used to search for an occurrence of a substring in a string

Name	Description
	When it finds the first matching occurrence, it returns the UTF-8 code point index of that occurrence. If it does not find the substring, it returns -1.
\$ltrim	Used to remove whitespace or the specified characters from the beginning of a string
\$regexFind	Used to apply a regular expression (regex) to a string It returns information about the <i>first</i> matched substring.
\$regexfindAll	Used to apply a regular expression (regex) to a string It returns information about all the matched substrings.
\$regexMatch	Used to apply a regular expression (regex) to a string It returns <code>true</code> if a matching substring is found; else, it returns <code>false</code> .
\$replaceOne	Used to replace the first occurrence of a matched string with the given input
\$replaceAll	Used to replace all the occurrences of a matched string with the given input
\$rtrim	Used to remove whitespace or the specified characters from the end of a string
\$split	Used to split a string into substrings based on a delimiter. It returns an array of substrings
\$strLenBytes	Used to find the length of a string. It returns the number of UTF-8 encoded bytes in a string
\$strLenCP	Used to find the length of a string. It returns the number of UTF-8 code points in a string
\$strcasecmp	Used to compare strings based on the casing of the letters in the string It returns: <ul style="list-style-type: none"> • 0 if the two strings are equivalent • 1 if the first string is greater than the second one • -1 if the first string is less than the second one
\$toLowerCase	Used to convert a string to lower case
\$toString	Used to convert a value to a string
\$trim	Used to remove whitespace or the specified characters from the beginning and end of a string
\$toUpperCase	Used to convert a string to upper case

Table 4.10: String Operators

Let us take a look at how to use the string operator, \$concat. The syntax for this operator is:

```
{ $concat: [ <expression1>, <expression2>, ... ] }
```

For example, the sample_training database has a trips collection, which specifies the start station name and end station name for each trip. Consider that the user wants to combine these two names with the delimiter as – and store this value in the source-destination field. To do so, the user can run a query with the \$concat operator as:

```
db.trips.aggregate([ { $project: { tripduration: 1, source_destination: { $concat: ["$start station name", " - ", "$end station name"] } } } ])
```

Figure 4.19 shows the output of this query.

```
sample_training> db.trips.aggregate([ { $project: { tripduration: 1, source_destination: { $concat: ["$start station name", " - ", "$end station name"] } } } ])[
{
  _id: ObjectId("572bb8222b288919b68abf60"),
  tripduration: 694,
  source_destination: 'Howard St & Centre St - E 17 St & Broadway'
},
{
  _id: ObjectId("572bb8222b288919b68abf61"),
  tripduration: 1376,
  source_destination: 'E 33 St & 2 Ave - South St & Whitehall St'
},
{
  _id: ObjectId("572bb8222b288919b68abf62"),
  tripduration: 1480,
  source_destination: 'Central Park S & 6 Ave - Central Park S & 6 Ave'
},
{
  _id: ObjectId("572bb8222b288919b68abf63"),
  tripduration: 615,
  source_destination: 'E 7 St & Avenue A - Norfolk St & Broome St'
},
{
  _id: ObjectId("572bb8222b288919b68abf64"),
  tripduration: 1770,
  source_destination: 'W 82 St & Central Park West - 9 Ave & W 22 St'
},
{
  _id: ObjectId("572bb8222b288919b68abf5a"),
  tripduration: 379,
  source_destination: 'E 31 St & 3 Ave - Broadway & W 32 St'
},
```

Figure 4.19: Output of Query with \$concat Operator

4.3 Summary

- Aggregation is processing of many documents in a collection and giving out a result.
- Aggregation pipeline is a series of steps, where some processing is done at each step and the result is passed to the next stage.
- Some of the stages of the aggregation pipeline include \$project, \$limit, \$match, \$skip, \$group, \$sort, \$addFields, \$out, and \$merge.
- Arithmetic operators, Boolean operators, string operators, and comparison operators are some of the operators in MongoDB that are used to manipulate data in the aggregation pipeline.

Test Your Knowledge

1. Which of the following statement is true about the aggregation pipeline in MongoDB?
 - a. It refers to a specific flow of operations that processes, transforms, and returns results.
 - b. It consists of one or more stages that process documents.
 - c. Each stage takes input from the previous stage.
 - d. Documents pass through the stages in random order.
2. Which of the following stage is usually the last stage of an aggregate pipeline?
 - a. \$group
 - b. \$sort
 - c. \$merge
 - d. \$skip
3. Consider that you have a collection `emp_detail` which consists of 20 documents. You must display only eight documents leaving the first two documents. Which of the following options will perform the given task using the `$limit` and `$skip` stages in any order?
 - a. `db.emp_detail.aggregate([{$limit: 10}, {$skip: 2}])`
 - b. `db.emp_detail.aggregate([{$skip: 2}, {$limit: 8}])`
 - c. `db.emp_detail.aggregate([{$limit: 8}, {$skip: 2}])`
 - d. `db.emp_detail.aggregate([{$skip: 2}, {$limit: 10}])`
4. Which of the following aggregation pipeline stages in MongoDB provides you to write the results of the aggregation pipeline to a collection?
 - a. \$project
 - b. \$group
 - c. \$out
 - d. \$merge
5. Which of the following string expression operator is used to split a string into substrings based on a delimiter and returns an array of substrings?
 - a. \$substr
 - b. \$split
 - c. \$trim
 - d. \$substrBytes

Answers to Test Your Knowledge

1	a, b, c
2	c
3	a, b
4	c, d
5	b

Try It Yourself

1. Create a database named Inventory and a collection named sales_invent.

2. Insert the following four documents into the sales_invent collection.

```
[  
  {  
    customername: "Richard",  
    gender:"M",  
    purchased_product:"cereals",  
    quantity:6,  
    price:60  
  },  
  { customername: "Williams",  
    gender:"M",  
    purchased_product:"Vegetables",  
    quantity:10,  
    price:150  
  },  
  { customername: "Emma",  
    gender:"F",  
    purchased_product:"Fruits",  
    quantity:8,  
    price:200  
  },  
  { customername: "John",  
    gender:"M",  
    purchased_product:"Baby Food",  
    quantity:3,  
    price:300  
  },  
  { customername: "Smith",  
    gender:"M",  
    purchased_product:"Fruits",  
    quantity:5,  
    price:180  
  }  
]
```

Using the sales_invent collection, perform the following tasks:

3. Exclude the `_id` field and display only the first ten documents of the `sales_invent` collection which include only the fields: `customername`, `purchased_product`, and `price`.
4. Use aggregation pipeline stages to group the documents by the `purchased_product` field and calculate the `Total sale amount` per product. Return only the products with `Total sale amount` is greater than or equal to 500.
5. Use aggregation pipeline stages to display only the first three documents of the `sales_invent` collection which includes only the `customername` and `purchased_product` fields where the `customername` is arranged in ascending order.
6. Add a field named `product_type:edibles` to all documents in the `sales_invent` collection.
7. In the first stage of aggregation pipeline, group by the `purchased_product` field and add the `quantity` fields into a new field named `Total_quantity`. In the second stage, write the output of first stage documents to a `Product_report` collection in the same Inventory database.
8. Use the arithmetic expression operator to calculate the `total price` calculated as `(price*quantity)` only for the last three documents.
9. Display only the product details where the `quantity` is greater than 5.
10. Use the string expression operator to concatenate the `customername` and the `purchased_product` field as `customername - purchased_product` in a new field named `customer_detail`. Display the details only for the `male` customer(s).



SESSION 5

DATABASE COMMANDS

Learning Objectives

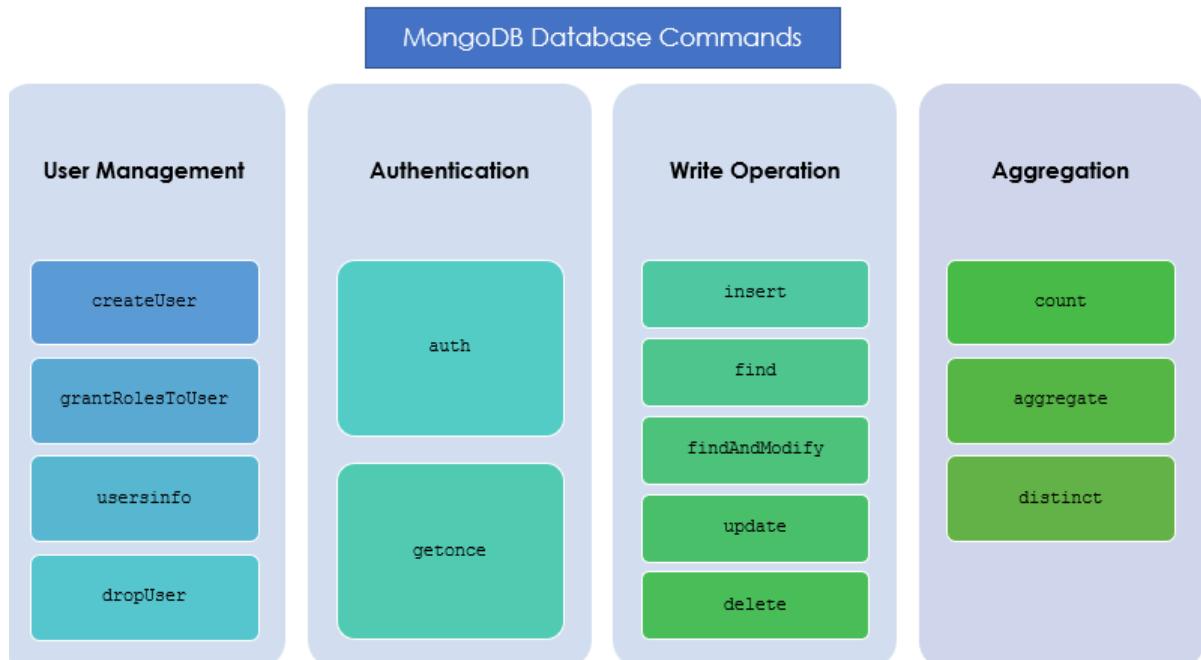
In this session, students will learn to:

- Explain the importance of database commands in MongoDB
- Describe the various types of database commands with examples

Database commands primarily involve operations related to creation and modification of user and collection data, authentication of users, and grouping of multiple documents. Queries involving database commands are often complex. This session explains various categories of database commands. This session also explores the parameters for each of the database commands and how to use them with examples.

5.1 Introduction to Database Commands

Developers use MongoDB database commands to create, modify, and update databases. Some of the categories in which MongoDB offers database commands include:



Before delving into these categories of commands, first let us understand how to run a command. The syntax to use the `runCommand` is:

```
db.runCommand( { <command> } )
```

Here, the user provides the command to be executed as a document or a string to the command.

Although `db.runCommand` runs the command against the current database, some commands are relevant only for the `admin` database. Therefore, the user must change the database object before running commands related to the `admin` database. Alternatively, the user can use `db.adminCommand` to run an administrative command against the `admin` database irrespective of the current database in use.

To run an administrative command against the `admin` database, the user can use the syntax:

```
db.adminCommand( { <command> } )
```



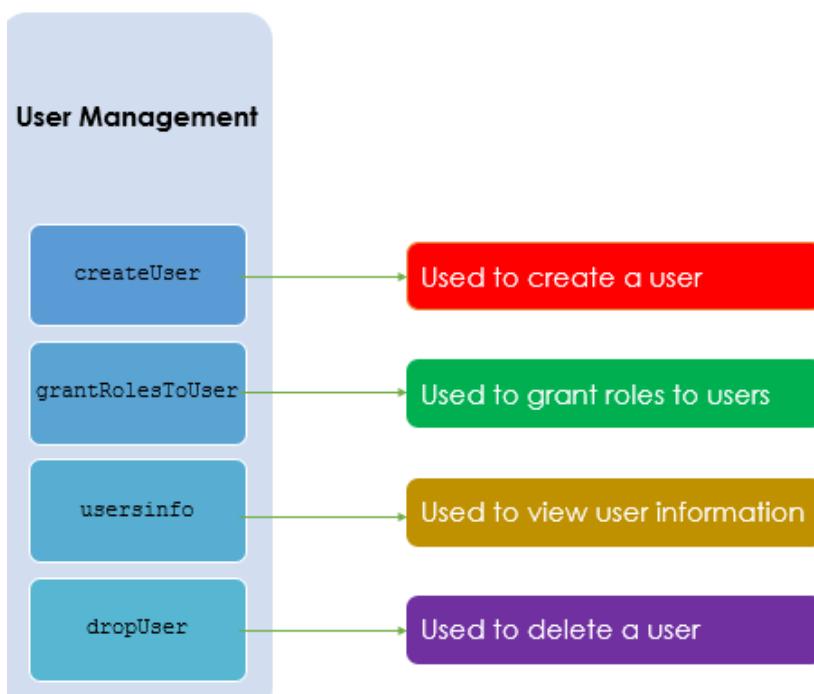
The `runCommand` command provides a helper to run specified database commands. This method provides a consistent interface between the shell and drivers and hence, is the most preferred method to issue database commands.

The `db.runCommand` and `db.adminCommand` commands take a database command as the parameter which can be a document or a string. If the database command is specified as a string, the string is automatically transformed into a document.

5.2 User Management Commands

User management commands in MongoDB allow operations related to users such as user creation/deletion, viewing user information, and role assignments.

MongoDB Database Commands



5.2.1 createUser Command

Users can use the `createUser` command to create a new user in the currently active database. If the user already exists, this command returns a **duplicate user** error message. The syntax for the `createUser` command is:

```
db.runCommand(  
{  
    createUser: "<name>",  
    pwd: passwordPrompt(), // Or "<cleartext password>"  
    customData: { <any information> },  
    roles: [  
        { role: "<role>", db: "<database>" } | "<role>",  
        ...  
    ],  
    writeConcern: { <write concern> },  
    authenticationRestrictions: [  
        { clientSource: [ "<IP|CIDR range>", ... ] },  
        serverAddress: [ "<IP|CIDR range>", ... ] },  
        ...  
    ],  
    mechanisms: [ "<scram-mechanism>", ... ], //Available  
    starting in MongoDB 4.0  
    digestPassword: <boolean>,  
    comment: <any>  
}
```

Table 5.1 discusses various parameters in the `createUser` command.

Parameter	Type	Description
<code>createUser</code>	string	Used to specify the name of the new user
<code>pwd</code>	string	Used to set the user's password either as a cleartext string or as <code>passwordPrompt</code> to prompt for the user's password
<code>customData</code>	document	Used to specify the extra information that an admin wishes to associate with a particular user
<code>roles</code>	array	Used to mention the role granted to the user
<code>digestPassword</code>	boolean	Used to specify whether the server or the client will digest the password

Parameter	Type	Description
writeConcern	document	Used to specify whether the level of write concern for the creation operation is at the transaction level or at the operation level
authentication Restrictions	array	Used to specify the restrictions posed by the server on the user; a list of IP addresses and Classless Inter-Domain Routing (CIDR) ranges from which the user is allowed to connect
mechanism	array	Used to specify the Salted Challenge Response Authentication Mechanism (SCRAM) mechanisms with valid SCRAM values as SCRAM-SHA-1 and SCRAM-SHA-256

Table 5.1: Parameters in the `createUser` Command

For example, to create a user named `User_1` on the `sample_training` database with the `readWrite` role, the user can use the query as:

```
use sample_training
db.runCommand({ createUser: "User_1", pwd: passwordPrompt(), roles: ["readWrite"] })
```

Figure 5.1 shows the creation of `User_1`. The system will prompt for a password. Type the password as `user1`.

```
sample_training> db.runCommand({ createUser: "User_1", pwd: passwordPrompt(), roles: ["readWrite"] })
Enter password
*****{ ok: 1 }
sample_training> |
```

Figure 5.1: Creation of `User_1`

Let us create another user `User_2` in the same database with the `readWrite` role.

```
db.runCommand({ createUser: "User_2", pwd: passwordPrompt(), roles: ["readWrite"] })
```

Figure 5.2 shows the creation of User_2. The system will prompt for a password. Type the password as user2.

```
sample_training> db.runCommand({ createUser: "User_2", pwd: passwordPrompt(), roles: ["readWrite"] })
Enter password
*****{ ok: 1 }
sample_training> |
```

Figure 5.2: Creation of User_2

5.2.2 grantRolesToUser Command

The `grantRolesToUser` command grants additional roles to a user. The syntax for the `grantRolesToUser` command is:

```
db.runCommand(
{
    grantRolesToUser: "<user>",
    roles: [ <roles> ],
    writeConcern: { <write
        concern> },
    comment: <any>
}
)
```

Assigning a role for a user on the currently active database involves:

- Specifying the role with the name of the role, for example, `readWrite`, or
- Specifying the role with a document, for example, `{ role: "<role>", db: "<database>" }`

However, to specify a role for a user that exists in a different database, the role with a document must be specified.

Consider the `sample_supplies` database. To grant the `read` role on the `sample-supplies` database to `User_1` of the `sample_training` database, the user must execute the query as:

```
db.runCommand({ grantRolesToUser: "User_1", roles: [
    { role: "read", db: "sample_supplies" },
    "readWrite" ] })
```

Figure 5.3 shows the result of the query execution.

```
sample_training> db.runCommand({ grantRolesToUser: "User_1", roles: [ { role: "read", db: "sample_supplies" }, "readWrite" ] })
{ ok: 1 }
sample_training> |
```

Figure 5.3: Specifying Additional `read` Role for `User_1`

An additional `read` role in a different database has been assigned to `User_1`.

5.2.3 `userInfo` Command

The `userInfo` command returns information about one or more users from the database. The syntax for the `userInfo` command is:

```
db.runCommand(
{
    userInfo: <various>,
    showCredentials: <Boolean>,
    showCustomData: <Boolean>,
    showPrivileges: <Boolean>,
    showAuthenticationRestrictions: <Boolean>,
    filter: <document>,
    comment: <any>
}
)
```

Table 5.2 lists the various forms of the `usersInfo` command.

Form	Description
<code>{ userInfo: 1 }</code>	Used to return information about the users in the database where the command is run
<code>{ userInfo: <username> }</code>	Used to return information about a specific user in the database where the command is run
<code>{ userInfo: { user: <name>, db: <db> } }</code>	Used to return information about the specific user in the mentioned database
<code>{ userInfo: [{ user: <name>, db: <db> }, ...] }</code> <code>{ userInfo: [<username1>, ...] }</code>	Used to return information about the specific users listed
<code>{ forAllDBs: true }</code>	Used to return information about users in all databases

Table 5.2: Forms of the `userInfo` Command

Consider that the user wants to view the information of `User_1` in the `sample_training` database. To do this, the user can execute the query as:

```
db.runCommand( { userInfo: { user: "User_1", db: "sample_training" }, showPrivileges: true })
```

Figure 5.4 shows the output of the query.

```
sample_training> db.runCommand( { userInfo: { user: "User_1", db: "sample_training" }, s
showPrivileges: true })
{
  users: [
    {
      _id: 'sample_training.User_1',
      userId: new UUID("4bc0950d-8141-41ac-ada0-8db69ea3d42e"),
      user: 'User_1',
      db: 'sample_training',
      mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ],
      roles: [
        { role: 'readWrite', db: 'sample_training' },
        { role: 'read', db: 'sample_supplies' }
      ],
      inheritedRoles: [
        { role: 'read', db: 'sample_supplies' },
        { role: 'readWrite', db: 'sample_training' }
      ],
      inheritedPrivileges: [
        {
          resource: { db: 'sample_supplies', collection: '' },
          actions: [
            'changeStream',
            'collStats',
            'dbHash',
            'dbStats',
            'find',
            'killCursors',
            'listCollections',
            'listIndexes'
          ]
        }
      ]
    }
  ]
}
```

Figure 5.4: Information on User_1 from the sample_training Database

5.2.4 dropUser Command

The dropUser command removes the user from the currently active database. The syntax for the dropUser command is:

```
db.runCommand(
{
  dropUser: "<user>",
  writeConcern: { <write concern> },
  comment: <any>
}
)
```

To drop the user named `User_2` from the `sample_training` database, the user can execute the query as:

```
db.runCommand({ dropUser: "User_2" })
```

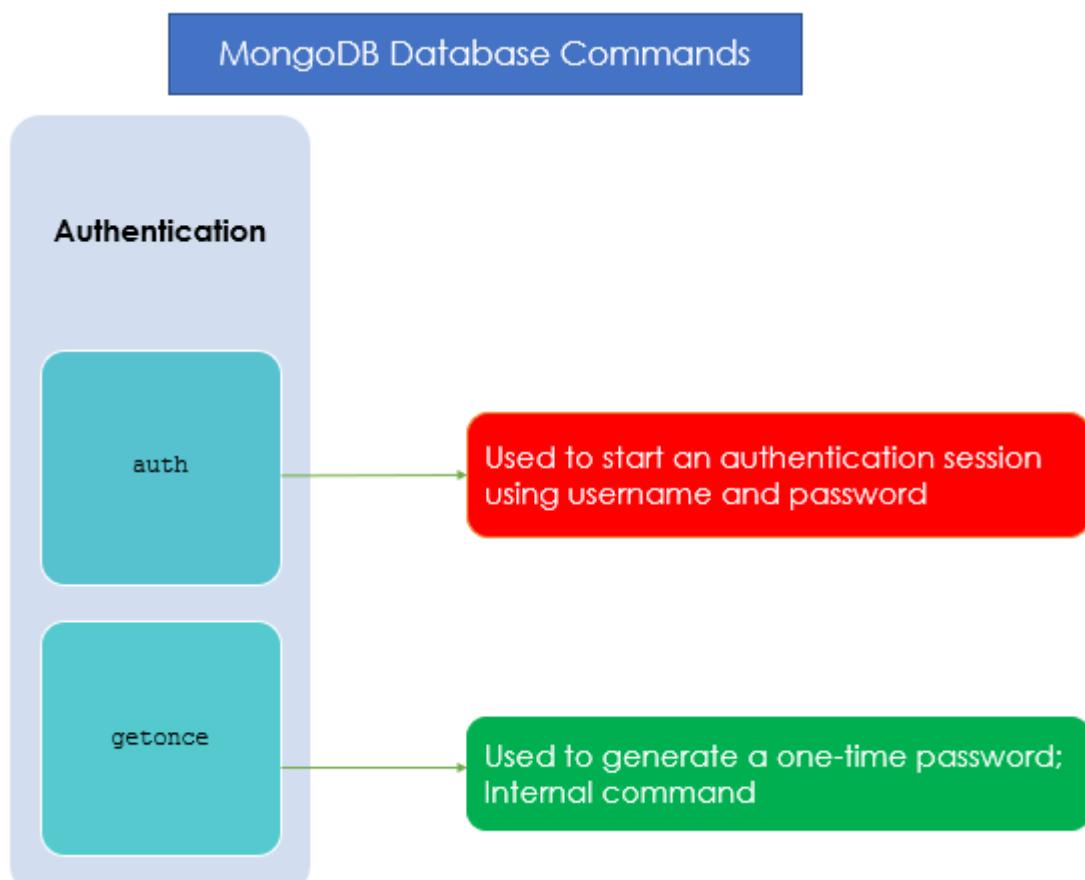
Figure 5.5 shows the output of the query.

```
sample_training> db.runCommand({ dropUser: "User_2" })
{ ok: 1 }
sample_training> |
```

Figure 5.5: Dropping User_2 from the sample_training Database

5.3 Authentication Commands

Authentication is the process in which the user credentials are validated against a stored set of values. MongoDB offers security through its authentication commands, which are:



This session discusses only the `auth` command in detail as the `getOnce` command is an internal command used by MongoDB.

5.3.1 Authenticate Command

The `db.auth` command authenticates a user using the x.509 authentication mechanism. This command returns 0 when authentication is not successful and 1 when the authentication is successful. The syntax for the `db.auth` command is:

```
db.auth( {  
    user: <username>,  
    pwd: passwordPrompt(), // or "<cleartext  
    password>"  
    mechanism: <authentication mechanism>,  
    digestPassword: <boolean>  
} )
```

Table 5.3 lists the parameters of the `db.auth` command.

Parameter	Type	Optional/ Mandatory	Description
user	string	Mandatory	Used to provide the name of the user with access privileges for this database
pwd	string	Mandatory	Used to specify the user's password either as a cleartext string or as <code>passwordPrompt()</code> to prompt for the user's password
mechanism	string	Optional	Used to specify an authentication mechanism
digestPassword	boolean	Optional	Used to determine whether to pre-hash the given password before using it with the specified authentication mechanism

Table 5.3: Parameters of the `db.auth` Command

There are two ways to prompt the user to enter a password. The user can omit the `password` field or use the `passwordPrompt` method to authenticate a username. The syntax for using the `passwordPrompt` method is:

```
db.auth( <username> )  
  
or  
  
db.auth( <username>, passwordPrompt() )
```

The queries will prompt the user to enter a password. To specify a cleartext password, the user can use the syntax as:

```
db.auth( <username>, <password> )
```



Authentication of user credentials is possible in MongoDB only after a connection to the shell is established.

To authenticate the user `User_1` with the password `user1` in the `sample_training` database, the user can execute the query as:

```
db.auth( "User_1", "user1" )
```

Figure 5.6 shows the output of the query as 1 indicating a successful authentication.

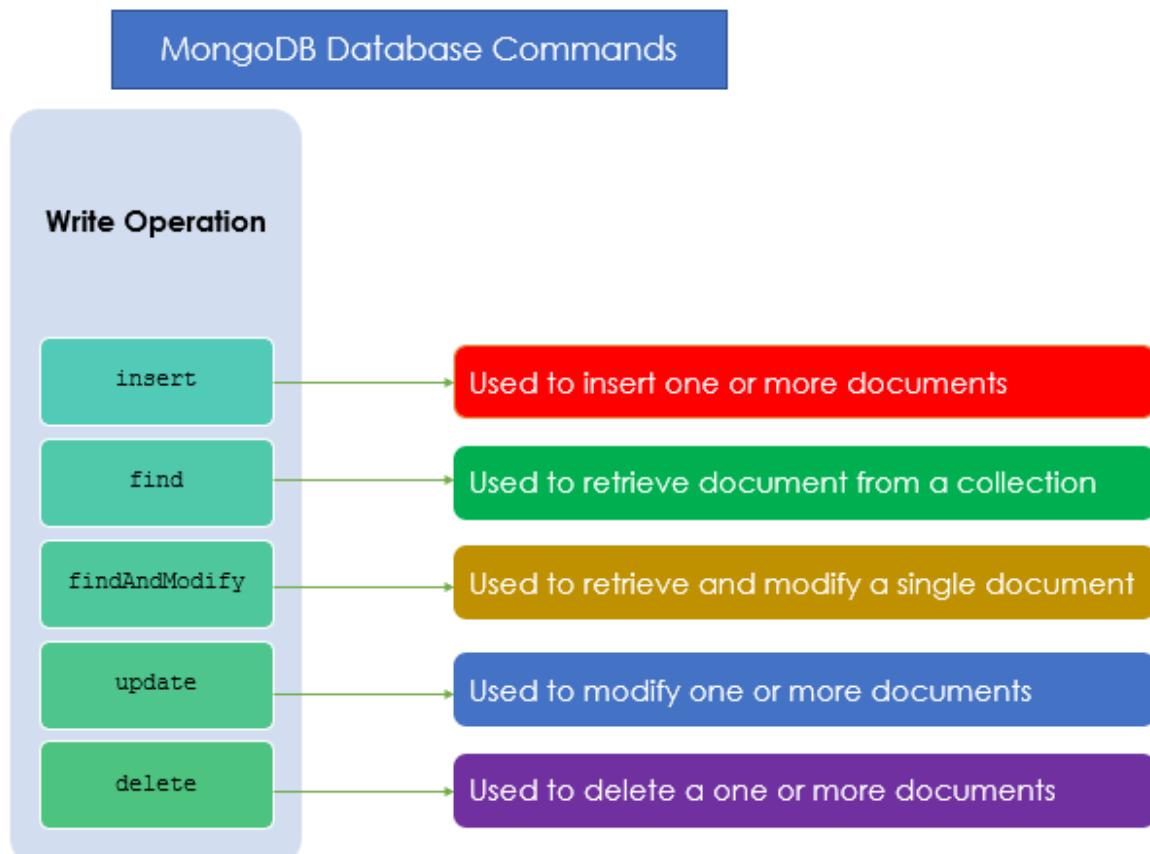
```
sample_training> db.auth( "User_1", "user1" )  
{ ok: 1 }  
sample_training> |
```

Figure 5.6: Authenticating User_1 with the Password user1

5.4 Write Operation Commands

A write operation is an action used to create or modify data in a MongoDB instance. In MongoDB, write operations target a single collection and affects one or more documents in that collection.

The commands offered by MongoDB to enable write operations are:



5.4.1 insert Command

The `insert` command inserts one or more documents into a collection. This command returns a document containing the status of all insert operations.

The syntax for the `insert` command is:

```
db.runCommand(  
{  
    insert: <collection>,  
    documents: [ <document>, <document>,  
    <document>, ... ],  
    ordered: <boolean>,  
    writeConcern: { <write concern> },  
    bypassDocumentValidation: <boolean>,  
    comment: <any>  
}  
)
```

Table 5.4 lists the parameters of the `insert` command.

Parameter	Type	Description
<code>insert</code>	string	Used to specify name of the target collection
<code>documents</code>	array	Used to provide the array of documents to be inserted into the named collection
<code>ordered</code>	boolean	Used to specify if the remaining commands must be performed in case this <code>insert</code> command fails; if set to <code>true</code> , all remaining commands will not be performed; if set to <code>false</code> , all remaining commands will be executed Optional; Default value is <code>true</code> .
<code>writeConcern</code>	document	Used to specify whether the level of write concern for the insert operation is at the transaction level or at the operation level Optional
<code>bypassDocumentValidation</code>	boolean	Used to insert documents that do not meet the validation criteria Optional
<code>comment</code>	any	Used to provide an arbitrary string that helps in tracing the operation through the database profiler, <code>currentOp</code> , and logs Optional

Table 5.4: Parameters of the `insert` Command

Consider the `accounts` collection in the `sample_analytics` database. If the user wants to insert a single document with `account_id` as 988877, `limit` as 2000, and `products` as `InvestmentFund`, then the user can execute the query as:

```
db.runCommand( { insert: "accounts", documents: [ { account_id: 988877, limit: 2000, products: ['InvestmentFund'] } ] } )
```

Figure 5.7 shows the output of the query.

```
sample_analytics> db.runCommand( { insert: "accounts", documents: [ { account_id: 988877, limit: 2000, products: [ 'InvestmentFund' ] } ] } ) { n: 1, ok: 1 }
```

Figure 5.7: Inserting a Single Document

To view the inserted document, the user can execute the query as:

```
db.accounts.find({ "account_id": 988877 })
```

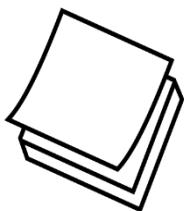
Figure 5.8 shows the output of the query.

```
sample_analytics> db.accounts.find({ "account_id": 988877 })
[
  {
    _id: ObjectId("647899949f9145cc454f992f"),
    account_id: 988877,
    limit: 2000,
    products: [ 'InvestmentFund' ]
  }
]
```

Figure 5.8: Insertion of the Document with account_id as 988877

5.4.2 find Command

The `find` command selects the documents that match the given query criteria from a collection and returns a cursor to the selected documents.



A cursor is a pointer to the resultant set of a query. Users can iterate through a cursor to perform operations on the results of the query. By default, cursors that are not opened within a session automatically time out after 10 minutes of inactivity. Cursors opened under a session will close when the session ends or times out.

The syntax for the `find` command is:

```
db.runCommand(  
{  
    find: <string>,  
    filter: <document>,  
    sort: <document>,  
    projection: <document>,  
    hint: <document or string>,  
    skip: <int>,  
    limit: <int>,  
    batchSize: <int>,  
    singleBatch: <bool>,  
    comment: <any>,  
    maxTimeMS: <int>,  
    readConcern: <document>,  
    max: <document>,  
    min: <document>,  
    returnKey: <bool>,  
    showRecordId: <bool>,  
    tailable: <bool>,  
    oplogReplay: <bool>,  
    noCursorTimeout: <bool>,  
    awaitData: <bool>,  
    allowPartialResults: <bool>,  
    collation: <document>,  
    allowDiskUse : <bool>,  
    let: <document> // Added in MongoDB 5.0  
}  
)
```

Table 5.5 lists some of the parameters in the `find` command.

Parameter	Type	Description
Find	string	Used to specify the name of the collection
filter	document	Used to provide the match criteria
Sort	document	Used to specify the order for sorting the result of the query

Parameter	Type	Description
projection	document	Used to specify the projection specification which lists the fields to be included in the result
Hint	string	Used to specify the index name as a string or as an index key pattern
Skip	positive integer	Used to list the number of documents to be skipped
Limit	Non-negative integer	Used to set the maximum number of documents to be returned
batchSize	Non-negative integer	Used to specify the number of documents to be returned in a batch of the result
singleBatch	boolean	Used to specify whether to close the cursor after returning the first batch of documents
maxTimeMS	positive integer	Used to provide the time limit for the processing operation on the cursor
readConcern	document	Used to set the read concern level as: readConcern: { level: <value> }
Max	document	Used to provide the upper bound for the given index
Min	boolean	Used to provide the lower bound for the given index
returnKey	boolean	Used to restrict the result to index keys, if set to true
showRecordID	boolean	Used to return the record identifier for each document, if set to true
noCursorTimeout	boolean	Used to prevent the server from timing out idle cursors
collation	document	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.5: Parameters of the `find` Command

Consider the `accounts` collection in the `sample_analytics` database. The user wants to find the documents where `limit` is less than 6000, display only the `account_id` and `limit` value, and sort the resulting documents by the `limit` value. To do this, the user can run the query as:

```
db.runCommand( { find: "accounts", filter: { limit: { $lt: 6000 } }, projection: { account_id: 1, limit:1 }, sort: { limit: 1 }})
```

Figure 5.9 shows the output of the query.

```
sample_analytics> db.runCommand( { find: "accounts", filter: { limit: 1 }, projection: { account_id: 1, limit:1 }, sort: { limit: 1 }})
{
  cursor: {
    firstBatch: [
      {
        _id: ObjectId("6478ca909f9145cc454fa38c"),
        account_id: 988877,
        limit: 2000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162661"),
        account_id: 417993,
        limit: 3000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),
        account_id: 113123,
        limit: 3000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee5816272e"),
        account_id: 170980,
        limit: 5000
      }
    ],
    id: Long("0"),
    ns: 'sample_analytics.accounts'
  },
  ok: 1
}
```

Figure 5.9: Result of the `find` Command

5.4.3 `findAndModify` Command

The `findAndModify` command filters, modifies, and returns a single document as result with or without the modifications.

The syntax for the `findAndModify` command is:

```
db.runCommand(  
{  
    findAndModify: <collection-name>,  
    query: <document>,  
    sort: <document>,  
    remove: <boolean>,  
    update: <document or aggregation  
    pipeline>,  
    new: <boolean>,  
    fields: <document>,  
    upsert: <boolean>,  
    bypassDocumentValidation: <boolean>,  
    writeConcern: <document>,  
    collation: <document>,  
    arrayFilters: <array>,  
    hint: <document|string>,  
    comment: <any>,  
    let: <document> // Added in MongoDB 5.0  
})
```

Table 5.6 lists some of the parameters in the `findAndModify` command.

Parameter	Type	Description
findAndModify	string	Used to specify the name of the collection in which a document must be searched for to modify or remove
query	document	Used to specify the selection criteria for the documents Optional; if omitted, defaults to an empty document
sort	document	Used to specify the document to be modified; the first document in the sorted result collection is modified
remove	boolean	Used to specify whether the document filtered by the query must be removed; either <code>remove</code> or <code>update</code> field must be set to true
update	Document or array	Used to specify the modifications to be made to the document selected by <code>query</code>
new	boolean	Used to specify whether the modified document must be returned

Parameter	Type	Description
fields	document	Used to list the fields to be returned
writeConcern	document	Used to specify whether the level of write concern for the operation is at the transaction level or at the operation level Optional
collation	document	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.6: Parameters of the `findAndModify` Command

By default, this command returns a document without the modifications. To return the document with the modifications, the user must include the `new` field set to `true`.

Consider the `accounts` collection in the `sample_analytics` database. The user wants to find the documents where `limit` is `2000`, modify the `limit` to `2500`, and display the modified document. To do this, the user can run the query as:

```
db.runCommand( { findAndModify: "accounts", query: { limit: 2000 }, update: { $set: { "limit": 2500 } }, new: true })
```

Figure 5.10 shows the modified document.

```
sample_analytics> db.runCommand( { findAndModify: "accounts", query: { limit: 2000 }, update: { $set: { "limit": 2500 } }, new: true })
{
  lastErrorObject: { n: 1, updatedExisting: true },
  value: {
    _id: ObjectId("6478ca909f9145cc454fa38c"),
    account_id: 988877,
    limit: 2500,
    products: [ 'InvestmentFund' ]
  },
  ok: 1
}
```

Figure 5.10: Result of the `findAndModify` Command



If no document matches the query condition, the `findAndModify` command returns a document with null in the `value` field.

5.4.4 update Command

The `update` command modifies one or more documents in a collection. A single `update` command can contain a single `update` statement or multiple `update` statements. The syntax for the `update` command is:

```
db.runCommand(  
{  
    update: <collection>,  
    updates: [  
        {  
            q: <query>,  
            u: <document or pipeline>,  
            c: <document>, // Added in MongoDB 5.0  
            upsert: <boolean>,  
            multi: <boolean>,  
            collation: <document>,  
            arrayFilters: <array>,  
            hint: <document|string>  
        },  
        ...  
    ],  
    ordered: <boolean>,  
    writeConcern: { <write concern> },  
    bypassDocumentValidation: <boolean>,  
    comment: <any>,  
    let: <document> // Added in MongoDB 5.0  
}  
)
```

Table 5.7 describes the parameters of the `update` command.

Parameter	Type	Description
<code>update</code>	string	Used to specify the name of the target collection
<code>updates</code>	array	Used to provide an array of one or more <code>update</code> statements which will perform modifications on the named collection
<code>ordered</code>	boolean	Used to specify if the remaining commands must be performed in case this <code>update</code> command fails

Parameter	Type	Description
		<ul style="list-style-type: none"> if set to <code>true</code>, the remaining commands will not be executed if set to <code>false</code>, the remaining commands will be executed Optional Default is <code>true</code>
<code>writeConcern</code>	<code>document</code>	Used to specify whether the level of write concern for the operation is at the transaction level or at the operation level Optional
<code>bypassDocumentValidation</code>	<code>boolean</code>	Used to update documents that do not meet the validation criteria Optional
<code>comment</code>	<code>any</code>	Used to provide an arbitrary string that helps in tracing the operation through the database profiler, <code>currentOp</code> , and logs Optional

Table 5.7: Parameters of the `update` Command

Table 5.8 lists the fields in the `updates` array of the `update` command.

Field	Type	Description
<code>q</code>	<code>document</code>	Used to specify the selection criteria for documents using the query selectors
<code>u</code>	<code>document</code> or <code>array</code>	Used to specify the modifications to be made in the documents that match the selection

Table 5.8: Fields of the `updates` Array

The user can either update a specific field in a single document or single field in multiple documents. Similarly, the user can also update multiple fields in a single document or multiple documents.

Consider the `accounts` collections in the `sample_analytics` database. This collection contains documents that have the `limit` value set to 3000. To view these documents, the user can execute the query as:

```
db.accounts.find({"limit": 3000})
```

Figure 5.11 shows two documents with limit as 3000.

```
sample_analytics> db.accounts.find({"limit": 3000})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162661"),
    account_id: 417993,
    limit: 3000,
    products: [ 'InvestmentStock', 'InvestmentFund' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),
    account_id: 113123,
    limit: 3000,
    products: [ 'CurrencyService', 'InvestmentStock' ]
  }
]
```

Figure 5.11: Viewing Documents with limit as 3000

To update all the documents with the limit value of 3000 to 4000, the user can execute the query as:

```
db.runCommand({update: "accounts", updates: [{q:
{limit: 3000},u: { $inc: { limit: 1000 } },multi:
true} ]})
```

Figure 5.12 shows the output of this query.

```
sample_analytics> db.runCommand({update: "accounts",updates: [{q: {limit: 3000},
u: { $inc: { limit: 1000 } },multi: true} ]})
{ n: 2, nModified: 2, ok: 1 }
sample_analytics> |
```

Figure 5.12: Updating the limit as 4000

To view the update, the user can execute the query as:

```
db.accounts.find({"limit": 4000})
```

Figure 5.13 shows the output of this query.

```
sample_analytics> db.accounts.find({"limit": 4000})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162661"),
    account_id: 417993,
    limit: 4000,
    products: [ 'InvestmentStock', 'InvestmentFund' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),
    account_id: 113123,
    limit: 4000,
    products: [ 'CurrencyService', 'InvestmentStock' ]
  }
]
```

Figure 5.13: Viewing the Updation

5.4.5 delete Command

The `delete` command deletes one or more documents from a collection. A single `delete` command can contain multiple `delete` specifications.

The syntax for the `delete` command is:

```
db.runCommand(
{
  delete: <collection>,
  deletes: [
    {
      q : <query>,
      limit : <integer>,
      collation: <document>,
      hint: <document|string>
    },
    ...
  ],
  comment: <any>,
  let: <document>, // Added in MongoDB 5.0
  ordered: <boolean>,
  writeConcern: { <write concern> }
})
```

Table 5.9 lists the parameters in the `delete` command.

Parameter	Type	Description
<code>delete</code>	string	Used to specify the name of the target collection from which the document must be deleted
<code>deletes</code>	array	Used to provide an array of delete statements within the <code>delete</code> command
<code>ordered</code>	boolean	Used to specify if the remaining commands must be executed in case the <code>delete</code> command fails <ul style="list-style-type: none"> • if set to <code>true</code>, the remaining commands will not be executed • if set to <code>false</code>, the remaining commands will be executed • Optional • Default is <code>true</code>
<code>writeConcern</code>	document	Used to specify whether the level of write concern for the operation is at the transaction level or at the operation level Optional

Table 5.9: Parameters of the `delete` Command

Table 5.10 lists the fields in the `deletes` array of the `delete` command.

Field	Type	Description
<code>q</code>	document	Used to specify the selection criteria for the documents using the query selectors
<code>limit</code>	Non-negative integer	Used to set the number of matching documents to be deleted; must be set to 0 to delete all matching documents and 1 to delete a single matching document
<code>collation</code>	document	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.10: Fields of the `deletes` Array

Consider the `accounts` collection in the `sample_analytics` database. This collection contains documents that has `limit` set to 7000. To view these documents, the user can run the query as:

```
db.runCommand( { find: "accounts", filter: { limit: { $eq: 7000 }}, projection: { account_id: 1, limit:1 } })
```

Figure 5.14 shows five documents that match this query.

```
sample_analytics> db.runCommand( { find: "accounts", filter: { limit: { $eq: 7000 }}, projection: { account_id: 1, limit:1 } })
{
  cursor: {
    firstBatch: [
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162458"),
        account_id: 852986,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee5816247a"),
        account_id: 777752,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162530"),
        account_id: 453851,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162565"),
        account_id: 354107,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee581625ad"),
        account_id: 385361,
        limit: 7000
      }
    ],
    id: Long("0"),
    ns: 'sample_analytics.accounts'
  },
}
```

Figure 5.14: Viewing all Documents with limit as 7000

To delete all documents with limit as 7000, the user can execute the query as:

```
db.runCommand( { delete: "accounts", deletes: [ { q: { limit: 7000 }, limit: 0 } ] } )
```

In the query, limit is set to 0 because all the documents must be deleted. Figure 5.15 shows the output of this query.

```
sample_analytics> db.runCommand( { delete: "accounts", deletes: [ { q: { limit: 7000 }, limit: 0 } ] } )
{ n: 5, ok: 1 }
sample_analytics> |
```

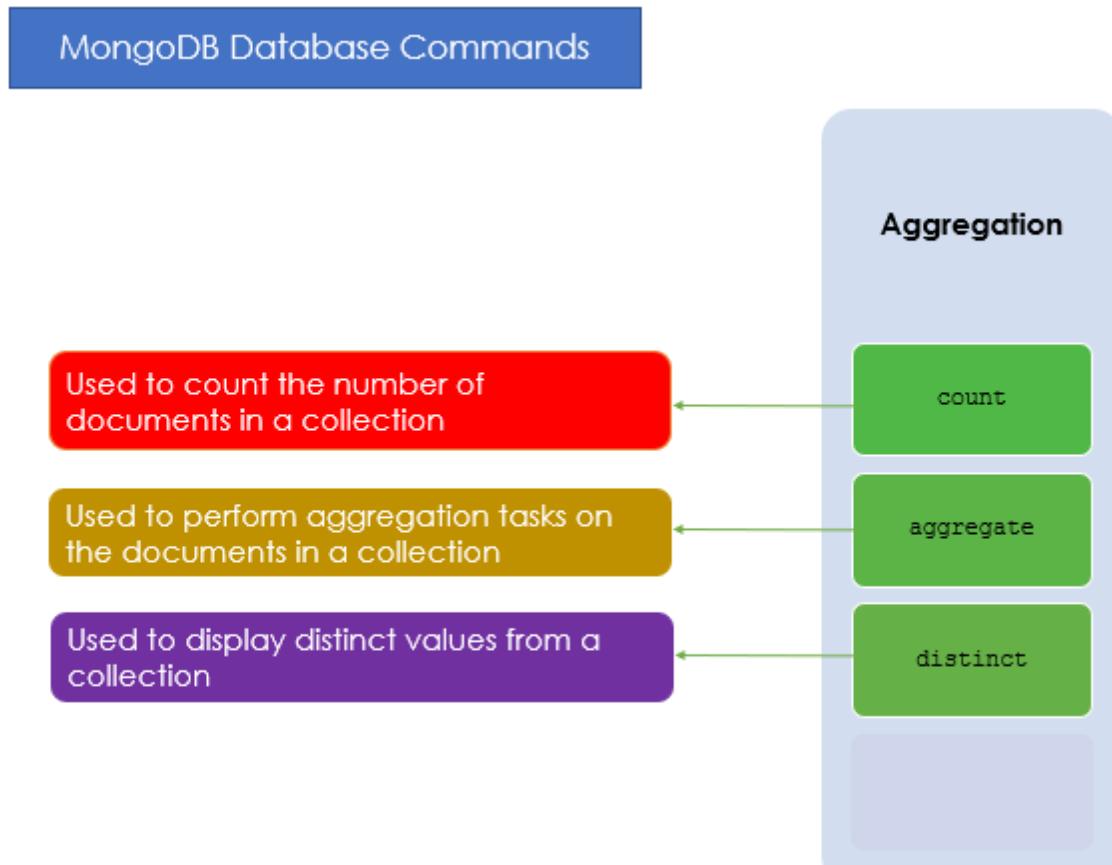
Figure 5.15: Deletion of documents with limit as 7000

5.5 Aggregation Commands

Aggregation commands allow the user to group values from multiple documents, perform operations on the grouped data, and return a single result.

To perform aggregation operations, the user can use aggregation pipelines or single aggregation methods.

Aggregation methods offered by MongoDB are:



5.5.1 count Command

As the name suggests, the `count` command counts the number of documents in a collection. It then returns a document that contains this count and the command status. The syntax of the `count` command is:

```

db.runCommand(
{
    count: <collection or view>,
    query: <document>,
    limit: <integer>,
    skip: <integer>,
    hint: <hint>,
    readConcern: <document>,
    collation: <document>,
    comment: <any>
}
)

```

Table 5.11 lists the parameters of the `count` command.

Parameter	Type	Description
count	string	Used to specify the name of the collection on which the count operation is to be performed
query	document	Used to specify the selection criteria to filter the documents to count in the collection
limit	integer	Used to specify the maximum number of matching documents to be returned
skip	integer	Used to specify the number of matching documents to be skipped before returning results
hint	string	Used to specify the index to be used
readConcern	document	Used to specify the read concern level of the operation
collation	document	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.11: Parameters of the `count` Command

Consider the `accounts` collection in the `sample_analytics` database. To count the number of documents that has `limit` less than or equal to 4000, the user can execute the query as:

```
db.runCommand({count: "accounts", query: {limit: {$lte: 4000}}})
```

Figure 5.16 shows the output of this query.

```
sample_analytics> db.runCommand( { count:'accounts',query: { limit: { $lte: 4000 } } } )
{ n: 3, ok: 1 }
sample_analytics> |
```

Figure 5.16: Counting Documents that Match a Query

5.5.2 aggregate Command

The aggregate command performs the aggregation operation using the aggregation pipeline.

The syntax for the aggregate command is:

```
db.runCommand(
{
  aggregate: "<collection>" || 1,
  pipeline: [ <stage>, <...> ],
  explain: <boolean>,
  allowDiskUse: <boolean>,
  cursor: <document>,
  maxTimeMS: <int>,
  bypassDocumentValidation: <boolean>,
  readConcern: <document>,
  collation: <document>,
  hint: <string or document>,
  comment: <any>,
  writeConcern: <document>,
  let: <document> // Added in MongoDB 5.0
}
)
```

Table 5.12 lists the parameters of the `aggregate` command.

Parameter	Type	Description
aggregate	string	Used to specify the name of the aggregation pipeline
pipeline	array	Used to give an array that transforms the list of documents as a part of the aggregation pipeline

Parameter	Type	Description
explain	boolean	Used to return information about the processing of the pipeline
allowDiskUse	boolean	Used to allow the command to write to the temporary files, if set to true
cursor	document	Used to specify the documents that can control the creation of the cursor object
maxTimeMS	non-negative integer	Used to set a time limit for the processing operations on a cursor
readConcern	document	Used to specify the read concern level: local, available, majority, or linearizable
collation	document	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks
comment	string	Used to provide an arbitrary string that helps in tracing the operation through the database profiler, currentOp, and logs Optional
writeConcern	document	Used to set the default write concern with the \$out or \$merge pipeline stages

Table 5.12: Parameters of the aggregate Command

Consider the accounts collection in the sample_analytics database. If the user wants a list of available limit and the number of documents in each limit, then the user can run the query as:

```
db.runCommand({aggregate: "accounts", pipeline: [
    {$project: { account_id: 1, limit:1 } },
    { $group : {
        _id : "$limit",
        count: { $count: {} }
    }}
]})
```

Figure 5.17 shows the output of this query.

```
sample_analytics> db.runCommand({aggregate: "accounts", pipeline: [{"$project": {"account_id": 1, "limit": 1}}, {"$group": {"_id": "$limit", "count": {"$count": {}}}}], cursor: {}})
{
  cursor: {
    firstBatch: [
      {"_id": 8000, "count": 6},
      {"_id": 10000, "count": 1701},
      {"_id": 4000, "count": 2},
      {"_id": 5000, "count": 1},
      {"_id": 2500, "count": 1},
      {"_id": 9000, "count": 31}
    ],
    id: Long("0"),
    ns: 'sample_analytics.accounts'
  },
  ok: 1
}
```

Figure 5.17: Result of the aggregate Command

5.5.3 distinct Command

The `distinct` command locates the distinct values for a specified field from a single collection. This command returns the resultant data as a document containing an array of distinct values. The syntax for this command is:

```
db.runCommand(
{
  distinct: "<collection>",
  key: "<field>",
  query: <query>,
  readConcern: <read concern document>,
  collation: <collation document>,
  comment: <any>
}
)
```

Table 5.13 lists the parameters of the `distinct` command.

Parameter	Type	Description
<code>distinct</code>	string	Used to specify the name of the collection to use for the query
<code>key</code>	string	Used to provide the field for which the command must return distinct values
<code>query</code>	document	Used to specify the documents from where the distinct values must be retrieved
<code>readConcern</code>	document	Used to specify the read concern level

Parameter	Type	Description
collation	document	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.13: Parameters of the distinct Command

Consider the accounts collection in the sample_analytics database. If the user wants a list of distinct products available, then the user can run the query as:

```
db.runCommand ( { distinct: "accounts", key: "products" } )
```

Figure 5.18 shows the output of the query.

```
sample_analytics> db.runCommand ( { distinct: "accounts", key: "products" } )
{
  values: [
    'Brokerage',
    'Commodity',
    'CurrencyService',
    'Derivatives',
    'InvestmentFund',
    'InvestmentStock'
  ],
  ok: 1
}
```

Figure 5.18: Distinct Values of the products Array Field

5.6 Summary

- Database commands allow users to create and edit databases.
- MongoDB offers database commands for user management, authentication, write operations, and aggregation.
- User management commands include user creation, user deletion, viewing of user information, and granting user roles.
- Authentication happens using the SCRAM mechanism in MongoDB.
- Write operation commands in MongoDB allows insertion of data to a collection, modifying existing data, and deleting data from a collection.
- Aggregation commands perform operations on grouped values from multiple documents to return a single result.

Test Your Knowledge

1. Which of the following option is used to run a command against the current database?
 - a. db.commandRun
 - b. db.runAdmin
 - c. db.runDatabase
 - d. db.runCommand

2. Which of the following options are the required fields of createUser command?
 - a. pwd
 - b. mechanisms
 - c. createUser
 - d. roles

3. Which of the following field of the deletes array field of delete command represents the query that matches the documents to delete?
 - a. query
 - b. q
 - c. qy
 - d. deleteQuery

4. Which of the following query will return the total number of documents in the inventory collection?
 - a. db.runCommand({count: "inventory"})
 - b. db.runCommand({aggregate: "inventory"})
 - c. db.runCommand({aggregate: {count:"inventory"} })
 - d. db.runCommand({sum: {count:"inventory"} })

5. Which of the following query returns the distinct values for the field designation from all the documents in the employee collection?
 - a. db.runCommand ({ distinct: "employee", : "designation" })
 - b. db.runCommand ({ unique: "employee", key: "designation" })
 - c. db.runCommand ({ distinct: "employee", key: "designation" })
 - d. db.runCommand ({ unique: "employee", field: "designation" })

Answers to Test Your Knowledge

1	d
2	a, c, d
3	b
4	a
5	c

Try it Yourself

1. Create a database named product and a collection named sales_product.
2. Insert the following four documents into the sales_product collection.

```
[  
    { product_id: 1243, product_type: "stationary", branch: "B-111", units_sold: 678, profit_millions: 5 },  
    { product_id: 1144, product_type: "grocery", branch: "B-112", units_sold: 2500, profit_millions: 8 },  
    { product_id: 1345, product_type: "baby items", branch: "B-113", units_sold: 1500, profit_millions: 3 },  
    { product_id: 1567, product_type: "pet care", branch: "B-114", units_sold: 1725, profit_millions: 4 }  
]
```

Using the sales_product collection, perform the following tasks:

3. Using database commands create three users named db_user1, db_user2 and db_user3 for the database product using the password db1, db2, and db3, respectively.
4. Grant readwrite role to user db_user1 and db_user2 on the product database and grant read role for both these users on the inventory database, which was created in the previous session.
5. Grant read role to user db_user3 on the product database and grant readwrite role for this user on the inventory database.
6. Use a database command to view the information of db_user1 and db_user3.
7. Use a database command to remove the user db_user2 from the product database.
8. Use database command to authenticate the user db_user1 on the product database.
9. Insert the document into the database product as:

```
{ product_id: 1899, product_type: "diary", branch: "B-115", units_sold: 3005, profit_millions: 6 }
```

10. Update the field branch as B-117 for the documents with product_type as "diary".
11. Using the findAndModify database command, find a product with product_type as "baby items", modify the product_type as "baby

foods”, and return the document with the modifications made after update.

12. Use aggregate database commands to retrieve the details of the product that has `profit_millions` greater than 5.



SESSION 6

MONGODB SHELL METHODS

Learning Objectives

In this session, students will learn to:

- List different MongoDB Shell methods
- Describe collection methods in MongoDB Shell
- Explain various database methods in MongoDB Shell
- Explain role management methods in MongoDB Shell
- Describe various user management methods in MongoDB Shell

As discussed in an earlier session, MongoDB Shell is the command-line interface that provides quick and easy access to the MongoDB database. MongoDB Shell provides various collection methods to create, delete, rename, and manage collections in a database. It also provides database methods to create, run, drop, and fetch data from collections. There are also role management methods and user management methods to create, update, and drop roles and users in a database. This session will provide an overview of different MongoDB Shell methods to manage collections, users, and roles in a MongoDB database.

6.1 Introduction to MongoDB Shell Methods

MongoDB provides a collection of methods that facilitate the users to:

- Create, insert, update, and delete documents, collections, and databases
- Change the way a query is executed
- Manage users and authentication
- Provide built-in roles and authorization based on roles
- Perform bulk operations
- Replicate data in multiple MongoDB servers
- Distribute data of very large datasets across multiple machines

Some of the important methods provided by MongoDB are:



6.2 Collection Methods

MongoDB offers a range of collection methods to perform the Create, Read, Update, and Delete (CRUD) operations which has been covered in earlier sessions.

- **Count**

MongoDB provides the `countDocuments` method to count the number of documents in a collection. The syntax for the `countDocuments` method is:

```
db.collection.countDocuments(query, options)
```

Table 6.1 describes the parameters of the countDocuments method.

Parameter	Fields	Type	Description
query	-	document	Specifies the criteria based on which the documents must be counted; if empty, all the documents are counted
options		document	Provides optional fields
	limit	integer	Specifies the maximum number of documents that must be counted
	skip	integer	Specifies the number of documents to skip before beginning the count
	hint	string or document	Specifies an index name to be used in the query
	maxTimeMS	integer	Specifies the maximum time the query can run

Table 6.1: Parameters of the countDocuments Method

In the sample_analytics database, consider that the user wants to know the total number of documents in the accounts collection. To count the number of documents, the user can run the query as:

```
db.accounts.countDocuments({})
```

The output of this query is shown in Figure 6.1.

```
sample_analytics> db.accounts.countDocuments()
1746
```

Figure 6.1: Output of the countDocuments Method

Consider that the user wants the total number of documents in the accounts collection that has the value of the limit field set greater than 10000. To get the required count, the user can execute the query as:

```
db.accounts.countDocuments( { limit: { $gte: 10000} })
```

The output of the query is shown in Figure 6.2.

```
sample_analytics> db.accounts.countDocuments( { limit: { $gte: 10000}})  
1701
```

Figure 6.2: Output of the countDocuments Method with Condition

- **Distinct**

MongoDB provides the `distinct` method to fetch distinct values for a field in a collection. The syntax for the `distinct` method is:

```
db.collection.distinct(field, query, options)
```

Table 6.2 describes the parameters of the `distinct` method.

Parameter	Fields	Type	Description
field	-	string	Specifies the field for which distinct values must be fetched
query	-	document	Specifies the criteria based on which distinct values must be fetched
options	collation	document	Specifies language-specific rules such as lowercase

Table 6.2: Parameters for the distinct method

Consider that the user wants to fetch the distinct values for the `storeLocation` field in the `products` document of the `accounts` collection. To get the desired output, the user can execute the query as:

```
db.accounts.distinct( "products" )
```

The output of the query is shown in Figure 6.3.

```
sample_analytics> db.accounts.distinct( "products" )
[
  'Brokerage',
  'Commodity',
  'CurrencyService',
  'Derivatives',
  'InvestmentFund',
  'InvestmentStock'
]
```

Figure 6.3: Output of the distinct Method

- **Find and Update**

MongoDB provides the `findOneAndUpdate` method to locate a document and update its fields based on the criteria specified in the method. The syntax for the `findOneAndUpdate` method is:

```
db.collection.findOneAndUpdate(filter, update, options)
```

Table 6.3 describes the parameters of the `findOneAndUpdate` method.

Parameter	Fields	Type	Description
filter	-	document	Specifies the criteria based on which the document must be updated; if empty, then the first document in the collection is updated
update	-	document	Specifies the updates that must be done
options		document	Specifies optional fields
	projection	document	Specifies the fields that must be returned; if omitted, then all the fields are returned
	sort	document	Specifies the order in which the documents matched by the filter parameter must be sorted
	maxTimeMS	number	Specifies the maximum

Parameter	Fields	Type	Description
			time in milliseconds within which the query must run; if exceeded, throws an error
	upsert	boolean	Converts an update to insert a new document if set to true and no matching document exists for the criteria specified in filter
	returnDocument	string	Specifies whether the original or the updated document will be returned <ul style="list-style-type: none"> If set to before, the original document is returned If set to after, the updated document is returned Takes precedence over returnNewDocument parameter
	returnNewDocument	boolean	Fetches the updated document if set to true
	collation	document	Specifies language-specific rules such as lettercase
	arrayFilters	array	Specifies the array criteria based on which elements in an array must be modified

Table 6.3: Parameters for the `findOneAndUpdate` Method

Consider that the user wants to find the document in the accounts collection, where `account_id` is 674364 and increase the value of its limit by 500. To view the existing limit in this document before performing the update operation, the user can execute the query as:

```
db.accounts.find({ "account_id" : 674364 })
```

The output of the query is shown in Figure 6.4.

```
sample_analytics> db.accounts.find({"account_id":674364})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
    account_id: 674364,
    limit: 10000,
    products: [ 'InvestmentStock' ]
  }
]
```

Figure 6.4: Output of the `find` Method with Specific `account_id`

The value in the `limit` field is 10000. To increase the value in the `limit` field by 500 for this document, execute the query as:

```
db.accounts.findOneAndUpdate({ "account_id" : 674364 },
{ $inc : { "limit" : 500 } }, {returnNewDocument:true})
```

The output of the query is shown in Figure 6.5.

```
sample_analytics> db.accounts.findOneAndUpdate({ "account_id" : 674364 },
{ $inc : { "limit" : 500 } }, {returnNewDocument:true})
{
  _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
  account_id: 674364,
  limit: 10500,
  products: [ 'InvestmentStock' ]
}
```

Figure 6.5: Output of the `findOneAndUpdate` Method

The value in the `limit` field is updated as 10500 in the returned document. Here, `$inc` is the update operator and the `returnNewDocument` option when set to `true` ensures that the updated document is returned.

- **Find and Replace**

MongoDB provides the `findOneAndReplace` method to locate a document and replace it based on the conditions specified in the method. The syntax for the `findOneAndReplace` method is:

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

Table 6.4 describes the parameters of the `findOneAndReplace` method.

Parameter	Fields	Type	Description
filter	-	document	Specifies the criteria based on which the document must be replaced; if empty, then the first document in the collection is replaced
replacement		document	Specifies the replacement document; <code>_id</code> value in the replacement document must be same as that of the replaced document
options		document	Specifies optional fields
	projection	document	Specifies the fields that must be returned; if omitted, then all the fields are returned
	sort	document	Specifies the order in which the documents matched by the filter parameter must be sorted
	maxTimeMS	number	Specifies the maximum time in milliseconds within which the query must run; if exceeded, throws an error
	upsert	boolean	Inserts a document specified by the replacement parameter if set to <code>true</code> and there is no document matching the criteria in the filter parameter
	returnDocument	string	Specifies whether the original or the updated document will be returned <ul style="list-style-type: none"> If set to <code>before</code>, the original document is returned

Parameter	Fields	Type	Description
			<ul style="list-style-type: none"> If set to after, the updated document is returned Takes precedence over returnNewDocument parameter
	returnNewDocument	boolean	Fetches the replaced document if set to true
	collation	document	Specifies language-specific rules such as lowercase

Table 6.4: Parameters of the `findOneAndReplace` Method

Consider that the user wants to find all documents in the accounts collection, where the value of the limit field is greater than 5000. In the first document returned, the user wants to replace the value of the products field as Commodity and the limit field as 2500. To view all the documents where the value of the limit field is greater than 5000 before performing the replacement operation, the user can execute the query as:

```
db.accounts.find({"limit":{$gte:5000}})
```

The output of the query is shown in Figure 6.6.

```
sample_analytics> db.accounts.find({"limit":{$gte:5000}})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
    account_id: 674364,
    limit: 10500,
    products: [ 'InvestmentStock' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),
    account_id: 198100,
    limit: 11500,
    products: [ 'Derivatives', 'CurrencyService', 'InvestmentStock' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238d"),
    account_id: 557378,
    limit: 10000,
    products: [ 'InvestmentStock', 'Commodity', 'Brokerage', 'CurrencyService' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162390"),
    account_id: 278603,
    limit: 10000,
    products: [ 'Commodity', 'InvestmentStock' ]
  }
]
```

Figure 6.6: Output of the `find` Method with Specific Limit Value

In the first document that is returned, the value of the `products` field is `InvestmentStock`, and the `limit` field is `10000`. To replace the value of the `products` field as `Commodity` and the `limit` field as `2500`, the user can execute the query as:

```
db.accounts.findOneAndReplace({ "limit" : { $gt : 5000
} },{ "product" : "Commodity", "limit" : 2500 },
{returnNewDocument:true})
```

The output of the query is shown in Figure 6.7.

```
sample_analytics> db.accounts.findOneAndReplace({ "limit" : { $gt : 5000
} },{ "product" : "Commodity", "limit" : 2500 },
{returnNewDocument:true})
{
  _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
  product: 'Commodity',
  limit: 2500
}
```

Figure 6.7: Output of the `findOneAndReplace` Method

The replaced document is returned.

- **Find and Delete**

MongoDB provides the `findOneAndDelete` method to locate a document and delete it based on the criteria specified in the method. The syntax for the `findOneAndDelete` method is:

```
db.collection.findOneAndDelete(filter, options)
```

Table 6.5 describes the parameters of the `findOneAndDelete` method.

Parameter	Fields	Type	Description
filter	-	document	Specifies the criteria based on which the document must be deleted; if empty, then the first document in the collection is deleted
options		document	Specifies optional fields
	writeConcern	document	Specifies the write concern of the document
	projection	document	Specifies the fields to be returned
	sort	document	Specifies the order in which the documents matched by the filter parameter must be sorted
	maxTimeMS	number	Specifies the maximum time in milliseconds within which the query must run; if exceeded, throws an error
	collation	document	Specifies language-specific rules such as lettercase

Table 6.5: Parameters for the `findOneAndDelete` Method

Consider that the user wants to find all the products that are 'Commodity' and sort the returned documents in ascending order on the basis of the limit field. To delete the first such document returned, the user can execute the query as:

```
db.accounts.findOneAndDelete({ "products" : "Commodity"},  
{sort: {"limit" : 1}})
```

The output of the query is shown in Figure 6.8.

```
sample_analytics> db.accounts.findOneAndDelete({ "products" : "Commodity"}, {sort: {"limit" : 1}})  
{  
  _id: ObjectId("5ca4bbc7a2dd94ee58162458"),  
  account_id: 852986,  
  limit: 7000,  
  products: [  
    'Derivatives',  
    'Commodity',  
    'CurrencyService',  
    'InvestmentFund',  
    'InvestmentStock'  
  ]  
}
```

Figure 6.8: Output of the `findOneAndDelete` Method

The deleted document is returned as the output of this method.

6.3 Database Methods

The database methods provided by MongoDB help to create, run, drop, and fetch data from collections and views in a database. Some of the database methods—`db.dropDatabase`, `db.runCommand`, `db.createCollection`, `db.getSiblingDB`—were covered in previous sessions.

- **Create View**

Views are read-only entities that are created from collections or other views in the same database. Views are created by running an aggregation pipeline on the source collection or view. MongoDB provides a method to create a view based on a collection or another view.

The syntax to create a view is:

```
db.createView(view, source, pipeline, collation)
(or)
db.createCollection(view, source, pipeline, collation)
```

Table 6.6 describes the parameters of the `createView` method.

Parameter	Type	Description
view	string	Specifies the name of the view to be created
source	string	Specifies the name of the source collection or view from which the view must be created
pipeline	array	Specifies the array that has the aggregation pipeline stages as its elements; cannot include the <code>\$out</code> and <code>\$merge</code> stages
collation	document	Specifies language-specific rules such as <code>lettercase</code>

Table 6.6: Parameters for the `createView` Method

Consider that the user wants to create a view named `AccountsLimit5000` from the `accounts` collection in the `sample_analytics` database. This view must contain only those documents where the value of the `limit` field is 5000. To create this view, the user can execute the query as:

```
db.accounts.createView("AccountsLimit5000", "accounts",
[ { $match: { limit: 5000 } } ])
```

The output of the query is shown in Figure 6.9.

```
sample_analytics> db.createView("AccountsLimit5000", "accounts", [ { $match:
{ limit: 5000 } } ] )
{ ok: 1 }
```

Figure 6.9: Output of the `createView` Method

A new view named `AccountsLimit5000` has been created.

To fetch documents from this view, the user can execute the query as:

```
db.AccountsLimit5000.find()
```

The output of the query is shown in Figure 6.10.

```
sample_analytics> db.AccountsLimit5000.find()
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816272e"),
    account_id: 170980,
    limit: 5000,
    products: [
      'InvestmentFund',
      'Brokerage',
      'CurrencyService',
      'Derivatives',
      'InvestmentStock'
    ]
  }
]
```

Figure 6.10: Fetch Documents from a View

- **Get Collection Names**

MongoDB provides the `getCollectionNames` method to get the names of all the collections and views in a database.

Consider that the user wants to get the names of all the collections and views in the `sample_analytics` database. To perform this task, the user can execute the query as:

```
db.getCollectionNames()
```

The output of the query is shown in Figure 6.11.

```
sample_analytics> db.getCollectionNames()
[ 'accounts', 'system.views', 'transactions', 'AccountsLimit5000' ]
```

Figure 6.11: Output of the `getCollectionNames` Method

- **Check Connection**

MongoDB provides the `getMongo` method to test the connection between `mongosh` and the database instance. To test the connection, the user can execute the query as:

```
db.getMongo()
```

The output of the query is shown in Figure 6.12.

```
sample_analytics> db.getMongo()
mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=20
00&appName=mongosh+1.8.2
```

Figure 6.12: Output of the `getMongo` Method

- **Statistics**

MongoDB provides the `stats` method that returns the use state of a database. To get the usage statistics of a database, the user can execute the query as:

```
db.stats()
```

The output of the query is shown in Figure 6.13.

```
sample_analytics> db.stats()
{
  db: 'sample_analytics',
  collections: 3,
  views: 1,
  objects: 3491,
  avgObjSize: 4686.686049842452,
  dataSize: 16361221,
  storageSize: 9797632,
  indexes: 3,
  indexSize: 114688,
  totalSize: 9912320,
  scaleFactor: 1,
  fsUsedSize: 159485407232,
  fsTotalSize: 511258390528,
  ok: 1
}
```

Figure 6.13: Output of the `stats` Method

6.4 Role Management Methods

The role management methods provided by MongoDB help to create, update, and drop roles. These methods also facilitate the users to grant and revoke privileges to roles. Let us discuss a few role management methods in MongoDB.

- **Create Role**

MongoDB provides a method to create a role in the database in which the method is executed. The syntax to create a role is:

```
db.createRole(role, writeConcern)
```

Table 6.7 describes the parameters of the `createRole` method.

Parameter	Field	Type	Description
role	role	string	Specifies the name of the role to be created
	privileges	array	Specifies the privileges that must be granted to the role; if empty, no privileges are granted
	roles	array	Specifies an array of roles from which privileges must be inherited for this role; if empty, no roles are inherited
	authentication Restrictions	array	Specifies the authentication restrictions enforced on the role
writeConcern		document	Specifies the level of write concern

Table 6.7: Parameters of the `CreateRole` Method

Consider that the user wants to create a new role named `mynewrole` on the `sample_analytics` database from the `admin` database.

To create a new role, the user can execute the query as:

```
use admin

db.createRole({ role: "mynewrole", privileges: [
  resource: { db: "sample_analytics", collection: "accounts" },
  actions: ["update", "insert", "remove"] },
  { resource: { db: "", collection: "" }, actions: ["find"] },
  roles: [{ role: "readWrite", db: "sample_analytics" }] },
  { w: "majority", wtimeout: 5000 })
```

The output of the query is as shown in Figure 6.14.

```
sample_analytics> use admin
switched to db admin
admin> db.createRole({ role: "mynewrole", privileges: [
  { resource: { db: "saadmin" } },
  { resource: { db: "sample_analytics", collection: "accounts" },
    actions: ["update", "insert", "remove"] },
  { resource: { db: "", collection: "" }, actions: ["find"] },
  roles: [{ role: "readWrite", db: "sample_analytics" }] },
  { w: "majority", wtimeout: 5000 })
{ ok: 1 }
```

Figure 6.14: Output of the `createRole` Method

The `mynewrole` role is created.

- **Get Role**

MongoDB provides the `getRole` method to get the roles from which privileges are inherited for this role. The syntax for this method is:

```
db.getRole(rolename, arguments)
```

Table 6.8 describes the parameters of the `getRole` method.

Parameter	Field	Type	Description
rolename	-	string	Specifies the name of the role
arguments	showAuthentication Restrictions	boolean	Includes the IP address from which the users of this role can connect to the database as part of the output if set to <code>true</code>

Parameter	Field	Type	Description
	showBuiltInRoles	boolean	Includes built-in roles as part of the output if set to true
	showPrivileges	boolean	Includes direct privileges and privileges inherited from other roles as part of the output if set to true

Table 6.8: Parameters of the `getRole` Method

Consider that the user wants to get the details about the `mynewrole` role. To perform this task, the user can execute the query as:

```
db.getRole( "mynewrole" )
```

The output of the query is shown in Figure 6.15.

```
admin> db.getRole("mynewrole")
{
  _id: 'admin.mynewrole',
  role: 'mynewrole',
  db: 'admin',
  roles: [ { role: 'readWrite', db: 'sample_analytics' } ],
  inheritedRoles: [ { role: 'readWrite', db: 'sample_analytics' } ],
  isBuiltIn: false
}
```

Figure 6.15: Output of the `getRole` Method

Consider that the user wants to get the roles and privileges of the `mynewrole` role. To perform this task, the user can execute the query as:

```
db.getRole( "mynewrole", { showPrivileges: true } )
```

The output of the query is shown in Figure 6.16.

```
admin> db.getRole( "mynewrole", { showPrivileges: true } )
{
  _id: 'admin.mynewrole',
  role: 'mynewrole',
  db: 'admin',
  privileges: [
    {
      resource: { db: 'sample_analytics', collection: 'accounts' },
      actions: [ 'insert', 'remove', 'update' ]
    },
    { resource: { db: '', collection: '' }, actions: [ 'find' ] }
  ],
  roles: [ { role: 'readWrite', db: 'sample_analytics' } ],
  inheritedRoles: [ { role: 'readWrite', db: 'sample_analytics' } ],
  inheritedPrivileges: [
    {
      resource: { db: 'sample_analytics', collection: 'accounts' },
      actions: [ 'insert', 'remove', 'update' ]
    },
    { resource: { db: '', collection: '' }, actions: [ 'find' ] },
    {
      resource: { db: 'sample_analytics', collection: '' },
      actions: [
        'changeStream'
      ]
    }
  ]
}
```

Figure 6.16: Output of the `getRole` Method with Privileges

The roles and privileges associated with the `mynewrole` role are displayed as output.

- **Update Role**

MongoDB provides the `updateRole` method to update privileges, roles, and restrictions of a user-defined role. The syntax for this method is:

```
db.updateRole(rolename, update, writeConcern)
```

Table 6.9 describes the parameters of the `updateRole` method.

Parameter	Field	Type	Description
rolename	-	string	Specifies the name of the role
update	privileges	array	Specifies the privileges that must be granted to the role; mandatory if <code>roles</code> field is not mentioned
	roles	array	Specifies the roles from which

Parameter	Field	Type	Description
			this role inherits; mandatory if privileges field is not mentioned
	authentication Restrictions	array	Specifies the authentication restrictions enforced by the server on the role such as IP addresses from which the users of this role can connect to the database

Table 6.9: Parameters of the updateRole Method

Consider that the user wants to update the roles and privileges of the mynewrole role. To perform this task, the user can execute the query as:

```
use admin

db.updateRole("mynewrole", { privileges: [{ resource:
{ db: "sample_analytics", collection: "accounts" },
actions: ["update", "createCollection",
"createIndex"] }], roles: [{ role: "read", db:
"sample_analytics" }] }, { w: "majority" })
```

The output of the query is shown in Figure 6.17.

```
admin> db.updateRole("mynewrole", { privileges: [{ resource: { db: "sample_analytics"
, collection: "accounts" }, actions: ["update", "createCollection", "createIndex"] }]
, roles: [{ role: "read", db: "sample_analytics" }] }, { w: "majority" })
{ ok: 1 }
```

Figure 6.17: Output of the updateRole Method

To view the updated role, the user can execute the query as:

```
db.getRole( "mynewrole")
```

The output of the query is shown in Figure 6.18.

```
admin> db.getRole( "mynewrole")
{
  _id: 'admin.mynewrole',
  role: 'mynewrole',
  db: 'admin',
  roles: [ { role: 'read', db: 'sample_analytics' } ],
  inheritedRoles: [ { role: 'read', db: 'sample_analytics' } ],
  isBuiltin: false
}
...
```

Figure 6.18: Output After the Role Update

It can be seen that the roles of the `mynewrole` role is updated.

- **Drop Role**

MongoDB provides the `dropRole` method to delete a user-defined role from the database in which the role was created. The syntax for this method is:

```
db.dropRole(roleName, writeConcern)
```

Table 6.10 describes the parameters of the `dropRole` method.

Parameter	Field	Type	Description
role	role	string	Specifies the name of the role to be dropped
writeConcern		document	Specifies the level of write concern

Table 6.10: Parameters of the `dropRole` Method

Consider that the user wants to drop the `mynewrole` role from the `admin` database. To drop the role, the user can execute the query as:

```
db.dropRole( "mynewrole", { w: "majority" } )
```

The output of the query is shown in Figure 6.19.

```
admin> db.dropRole( "mynewrole", { w: "majority" } )
{ ok: 1 }
```

Figure 6.19: Output of the dropRole Method

The `mynewrole` method is dropped.

6.5 User Management Methods

User management methods provided by MongoDB help to create a user, get roles to the user, change user password, and drop the user. Let us now learn about a few user management methods in MongoDB.

- **Create User**

MongoDB provides the `createUser` method to create a user in the database in which the method is executed. The syntax to create a user is:

```
db.createUser(user, writeConcern)
```

Table 6.11 describes the parameters of the `createUser` method.

Parameter	Field	Type	Description
user	user	string	Specifies the name of the user to be created
	pwd	string	Specifies the password of the user which can be a cleartext string or a passwordPrompt that prompts the user for password
	customData	document	Contains random information about the user that the admin wishes to store such as employee ID or full name of the user
	roles	array	Specifies the roles assigned to the user
	authentication Restriction	array	Specifies the authentication restrictions enforced on the user

Parameter	Field	Type	Description
writeConcern		document	Specifies the level of write concern

Table 6.11: Parameters of the `CreateUser` Method

Consider that the user wants to create a user named `user_account1` with `readWrite` and `dbAdmin` roles for the `accounts` collection in the `sample_analytics` database. To create a new user, the user can execute the query as:

```
db.createUser({ user: "user_account1", pwd:  
    passwordPrompt(), roles: ["readWrite", "dbAdmin"] })
```

When executed, the query prompts for a password. The user can enter `account1` as the password. The output of the query is shown in Figure 6.20.

```
switched to db sample_analytics  
sample_analytics> db.createUser({user: "user_account1", pwd: passwordPrompt(),  
...     roles: [ "readWrite", "dbAdmin" ] })  
Enter password  
*****{ ok: 1 }
```

Figure 6.20: Output of the `createUser` Method

The user named `user_account1` is created. Users can also be created without specifying any roles.

- **Change User Password**

MongoDB provides the `changeUserPassword` method to change the password for the user created in the current database. The syntax for this method is:

```
db.changeUserPassword(username, password)
```

Table 6.12 describes the parameters of the `changeUserPassword` method.

Parameter	Type	Description
username	string	Specifies the name of the user for whom the password must be changed
password	string	Specifies the new password of the user which can be a cleartext string or a passwordPrompt that prompts the user for password

Table 6.12: Parameters of the changeUserPassword Method

Consider that the user wants to change the password for the user_account1 user. To change the password, the user can execute the query as:

```
db.changeUserPassword("user_account1", passwordPrompt())
```

When executed, the query prompts for a password. Enter user1 as the new password. The output of the query is shown in Figure 6.21.

```
sample_analytics> db.changeUserPassword("user_account1", passwordPrompt())
Enter password
*****{ ok: 1 }
```

Figure 6.21: Output of the changeUserPassword Method

The password for the user_account1 user is now changed to user1.

- **Get User**

MongoDB provides the getUser method to get user details such as password, privileges, and custom data of the user. The syntax for this method is:

```
db.getUser(username, args)
```

Table 6.13 describes the parameters of the `getUser` method.

Parameter	Field	Type	Description
username	-	string	Specifies the name of the user for whom the details must be obtained
args	showCredentials	boolean	Displays the password of the user if set to true
	showCustomerData	boolean	Displays the custom data if set to true
	showPrivileges	boolean	Displays the entire privilege set of the user if set to true
	showAuthentication Restrictions	boolean	Displays the authentication restrictions of the user such as the allowed IP addresses to connect to the database if set to true

Table 6.13: Parameters of the `getUser` Method

Consider that the user wants to get details about the `user_account1` user in the `sample_analytics` database. To perform this task, the user can execute the query as:

```
db.getUser("user_account1")
```

The output of the query is shown in Figure 6.22.

```
sample_analytics> db.getUser("user_account1")
{
  _id: 'sample_analytics.user_account1',
  userId: new UUID("19cb9e6d-f088-402c-b9e9-7a041dee3321"),
  user: 'user_account1',
  db: 'sample_analytics',
  roles: [
    { role: 'readWrite', db: 'sample_analytics' },
    { role: 'dbAdmin', db: 'sample_analytics' }
  ],
  mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ]
}
```

Figure 6.22: Output of the `getUser` Method

- **Drop User**

MongoDB provides the `dropUser` method to drop the user from the database.

The syntax for this method is:

```
db.dropUser(username, writeConcern)
```

Table 6.14 describes the parameters of the `dropUser` method.

Parameter	Field	Type	Description
username	-	string	Specifies the name of the user for whom the details must be obtained
args	showCredentials	boolean	Displays the password of the user if set to true
	showCustomerData	boolean	Displays the user data if set to true
	showPrivileges	boolean	Displays the entire privilege set of the user if set to true
	showAuthentication Restrictions	boolean	Displays the authentication restrictions of the user if set to true

Table 6.14: Parameters of the `dropUser` Method

Consider that the user wants to drop the `user_account1` user in the `sample_analytics` database. To perform this task, the user can execute the query as:

```
db.dropUser("user_account1", {w: "majority",  
wtimeout: 5000})
```

The output of the query is shown in Figure 6.23.

```
sample_analytics> db.dropUser("user_account1", {w: "majority", wtimeout: 5000})  
{ ok: 1 }
```

Figure 6.23: Output of the dropUser Method

6.6 Summary

- MongoDB Shell provides various methods such as collection methods, database methods to create, insert, update, and delete documents, collections, and databases.
- MongoDB Shell includes methods to locate documents based on specified conditions and update, replace, or delete the first document in the output.
- MongoDB allows views to be created by running aggregation pipelines on the source collections or other views.
- MongoDB Shell provides methods that fetch statistical information about the collections or users.
- MongoDB Shell also provides role management methods and user management methods to create, update, and drop roles and users in a database.

Test Your Knowledge

1. Which of the following options are used to count all the documents in the inventory collection?
 - a. db.inventory.countDocuments({})
 - b. db.inventory.countDocuments()
 - c. db.inventory.countDocuments().all()
 - d. db.inventory.countAllDocuments()

2. Which of the following option is the correct syntax of the findOneAndUpdate method?
 - a. db.collection.findOneAndUpdate(filter, query, update)
 - b. db.collection.findOneAndUpdate(field, update, options)
 - c. db.collection.findOneAndUpdate(filter, update, options)
 - d. db.collection.findOneAndUpdate(query, field, update)

3. Consider that you have created a collection named product_price with the fields: product_id, product_name, and product_price. Which of the following query will you use to create a view named Product_price_lessthan_2000 from the product_price collection in the product database?
 - a. db.createView("Product_price_lessthan2000", "product", [{ \$match: { product_price: {\$lt:2000} } }])
 - b. db.product.createView("Product_price_lessthan2000", "product", [{ \$match: { product_price: {\$lt:2000} } }])
 - c. db.createView("Product_price_lessthan2000", "product", [{ \$group: { product_price: {\$lt:2000} } }])
 - d. db.createView("Product_price_lessthan2000", "product", [{ \$project: { product_price: {\$lt:2000} } }])

4. Which of the following query will you use to display the roles and privileges of the role named user1_role from the product database.
 - a. db.viewRole("user1_role", { privileges: true })
 - b. db.getRole("user1_role", { showPrivileges: true })
 - c. db.product.getRole("user1_role", { showPrivileges: true })
 - d. db.viewRolePrivileges("mynewrole", { showPrivileges: true })

5. Which of the following user management method will allow you to change the password of a user for the current database?
- a. db.userPasswordchange(username, password)
 - b. db.PasswordChange(username, password)
 - c. db.userChangePassword(username, password)
 - d. db.changeUserPassword(username, password)

Answers to Test Your Knowledge

1	a, b
2	c
3	a
4	b
5	d

Try it Yourself

1. Create a database named Student and a collection named sub_marks.

2. Insert the following four documents into the sub_marks collection.

```
[  
{  
    Name: "Wilson Churchill",  
    rollno:1234,  
    age:16,  
    program_language:["C", "C++", "Java"],  
    Total_marks:280  
,  
    {  
        Name: "Adam Smith",  
        rollno:2457,  
        age:17,  
        program_language:["Java", "C++", "Python"],  
        Total_marks:270  
,  
        {  
            Name: "Rita Richard", rollno:1221,  
            age:15,  
            program_language:["Perl", "Ruby", "Java"], Total_marks:268  
,  
            {  
                Name: "Michael Franklin",  
                rollno:2134,  
                age:15,  
                program_language:["Python", "Java", "C#"], Total_marks:290  
            }  
        }  
    }  
}
```

Using the sub_marks collection, perform the following tasks:

3. Count the number of students who have learned 'java' as one of the subjects.
4. Fetch the distinct subjects learned by students.
5. Use the collection method to find the student named 'Rita Richard' and update the Total_marks field as 288.
6. Create a view named subject_marks which contains only those documents where the Total_marks field is greater than 275.

7. Display the use state of the database Student.
8. Create a new role named `role1` in the Student database and grant permissions to `insert`, `update`, and `remove` data from the `stu_marks` collection.
9. Display detailed information about the `role1` role.
10. Update `role1` and grant permission to `update` the new collection `stu_detail` in the Student database. Also, add a new role named `user2` and grant `read` permission to the `stu_detail` collection.
11. Drop `role1` from the Student database.
12. Create a user `stu_user1` for the `stu_mark` collection with `readwrite` permission.
13. Display the details of the user `stu_user1` with password.



SESSION 7

MONGODB INDEXING

Learning Objectives

In this session, students will learn to:

- Explain indexes
- Describe the types of indexes
- List the advantages of using the indexes
- Explain how to create and drop indexes

In a book, the first page usually contains a table of contents that lists the chapters, topics, and subtopics with page numbers. This helps in navigating to a particular chapter, topic, or subtopic in the book instead of iterating through all the pages to find the required piece of information. Similarly, indexes are ready reckoners for the database. They help in reaching the required data in a short duration of time with less overhead.

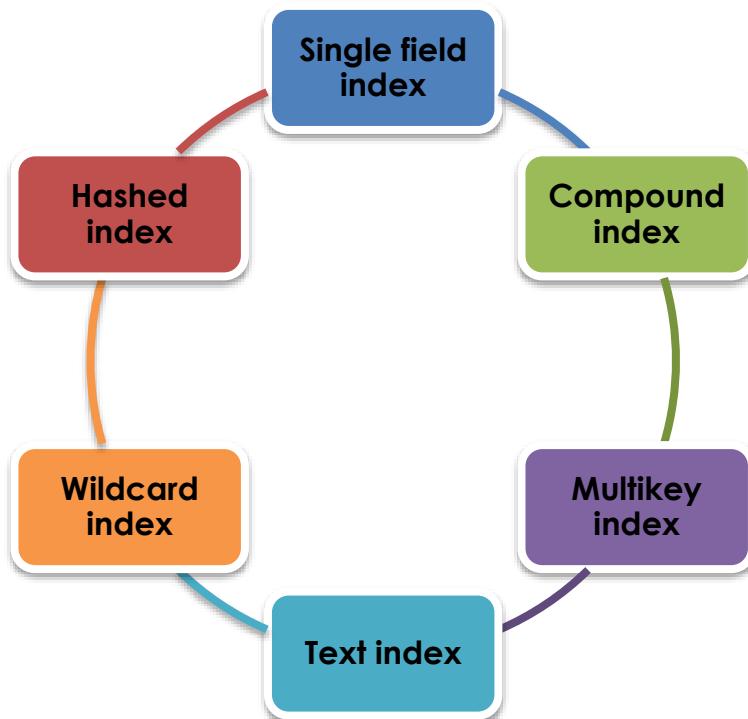
This session will explain indexes and different types of indexes in MongoDB. It will also explain the benefits of using indexes. Finally, it will explain how to create and drop indexes.

7.1 Introduction to Indexes

Indexes are nothing but data structures that store the values of a field or multiple fields in ascending or descending order. After the index is created, the user can easily query for a specific document with a low response time. This is because the database engine looks up the index to find a particular document instead of traversing through the entire collection to locate a specific document. This improves the query performance and reduces the overhead on the database engine of traversing through whole collections. The ordering of the index helps in easily accessing data in equality-based and range-based queries.

By default, MongoDB adds the `_id` field for each document in the collection. This field is unique for each document. It prevents duplicate documents from getting inserted into the collection. An index is created by default on the `_id` field and it cannot be dropped. It is known as the Default index.

MongoDB supports different types of indexes:



7.2 Single Field Index

MongoDB allows users to create an index on a single field of documents in a collection. The field on which index is created must be a prominent field that is used when searching for documents in that collection. Such an index created on a single field is called as single field index.

The general syntax for creating a single field index is:

```
db.Collection_name.createIndex(key,option,commitQuorum)
```

The `createIndex` command takes the parameters described in Table 7.1.

Parameter	Description
key	It is a key-value pair, where the key specifies the field on which the index must be created, and the value specifies the order of the index. It takes the value 1 for ascending order and the value -1 for descending order. This is a required parameter.
option	It is a document that contains options, which specify how the index should be created. This is an optional parameter.
commitQuorum	It specifies how many replica sets, which contain data and participate in voting, must respond with successful index creation before the primary marks the index as ready to use. This is an optional parameter.

Table 7.1: Parameters of the `createIndex` Command

The `option` parameter in the `createIndex` command takes the parameters described in Table 7.2.

Parameter	Description
background	This is a boolean value. If set to true, it specifies that the index must be built in the background without interfering with the other database activities. By default, this parameter is set to false.
unique	This is a boolean value. If set to true, it creates a unique index that does not allow insertion of more than one document with same index key. By default, this parameter is set to false.

Table 7.2: List of Options for Creating Single Field Index

<code>name</code>	This is a <code>string</code> value that specifies a name for the index. If a name is not specified, MongoDB assigns a name which is a combination of indexed field names and the sort order.
<code>sparse</code>	This is a <code>boolean</code> value. If set to <code>true</code> , it includes only those documents in the index that have the specified field. By default, this parameter is set to <code>false</code> .
<code>expireAfterSeconds</code>	This is an <code>integer</code> value that specifies time in seconds. This time is Time-To-Live (TTL) that controls how long the documents in the collection will be retained by MongoDB.

To understand the importance of indexing, in the `sample_analytics` database, let us find a document that has `account_id` as 781775 in the `grades` collection. To view the execution details of the query, let us also use the `explain("executionStats")` method. To do this, the user can run the query:

```
db.accounts.find({"account_id":781775}).explain
    ("executionStats")
```

Figure 7.1 shows the output of this query.

```

sample_analytics> db.accounts.find({"account_id":781775}).explain("executionStats")
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'sample_analytics.accounts',
    indexFilterSet: false,
    parsedQuery: { account_id: { '$eq': 781775 } },
    queryHash: 'C7AD3977',
    planCacheKey: 'C7AD3977',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { account_id: { '$eq': 781775 } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 9,
    totalKeysExamined: 0,
    totalDocsExamined: 1746,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { account_id: { '$eq': 781775 } },
      nReturned: 1,
      executionTimeMillisEstimate: 6,
      works: 1748,
      advanced: 1,
      needTime: 1746,
      needYield: 0,
      saveState: 1,
      restoreState: 1,
      isEOF: 1,
      direction: 'forward',
    }
  }
}

```

Figure 7.1: Output of `explain("executionStats")` Method Before Indexing

Note that in the execution status, the total number of documents examined is shown as 1746.

Now, let us create an index for the accounts collection in the sample_analytics database based on the account_id field in the ascending order. To do this, the query to run is:

```
db.accounts.createIndex({ "account_id":1 })
```

The index is created as specified in the query.

Let us now run the previous query again. Figure 7.2 shows the output of the query.

```

sample_analytics> db.accounts.find({"account_id":781775}).explain("executionStats")
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'sample_analytics.accounts',
    indexFilterSet: false,
    parsedQuery: { account_id: { '$eq': 781775 } },
    queryHash: 'C7AD3977',
    planCacheKey: 'E975A254',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { account_id: 1 },
        indexName: 'account_id_1',
        isMultiKey: false,
        multiKeyPaths: { account_id: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { account_id: [ '[781775, 781775]' ] }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 8,
    totalKeysExamined: 1,
    totalDocsExamined: 1,
    executionStages: {
      stage: 'FETCH',
      nReturned: 1,
    }
  }
}

```

Figure 7.2: Output of `explain ("executionStats")` Method After Indexing

Note that in the output the number of documents examined is 1. This indicates that after the index is created, the number of documents examined reduces, thereby, reducing the time required to complete the query execution.

Now, let us create another index for the `accounts` collection based on the `limit` field in the descending order. To do this, the query to run is:

```
db.accounts.createIndex({"limit": -1})
```

The index is created as specified in the query. To view all the indexes created for the `accounts` collection, run the query as:

```
db.accounts.getIndexes()
```

Figure 7.3 shows the output of this query.

```
sample_analytics> db.accounts.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { account_id: 1 }, name: 'account_id_1' },
  { v: 2, key: { limit: -1 }, name: 'limit_-1' }
]
```

Figure 7.3: Indexes Created for the accounts Collection

Note that the index on the `_id` field is created by default when the collection is created. The other two indexes are user created indexes.

7.3 Compound Index

In MongoDB, users can create indexes based on multiple fields. These types of indexes are known as compound indexes. Users can create indexes that reference a maximum of 32 fields. For each field, the sort order must be specified. The syntax for creating a compound index is:

```
db.collections.createIndex({fieldName1:ord1,
                            fieldName2:ord2, ...} )
```

The order in which the fields are specified in the syntax is important for the index to function efficiently. The documents will be sorted on `fieldName1`, then within each value of `fieldName1`, the documents are sorted on `fieldName2`, and so on. Changing the order of the fields in the `createIndex` command will produce different results in terms of efficiency of the index.

In the `sample_training` database, consider that the user wants to create a compound index named `compoundIndex` for the `grades` collection. The index must first be created on the `class_id` field in the ascending order and then on the `student_id` field in the descending order. To do this, the user can execute the query as:

```
db.grades.createIndex({class_id: 1, student_id:-1},
                      {name: "compoundIndex"})
```

Figure 7.4 shows the output of this query.

```
sample_training> db.grades.createIndex({class_id: 1, student_id:-1},  
  {name: "compoundIndex"})  
compoundIndex
```

Figure 7.4: Compound Index Created

To view the indexes created for the `grades` collection, run the query as:

```
db.grades.getIndexes()
```

Figure 7.5 shows the output of this query.

```
sample_training> db.grades.getIndexes()  
[  
  { v: 2, key: { _id: 1 }, name: '_id_' },  
  { v: 2, key: { class_id: 1, student_id: -1 }, name: 'compoundIndex' }  
]
```

Figure 7.5: Indexes Created for the `grades` Collection

7.4 Multikey Index

Indexes created on array fields are multikey indexes. In this case, an index is created on each element in the array. Multikey indexes facilitate efficient querying of the arrays in documents. The syntax for creating a multikey index is:

```
db.collections.createIndex(arrayname : type)
```

In the syntax, `arrayname` specifies the name of the array field on which the index must be created, and `type` specifies the sort order. The index must not be specified as a multikey index because if the specified field is an array, MongoDB will automatically create a multikey index.

Consider the `accounts` collection of the `sample_analytics` database. The user wants to create a multikey index named `multiKeyIndex` on the `products` field, which is an array, in the ascending order. To do so, the user can execute the query as:

```
db.accounts.createIndex({products:1},  
  {name: "multiKeyIndex"})
```

Multikey index is created for the `products` field.

To view the indexes created on the `accounts` collection, the user can execute the query as:

```
db.accounts.getIndexes()
```

Figure 7.6 shows the output of this query.

```
sample_analytics> db.accounts.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { account_id: 1 }, name: 'account_id_1' },
  { v: 2, key: { limit: -1 }, name: 'limit_-1' },
  { v: 2, key: { products: 1 }, name: 'multiKeyIndex' }
]
```

Figure 7.6: Indexes Created for the `accounts` Collection

7.4.1 Compound Multikey Index

Users can also create a compound multikey index. This index can have only one array field as an indexed field. If the documents in a collection have two array fields A and B, the index can include field A or field B, but not both the fields. If some documents in a collection have field A as an array and others have field B as an array, then the index can include both the fields.

If a compound multikey index is created on a collection, any document that violates the index cannot be inserted into the collection. That is, say an index has been created to include field A, which exists for some documents. The index includes field B, which exists for the other documents. In this case, a document that includes both array fields A and B cannot be inserted into the collection.

7.4.2 Index on Fields Embedded in Arrays

An array field may have other fields embedded in it. Indexes can be created on these embedded fields. In the `sample_training` database, the `grades` collection has an array field named `scores`. This array has `type` and `score` as embedded fields, as shown in Figure 7.7.

```
{
  _id: ObjectId("56d5f7eb604eb380b0d8d8e5"),
  student_id: 2,
  scores: [
    { type: 'exam', score: 89.1838715782135 },
    { type: 'quiz', score: 60.78999591419918 },
    { type: 'homework', score: 80.1394331814776 },
    { type: 'homework', score: 87.07482638428962 }
  ],
  class_id: 452
},
{
  _id: ObjectId("56d5f7eb604eb380b0d8d8ce"),
  student_id: 0,
  scores: [
    { type: 'exam', score: 78.40446309504266 },
    { type: 'quiz', score: 73.36224783231339 },
    { type: 'homework', score: 46.980982486720535 },
    { type: 'homework', score: 76.67556138656222 }
  ],
  class_id: 339
}
```

Figure 7.7: Data in the grades Collection

To create a multikey index on the `score.type` and `score.score` fields, the user can execute the query as:

```
db.grades.createIndex({ "scores.type":1, "scores.score":1 },
                      {name: "embeddedFieldIndex"})
```

Figure 7.8 shows the output of this query.

```
sample_training> db.grades.createIndex({ "scores.type":1, "scores.score":1 },
{name: "embeddedFieldIndex"})
embeddedFieldIndex
```

Figure 7.8: Multikey Index on Embedded Fields in an Array

Indexes created on fields embedded in the array help users to run queries that include one of the indexed fields or both the indexed fields. For example, the user can view the documents where `scores.type` is `exam`. To do so, the user can use the query as:

```
db.grades.find( { "scores.type": "exam" } )
```

The user can also view the documents where `scores.type` is quiz and `scores.score` is greater than 50. To do so, the user can use the query as:

```
db.grades.find( { "scores.type": "quiz",  
    "scores.score": { $gt: 50 } } )
```

The indexes can also be used to sort the data. Consider that the user wants to view the documents where `scores.type` is exam and sort `scores.score` for each type of value in the ascending order. The user can use the query as:

```
db.grades.find( { "scores.type": "exam" }  
).sort( { "scores.score": 1 } )
```

7.5 Text Index

MongoDB allows users to create indexes on texts or strings. These types of indexes are called text indexes. Text indexes can be created on a text field or on array of text elements and help in executing text-based searches. A document can have only one text index, but the index can include many fields. The syntax for creating a text Index is:

```
db.collections.createIndex(key, options, commitQuorum)
```

In this syntax,

- `key` specifies the field name on which the index must be created
- `option` specifies how the index must be created
- `commitQuorum` specifies the smallest number of replica sets that must report the successful creation of the index before the index is marked as ready for use

The `option` parameter takes various parameters as described in Table 7.3.

Parameter	Description
<code>weights</code>	This is a document that takes values ranging from 1 to 99999. These values indicate the significance of an indexed field with respect to the other fields in terms of score.

default_language	This is a string value that specifies the language, which is used to determine rules for the stemmer and tokenizer and the list of stop words. By default, this parameter is set to English.
language_override	This is a string value that specifies the language to use to override the default language. By default, this parameter is set to language.
textIndexVersion	This is an integer value which is optional and is used to specify a version number for the index. If this parameter is used, the specified version number will replace the default version number of the index.

Table 7.3: List of Options for Creating Text Index

Consider the `inspections` collection of the `sample_training` database. The user wants to create a text index on the `business_name` field. To do this, the user can execute the query as:

```
db.inspections.createIndex({"business_name":1})
```

7.6 Wildcard Index

When creating any of the different types of indexes discussed until now, the field on which the index must be created is specified as a parameter in the `createIndex` command. Now, since MongoDB allows the schema to be flexible or unstructured, it could happen that all documents do not have similar fields on which the index can be created. In such cases, wildcard indexes are created. Wildcard indexes can be created on:

- Unknown fields
- Unstructured schema

7.6.1 Wildcard Index on a Specific Field

The syntax for creating a wildcard index on a specific field is:

```
db.collections.createIndex({"fieldA.$**" : 1})
```

This syntax is used to create an index on all the values in the `fieldA` field. If this field is a document or an array, the index iterates through the document or array and creates an index for each field in the document or array.

For example, the `inspections` collection in the `sample_training` database contains a field named `address`, which contains nested fields such as `city`, `zip`, `street`, and `number` as shown in Figure 7.9.

```
[  
 {  
   _id: ObjectId("56d61033a378eccde8a83550"),  
   id: '10057-2015-ENFO',  
   certificate_number: 6007104,  
   business_name: 'LD BUSINESS SOLUTIONS',  
   date: 'Feb 25 2015',  
   result: 'Pass',  
   sector: 'Tax Preparers - 891',  
   address: {  
     city: 'NEW YORK',  
     zip: 10030,  
     street: 'FREDERICK DOUGLASS BLVD',  
     number: 2655  
   },  
   multi: true  
 },
```

Figure 7.9: Fields in the `inspection` Collection

For creating an index on the `address` field, the query to use is:

```
db.inspections.createIndex( { "address.$**" : 1 } )
```

This index can now help with queries such as:

```
db.inspections.find({ "address.city" : "NEW YORK" })
```

```
db.inspections.find({ "address.zip" : {$gt:10029} })
```

7.6.2 Wildcard Index on All Fields

The syntax for creating a wildcard index on all fields in a document is:

```
db.collection.createIndex( { "$**" : 1 } )
```

This syntax is used to create an index on all the fields of the documents in the specified collection. If this field is a document or an array, the index iterates through the document or array and creates an index for each field in the document or array.

For example, to create an index on all the fields in the `inspections` collection, the user can run the query as:

```
db.inspections.createIndex( { "$**" : 1 } )
```

7.6.3 Wildcard Indexes Including or Excluding Multiple Fields

Users can specify multiple fields for indexing when creating wildcard indexes. The syntax for creating this type of wildcard index is:

```
db.collection.createIndex({ "$**" : 1 },
{ "wildcardProjection" : { "fieldA" : 1,
"fieldB.fieldC" : 1 } })
```

In this syntax, `fieldA` and `fieldB.fieldC` are the specified fields on which the index must be created. For example, for the `transactions` collection of the `sample_analytics` database, the user wants to create an index on all the values in the `name` and `customer.gender` fields. The user can run the query as:

```
db.sales.createIndex({ "$**" : 1 },
{ "wildcardProjection" : { "name" : 1,
"customer.gender" : 1 } })
```

Similarly, users can specify multiple fields that should not be included in the index when creating wildcard indexes.

The syntax for creating this type of wildcard index is:

```
db.collection.createIndex({ "$**" : 1 },
{ "wildcardProjection" : { "fieldA" : 0,
"fieldB.fieldC" : 0 } })
```

In this syntax, `fieldA` and `fieldB.fieldC` are the specified fields which must not be included in the index.

For example, for the `grades` collection, to create an index on all the values except the values in the `class_id` and `scores.score` fields, the user can run the query as:

```
db.grades.createIndex( { "$**" : 1 }, {  
    "wildcardProjection" : { "class_id" : 0,  
        "scores.score" : 0 } } )
```

7.7 Hashed Index

Hashing involves using an algorithm to compute the hash value for each of the values in the specified field to be indexed. A hash index is created using these hashed values. Hash indexes cannot be created for an array field or a document field. The syntax for creating a hashed index field is:

```
db.collections.createIndex({fieldname: "hashed"})
```

For example, to create a hashed index for the `student_id` field of the `grades` collection in the `sample_training` database, users can run the query as:

```
db.inspections.createIndex({student_id: "hashed"})
```

Hashed indexes can also be included in compound indexes. The syntax for creating a compound index with a hashed index is:

```
db.collections.createIndex ({"fieldA" :  
    1, "fieldB" : "hashed", "fieldC" : -1 })
```

Consider that the user wants to create a compound index with a hashed index for the `grades` collection. The index must sort the `student_id` field in the ascending order, hash the `class_id` field, and sort the `scores` field in the descending order. The users can run the query as:

```
db.inspections.createIndex({"student_id" : 1,  
    "class_id" : "hashed", "scores" : -1})
```



Hashed indexes are exceedingly used in database sharding.

7.8 Drop Indexes

Created indexes can be dropped using the `dropIndexes` command. The syntax for `dropIndexes` command is:

```
db.collection.dropIndexes({indexes})
```

The `indexes` parameter specifies the index or indexes to be dropped. If the parameter is omitted, all the indexes created for the specified collection are dropped, except the default index created on the `_id` field.

To drop indexes created on the `inspection` collection in the `sample_training` database, let us first view the indexes created using the query as:

```
db.inspections.getIndexes()
```

Figure 7.10 shows the output of this query.

```
sample_training> db.inspections.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id' },
  { v: 2, key: { business_name: 1 }, name: 'business_name_1' },
  { v: 2, key: { 'address.$**': 1 }, name: 'address.$**_1' },
  { v: 2, key: { '$**': 1 }, name: '$**_1' },
  { v: 2, key: { student_id: 'hashed' }, name: 'student_id_hashed' },
  {
    v: 2,
    key: { student_id: 1, class_id: 'hashed', scores: -1 },
    name: 'student_id_1_class_id_hashed_scores_-1'
  }
]
```

Figure 7.10: List of Indexes Created for the `inspections` Collection

To drop the ascending index created on the `business_name` field, the user can run the query as:

```
db.inspections.dropIndexes("business_name_1")
```

MongoDB drops the specified index.

To drop multiple indexes from the `inspections` collection, an array of the index names to be dropped must be passed to the `dropIndexes` command. To do so, the user can run the query as:

```
db.inspections.dropIndexes(["address.$**_1", "$**_1"])
```

To view the remaining indexes on the `inspections` collection, the user can run the query as:

```
db.inspections.getIndexes()
```

Figure 7.11 shows the output of this query.

```
sample_training> db.inspections.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { student_id: 'hashed' }, name: 'student_id_hashed' },
  {
    v: 2,
    key: { student_id: 1, class_id: 'hashed', scores: -1 },
    name: 'student_id_1_class_id_hashed_scores_-1'
  }
]
```

Figure 7.11: List of Indexes Created for the `inspections` Collection

To drop all the indexes created for the `inspections` collection, the user can run the query as:

```
db.inspections.dropIndexes()
```

To view the remaining indexes on the `inspections` collection, the user can run the query as:

```
db.inspections.getIndexes()
```

Figure 7.12 shows the output of this query.

```
sample_training> db.inspections.getIndexes()
[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```

Figure 7.12: List of Indexes Created for the `inspections` Collection

7.9 Summary

- Indexes are data structures that store the values of a field or multiple fields in ascending or descending order.
- Single key index is created on a single field by specifying the order of sorting.
- Compound index is the index created on more than one field.
- Multikey index is an index created on an array field.
- Wildcard index helps in building indexes on unknown fields.
- Hashed indexes are the indexes created by hashing the value in the indexed field.
- All indexes created for a collection can be dropped except the default index created on the `_id` field.

Test Your Knowledge

1. Which of the following statements are true about creating indexes in MongoDB?
 - a. It provides a default index named `_id` for each collection.
 - b. Default indexes cannot be dropped.
 - c. An index cannot hold references to more than one field of the documents.
 - d. Indexes store the references in ascending or descending order.
2. Which of the following option is the correct syntax for creating multikey index?
 - a. `db.collections.createIndex(<fieldname, type>)`
 - b. `db.collections.createIndex(<arrayname, type>)`
 - c. `db.collections.createIndex(<indexname>)`
 - d. `db.collections.createIndex(<key, type>)`
3. Which of the following is the correct syntax for creating a wildcard index on all fields in a document except the `_id` field?
 - a. `db.collection.createIndex({ "#**" : 1 })`
 - b. `db.collection.createIndex({ "&**" : 1 })`
 - c. `db.collection.createIndex({ "$**" : 1 })`
 - d. `db.collection.createIndex({ "****" : 1 })`
4. Which of the following statements are true about wildcard index?
 - a. Users can create indexes on unknown fields.
 - b. Users can create indexes on unstructured schema.
 - c. Users can create the index either using `createIndex` or `createIndexes` commands.
 - d. Wildcard index does not omit the `_id` field by default.
5. Which of the following data type is not supported by hash index?
 - a. strings
 - b. integer
 - c. double
 - d. arrays

Answers to Test Your Knowledge

1	a, b, d
2	b
3	c
4	a, b, c
5	d

Try it Yourself

1. Create a database named Customer and a collection named customer_purchase.
2. Insert the following four documents into the customer_purchase collection.

```
[  
{  
customer_id: 1231,  
customer_name:"Alexander",  
city:"Atlanta",  
purchased:[  
{type:'diary',amount:2000},  
{type:'dried_goods',amount:1500},  
{type:'snacks',amount:0},  
{type:'care_products', amount:200},  
{type:'vegetable_fruits',amount:1000}  
],  
{  
customer_id: 1267,  
customer_name:"Daniel",  
city:"Boston",  
purchased:[  
{type:'diary',amount:1000},  
{type:'dried_goods',amount:1200},  
{type:'snacks',amount:200},  
{type:'care_products', amount:700},  
{type:'vegetable_fruits',amount:500}  
],  
{  
customer_id: 1342,  
customer_name:"Jackson",  
city:"Atlanta",  
purchased:[  
{type:'diary',amount:600},  
{type:'dried_goods',amount:700},  
{type:'snacks',amount:100},  
{type:'care_products', amount:0},  
{type:'vegetable_fruits',amount:400}  
],  
{  
customer_id: 1346,  
customer_name:"Gabriel",
```

```

city:"Texas",
purchased:[
  {type:'diary',amount:700},
  {type:'dried_goods',amount:300},
  {type:'snacks',amount:400},
  {type:'care_products', amount:800},
  {type:'vegetable_fruits',amount:0}
] },
customer_id: 1323,
customer_name:"Nicholas",
city:"Texas",
purchased:[
  {type:'diary',amount:300},
  {type:'dried_goods',amount:100},
  {type:'snacks',amount:0},
  {type:'care_products', amount:800},
  {type:'vegetable_fruits',amount:200}
] }
]

```

Using the `customer_purchase` collection, perform the following tasks:

3. Write a query to find the number of customers from Boston city and use the execution statistics command to view the number of documents examined to find the result of the query.
4. Create an index on the `city` field.
5. Create a compound index on the `customer_id` and `customer_name` fields.
6. Create a multikey index on the `purchased` array field.
7. View the indexes created on the `customer_purchase` collection.
8. Drop the index created on the `customer_name` field.
9. Write a query to find the number of customers from Boston city and use the execution statistics command to view the number of documents examined to find the result of the query. Compare the number of documents reviewed before and after indexing.
10. Create a multikey index on the embedded array fields `purchased.type` and `purchased.amount`.
11. Create a text index on the `customer_name` field.
12. Drop the multikey index created on the `purchased.amount` and `purchased.type` fields.
13. Create the wildcard indexes on all the array elements of the `purchased` field.
14. View the indexes created on the `customer_puchase` collection.



SESSION 8

MONGODB REPLICATION AND SHARDING

Learning Objectives

In this session, students will learn to:

- Explain replication
- Describe the ways to implement replication
- Explain sharding
- Describe the ways to implement sharding

When dealing with large data sets ensuring high availability of the database is the most important requirement for any database technology. To ensure that the data requested by multiple global clients is available without downtime, multiple copies of data must be maintained. Replication refers to maintaining multiple copies of a database and allowing the copies to serve the clients in case the original database is unavailable. In addition, a fast-growing database can also be handled by distributing the data and maintaining it across multiple servers. This is referred to as sharding.

Replication helps in maintaining multiple copies of data on several servers to ensure high availability of data. Sharding helps in distributing data across multiple servers to manage high traffic and facilitate easy retrieval of data. This session covers the concept and implementation of replication and sharding in

MongoDB.

8.1 Replication

Replication is creating copies of existing data on different servers. It helps clients to access data without any interruptions, especially during technical glitches. Although data becomes redundant, replication facilitates high availability of data by ensuring that one copy of the data is available for access when the primary data source fails.

8.1.1 Replication Architecture

Let us now understand how MongoDB implements replication. The main data server in which the read and write operations are carried out on the database is known as the primary server. The servers which store copy of the database are called secondary servers. The complete set of a primary along with its secondary servers is called a replica set. A replica set must have a minimum of one primary and two secondary servers.

Figure 8.1 depicts the Primary-Secondary-Secondary (P-S-S) configuration of a replica set in MongoDB where a primary server has two secondary members. Each of the three servers have a separate `mongod` instance. Any write or update operation on the primary server is recorded in the operations log called the oplog. The oplog from the primary server is replicated by the secondary servers and then, the secondary servers update their respective databases from the oplog.

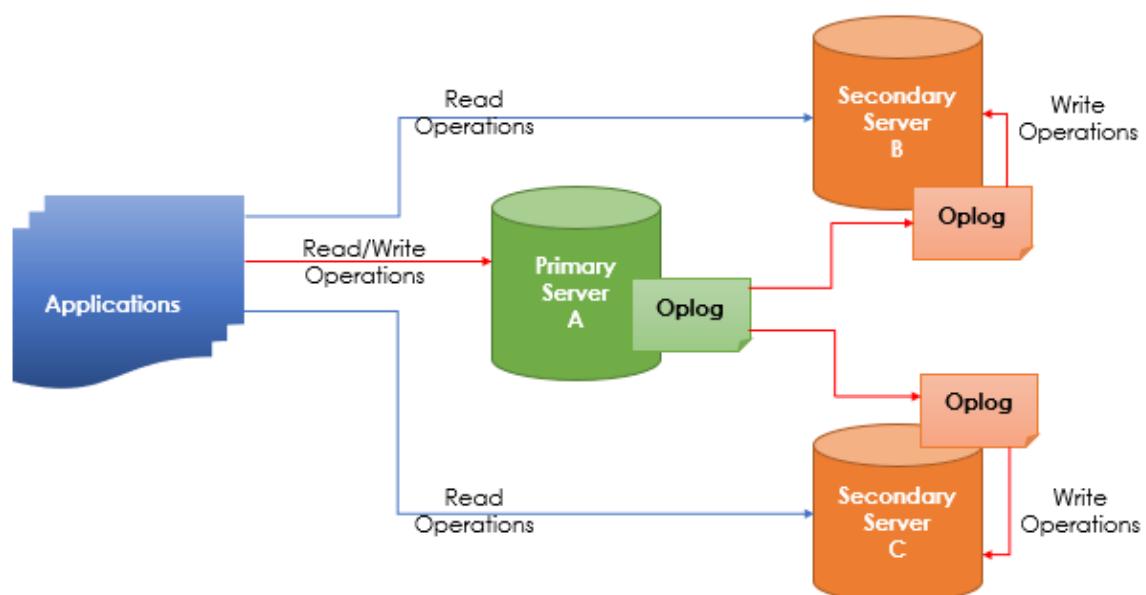


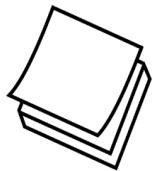
Figure 8.1: Replication Set Operations

In case the primary server fails, an election is held between the secondary servers.

Replica sets can trigger an election if:

- A new server is added.
- A new replica set is added.
- Secondary servers lose connectivity with primary server beyond the configured timeout(10 seconds by default).
- Replica set maintenance is scheduled.

During the election process, secondary servers can serve the read requests if configured. However, the write operation resumes only after the election process is completed. As a result of the election, one of the secondary servers becomes the primary server and will then be capable of handling both read and write operations on the database. Once the original primary server resumes its operation, the recently elected primary server becomes a secondary server once again.



A replica set can include a server that can only vote in election. Such a server is called arbiter and it does not contain copy of the databases. A configuration that has a primary server, a secondary server, and an arbiter is set to follow the Primary-Secondary-Arbitrator (P-S-A) architecture.

8.1.2 Replication Methods

Table 8.1 lists the important methods employed to implement the replication process. Here, `rs` denotes replica set.

Name	Description
<code>rs.add</code>	Adds a new member to an existing replica set
<code>rs.addArb</code>	Adds an arbiter to the replica set
<code>rs.conf</code>	Returns a replica set configuration document
<code>rs.freeze</code>	Prevents the member from seeking the role of primary server in election
<code>rs.initiate</code>	Initiates a new replica set
<code>rs.printReplicationInfo</code>	Prints a formatted report of the replica set status as per the primary server
<code>rs.printSecondaryReplicationInfo</code>	Prints a formatted report of the replica set status as per the secondary server

Name	Description
rs.reconfig	Applies a new configuration object to an existing replica set
rs.reconfigForPSASet	Performs reconfiguration changes on a PSA replica set or on a replica set that is changing to a P-S-A architecture
rs.remove	Removes a member from a replica set
rs.status	Returns the state of the replica set as a document
rs.stepDown	Makes the current primary server to become a secondary server which forces an election
rs.syncFrom	Overrides the default sync target selection and sets the member of the specified target for this server to sync from

Table 8.1: Replication Methods

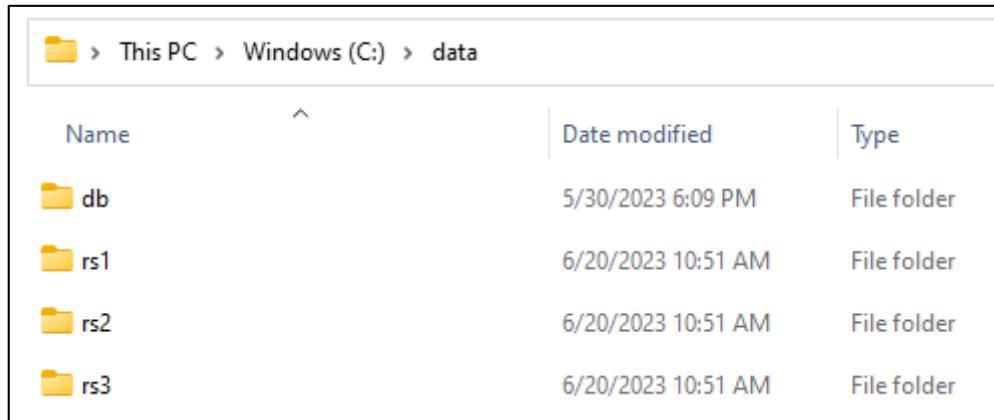
8.1.3 Implementing Replication Using the P-S-S Architecture

Now, let us see how to implement the P-S-S architecture in MongoDB with the help of an example. Consider that:

- The local machine will host all three servers: one primary and two secondaries.
- The `mongod` instance of the primary server will run through port 27017 and will await the requests from the application client.
- Two secondary servers will run through ports 27019 and 27020, respectively.
- Thus, there will be three instances of `mongod` running on the local machine and `mongosh` will be used to communicate with these instances.
- Database, collection, and documents will be created in the primary server and can be read from all the three servers.
- The user must use port 27017 to run queries on the primary server.
- The user must use ports 27019 and 27020 to run queries on the secondary servers.

To implement replication:

1. Create three folders by name `rs1`, `rs2`, and `rs3` in the `c:\data` folder to store the three replica sets as shown in Figure 8.2.



Name	Date modified	Type
db	5/30/2023 6:09 PM	File folder
rs1	6/20/2023 10:51 AM	File folder
rs2	6/20/2023 10:51 AM	File folder
rs3	6/20/2023 10:51 AM	File folder

Figure 8.2: Creation of Replica Set Folders `rs1`, `rs2`, and `rs3`

2. Run `services.msc`.
3. To stop the running instance of MongoDB server. In the **Services** window, right-click MongoDB Server and then click **Stop**, as shown in Figure 8.3.

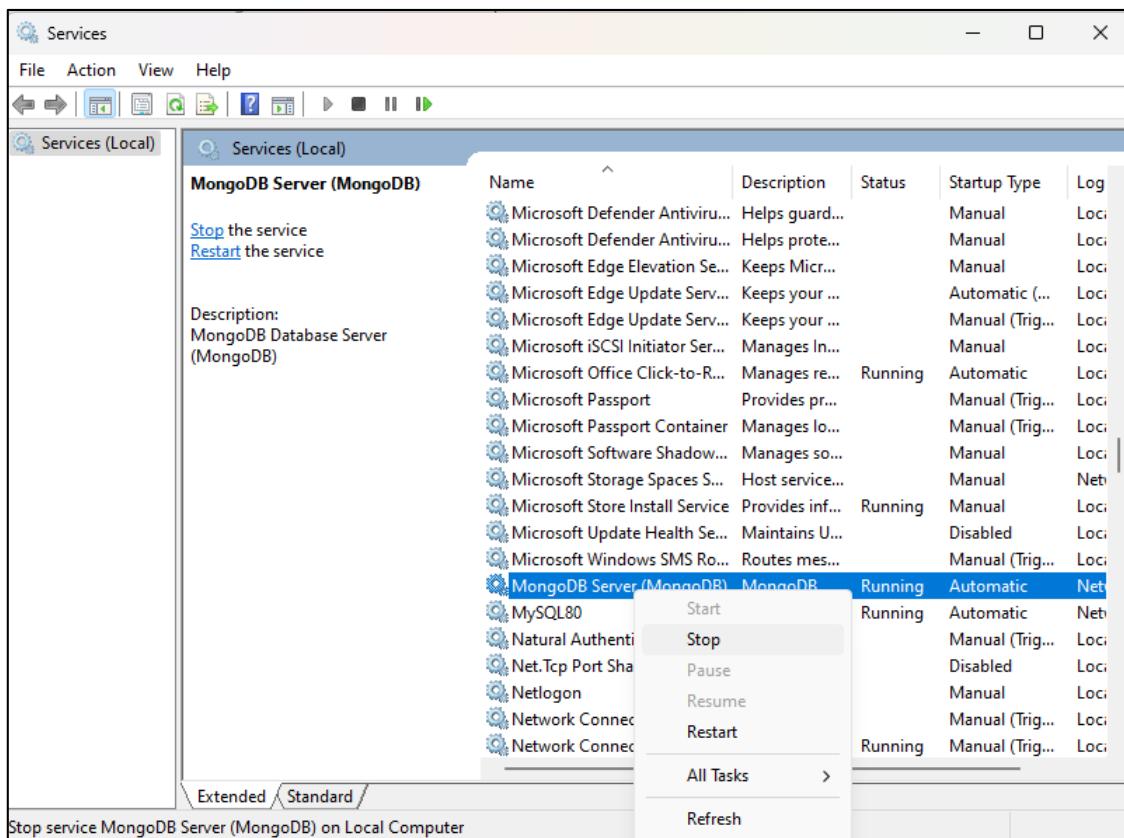
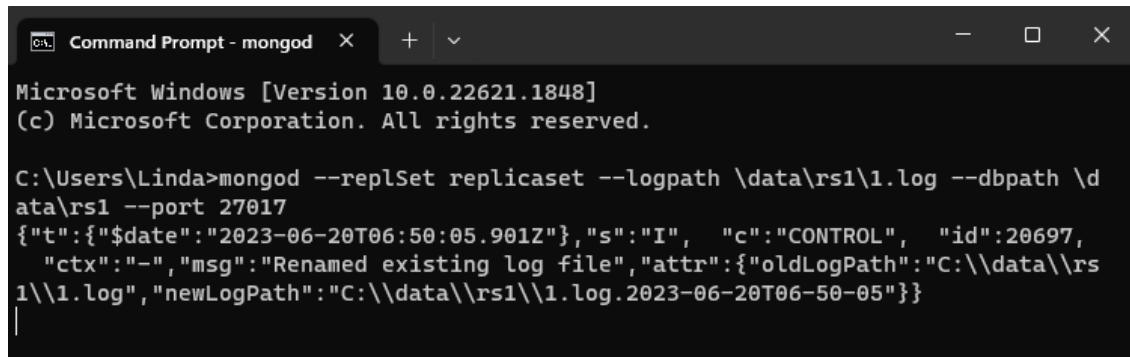


Figure 8.3: Stopping the MongoDB Instance

4. Open the command prompt.
5. Start the `mongod` instance connecting to the port 27017 using the command:

```
mongod --replSet replicaset --logpath  
\\data\\rs1\\1.log --dbpath \\data\\rs1 --port 27017
```

Figure 8.4 shows the output of the command.



The screenshot shows a Windows Command Prompt window. The title bar says "Command Prompt - mongod". The window content shows the following text:
Microsoft Windows [Version 10.0.22621.1848]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Linda>mongod --replSet replicaset --logpath \\data\\rs1\\1.log --dbpath \\data\\rs1 --port 27017
{ "t": {"\$date": "2023-06-20T06:50:05.901Z"}, "s": "I", "c": "CONTROL", "id": 20697,
"ctx": "-", "msg": "Renamed existing log file", "attr": {"oldLogPath": "C:\\\\data\\\\rs1\\\\1.log", "newLogPath": "C:\\\\data\\\\rs1\\\\1.log.2023-06-20T06-50-05"} }
|

Figure 8.4: Primary Server: mongoDb Server Instance on Port 27017

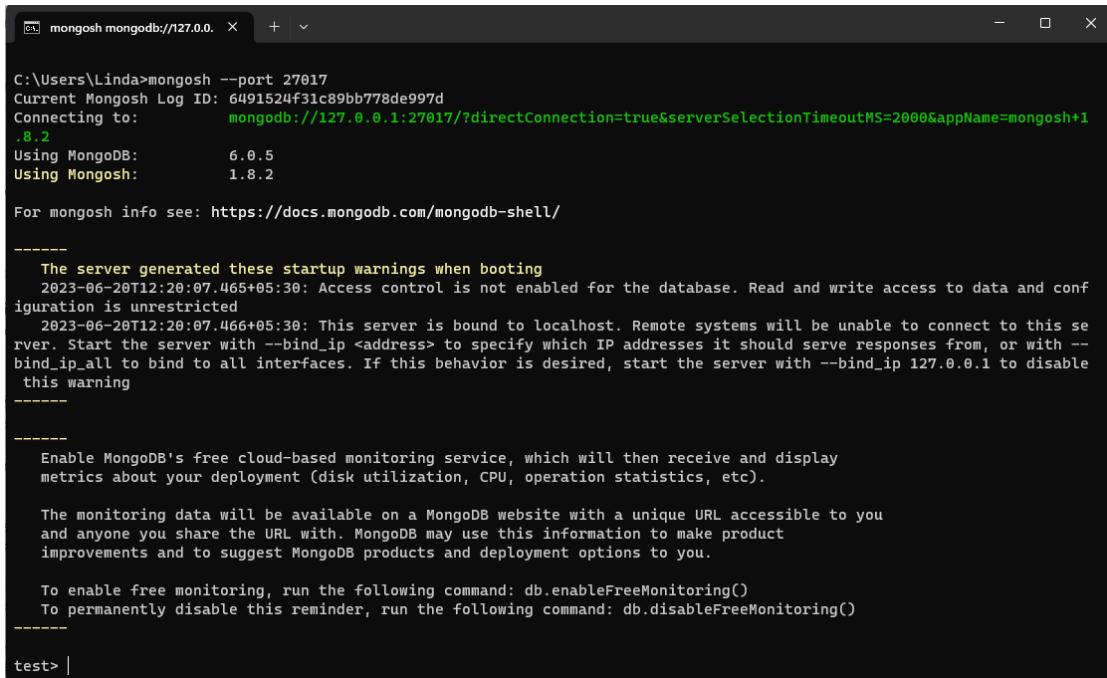


When a user runs the command given in Figure 8.4, the output may be a simple cursor blinking if the command is run for the first time. Figure 8.4 shows the output when the same command is run for the second time.

6. In another command prompt, connect to the `mongod` instance using the `mongosh` command:

```
mongosh --port 27017
```

Figure 8.5 shows the output of the command.



```
C:\Users\Linda>mongosh --port 27017
Current Mongosh Log ID: 6491524f31c89bb778de997d
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1
.8.2
Using MongoDB:    6.0.5
Using Mongosh:    1.8.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-06-20T12:20:07.465+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2023-06-20T12:20:07.466+05:30: This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or with --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to disable this warning
-----

Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()

test> |
```

Figure 8.5: mongoshell Command to Connect to Port 27017

7. Configure the server to listen to port 27017 as primary server using the command as:

```
rsconfig={_id:"replicaset",members:[{_id:0,host:"localhost:27017"}]}
```

Figure 8.6 shows the output of the command.

```
test> rsconfig={_id:"replicaset",members:[{_id:0,host:"localhost:27017"}]}
{ _id: 'replicaset', members: [ { _id: 0, host: 'localhost:27017' } ] }
test> |
```

Figure 8.6: Configuring Port 27017 as Primary Server

8. Include the primary server as part of the replica set using the command:

```
rs.initiate(rsconfig)
```

Figure 8.7 shows the output of the command.

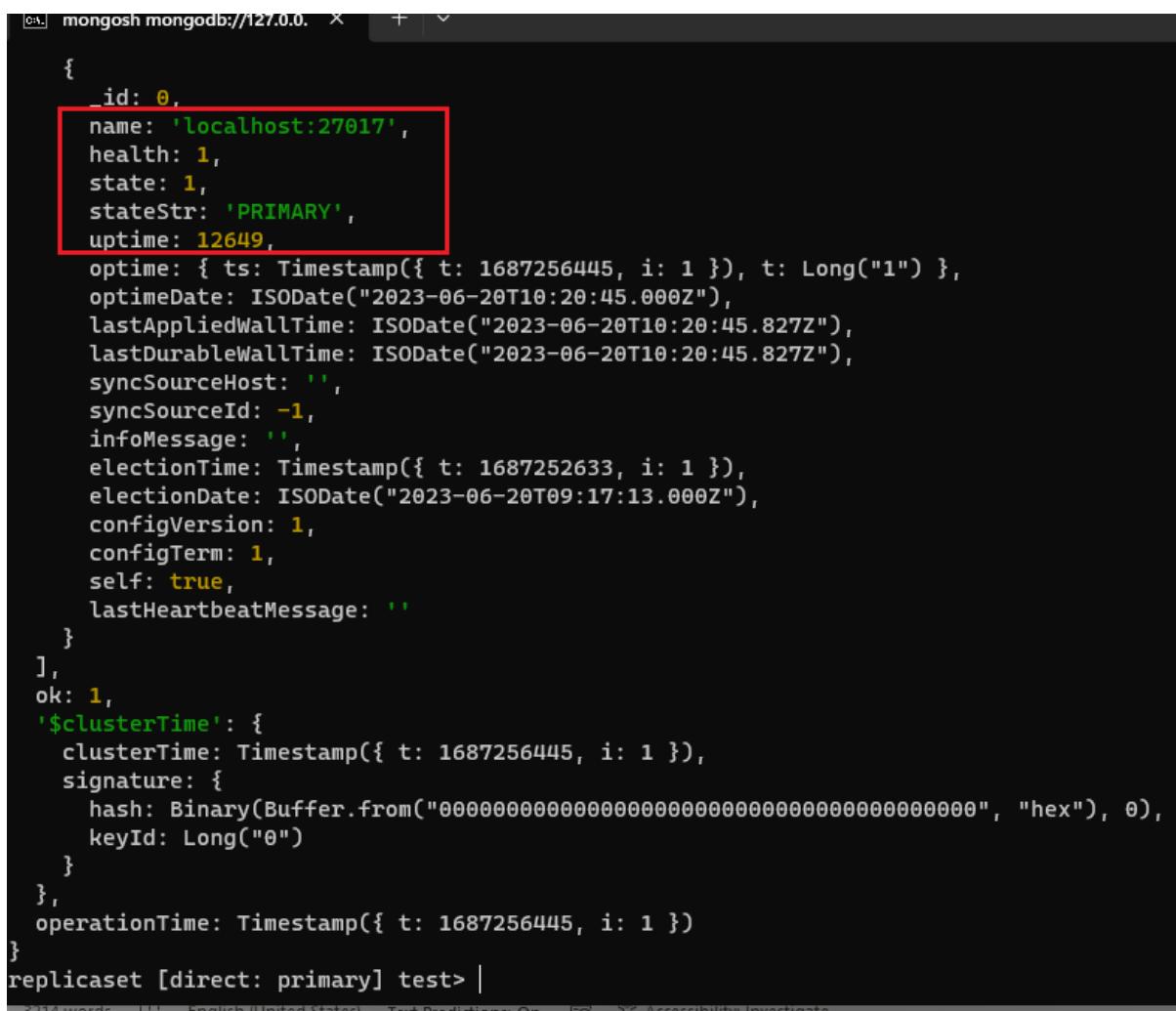
```
test> rs.initiate(rsconfig)
{ ok: 1 }
replicaset [direct: other] test> |
```

Figure 8.7 Making the Primary Server as a Part of Replica Set

9. Confirm whether port 27017 has been configured as primary server using the command:

```
rs.status()
```

Figure 8.8 shows that port 27017 is assigned to the primary server.



```
mongosh mongodb://127.0.0.1:27017/test> rs.status()
{
  "_id": 0,
  "name": "localhost:27017",
  "health": 1,
  "state": 1,
  "stateStr": "PRIMARY",
  "uptime": 12649,
  "optime": { ts: Timestamp({ t: 1687256445, i: 1 }), t: Long("1") },
  "optimeDate": ISODate("2023-06-20T10:20:45.000Z"),
  "lastAppliedWallTime": ISODate("2023-06-20T10:20:45.827Z"),
  "lastDurableWallTime": ISODate("2023-06-20T10:20:45.827Z"),
  "syncSourceHost": "",
  "syncSourceId": -1,
  "infoMessage": "",
  "electionTime": Timestamp({ t: 1687252633, i: 1 }),
  "electionDate": ISODate("2023-06-20T09:17:13.000Z"),
  "configVersion": 1,
  "configTerm": 1,
  "self": true,
  "lastHeartbeatMessage": ""
}
],
"ok": 1,
"$clusterTime": {
  "clusterTime": Timestamp({ t: 1687256445, i: 1 }),
  "signature": {
    "hash": Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
    "keyId": Long("0")
  }
},
"operationTime": Timestamp({ t: 1687256445, i: 1 })
}
replicaset [direct: primary] test> |
```

Figure 8.8: Confirming the Assignment of Port 27017 to Primary Server

Now that the primary server is configured, the next step is to configure the secondary servers.

10. Open a new command prompt.
11. Start the mongod instance connecting to the port 27019 using the command:

```
mongod --replSet replicaset --logpath  
\\data\\rs2\\2.log --dbpath \\data\\rs2 --port 27019
```

Figure 8.9 shows the output of the command.

```
Microsoft Windows [Version 10.0.22621.1848]  
(c) Microsoft Corporation. All rights reserved.  
C:\\Users\\Linda>mongod --replSet replicaset --logpath \\data\\rs2\\2.log --dbpath \\data\\rs2 --port 27019  
|
```

Figure 8.9: Secondary Server 1: mongoDb Server Instance on Port 27019

12. Open a new command prompt.
13. Start the mongod instance connecting to the port 27020 using the command:

```
mongod --replSet replicaset --logpath  
\\data\\rs3\\3.log --dbpath \\data\\rs3 --port 27020
```

Figure 8.10 shows the output of the command.

```
Microsoft Windows [Version 10.0.22621.1848]  
(c) Microsoft Corporation. All rights reserved.  
C:\\Users\\Linda>mongod --replSet replicaset --logpath \\data\\rs3\\3.log --dbpath \\data\\rs3 --port 27020  
|
```

Figure 8.10: Secondary Server 2: mongoDb Server Instance on Port 27020

14. At the mongoshell prompt connected to the primary server, add the port 27019 to the primary server's replica set using the command:

```
rs.add("localhost:27019")
```

Figure 8.11 shows the output of the command.

```
replicaset [direct: primary] test> rs.add("localhost:27019")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687257996, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1687257996, i: 1 })
}
replicaset [direct: primary] test> |
```

Figure 8.11: Adding Port 27019 to the Primary Server Replica Set

15. Similarly, add the port 27020 to the primary server's replica set using the command:

```
rs.add("localhost:27020")
```

Figure 8.12 shows the output of the command.

```
replicaset [direct: primary] test> rs.add("localhost:27020")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687258159, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1687258159, i: 1 })
}
replicaset [direct: primary] test> |
```

Figure 8.12: Adding Port 27020 to the Primary Server Replica Set

16. Confirm the configurations of the three ports using the command:

```
rs.status()
```

Figures 8.13, 8.14, and 8.15 show the status of the replica set and the three ports.

```

replicaset [direct: primary] test> rs.status()
{
  set: 'replicaset',
  date: ISODate("2023-06-20T10:53:06.392Z"),
  myState: 1,
  term: Long("1"),
  syncSourceHost: '',
  syncSourceId: -1,
  heartbeatIntervalMillis: Long("2000"),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
  votingMembersCount: 3,
  writableVotingMembersCount: 3,
  optimes: {
    lastCommittedOpTime: { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },
    lastCommittedWallTime: ISODate("2023-06-20T10:52:57.145Z"),
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },
    appliedOpTime: { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },
    durableOpTime: { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },
    lastAppliedWallTime: ISODate("2023-06-20T10:52:57.145Z"),
    lastDurableWallTime: ISODate("2023-06-20T10:52:57.145Z")
  },
  lastStableRecoveryTimestamp: Timestamp({ t: 1687258327, i: 1 }),
  electionCandidateMetrics: {
    lastElectionReason: 'electionTimeout',
    lastElectionDate: ISODate("2023-06-20T09:17:13.020Z"),
    electionTerm: Long("1"),
    lastCommittedOpTimeAtElection: { ts: Timestamp({ t: 1687252632, i: 1 }), t: Long("-1") },
    lastSeenOpTimeAtElection: { ts: Timestamp({ t: 1687252632, i: 1 }), t: Long("-1") },
    numVotesNeeded: 1,
    priorityAtElection: 1,
    electionTimeoutMillis: Long("10000"),
    newTermStartDate: ISODate("2023-06-20T09:17:13.054Z"),
  }
}

```

Figure 8.13: Status of Replica set

```

members: [
  {
    _id: 0,
    name: 'localhost:27017',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 14581,
    optime: { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },
    optimeDate: ISODate("2023-06-20T10:52:57.000Z"),
    lastAppliedWallTime: ISODate("2023-06-20T10:52:57.145Z"),
    lastDurableWallTime: ISODate("2023-06-20T10:52:57.145Z"),
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    electionTime: Timestamp({ t: 1687252633, i: 1 }),
    electionDate: ISODate("2023-06-20T09:17:13.000Z"),
    configVersion: 5,
    configTerm: 1,
    self: true,
    lastHeartbeatMessage: ''
  },
]

```

Figure 8.14: Status of Port 27017

```
{  
    "_id": 1,  
    "name": "localhost:27019",  
    "health": 1,  
    "state": 2,  
    "stateStr": "SECONDARY",  
    "uptime": 390,  
    "optime": { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },  
    "optimeDurable": { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },  
    "optimeDate": ISODate("2023-06-20T10:52:57.000Z"),  
    "optimeDurableDate": ISODate("2023-06-20T10:52:57.000Z"),  
    "lastAppliedWallTime": ISODate("2023-06-20T10:52:57.145Z"),  
    "lastDurableWallTime": ISODate("2023-06-20T10:52:57.145Z"),  
    "lastHeartbeat": ISODate("2023-06-20T10:53:06.270Z"),  
    "lastHeartbeatRecv": ISODate("2023-06-20T10:53:06.228Z"),  
    "pingMs": Long("0"),  
    "lastHeartbeatMessage": "",  
    "syncSourceHost": "localhost:27017",  
    "syncSourceId": 0  
},  
{  
    "_id": 2,  
    "name": "localhost:27020",  
    "health": 1,  
    "state": 2,  
    "stateStr": "SECONDARY",  
    "uptime": 226,  
    "optime": { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },  
    "optimeDurable": { ts: Timestamp({ t: 1687258377, i: 1 }), t: Long("1") },  
    "optimeDate": ISODate("2023-06-20T10:52:57.000Z")  
}
```

Figure 8.15: Status of Ports 27019 and 27020

Note that in Figure 8.15 the server instance on port 27017 is mentioned as the source host for the secondary server.

17. Check if the primary server is primary using the command:

```
rs.isMaster()
```

Figure 8.16 shows the output of the command. Note the phrase 'isMaster:true'.

```
replicaset [direct: primary] test> rs.isMaster()
{
  topologyVersion: {
    processId: ObjectId("64914c1f3d274b57f20991a7"),
    counter: Long("10")
  },
  hosts: [ 'localhost:27017', 'localhost:27019', 'localhost:27020' ],
  setName: 'replicaset',
  setVersion: 5,
  ismaster: true,
  secondary: false,
  primary: 'localhost:27017',
  me: 'localhost:27017',
  electionId: ObjectId("7fffffff0000000000000001"),
  lastWrite: {
    opTime: { ts: Timestamp({ t: 1687270855, i: 1 }), t: Long("1") },
    lastWriteDate: ISODate("2023-06-20T14:20:55.000Z"),
    majorityOpTime: { ts: Timestamp({ t: 1687270855, i: 1 }), t: Long("1") },
    majorityWriteDate: ISODate("2023-06-20T14:20:55.000Z")
  },
}
```

Figure 8.16: Confirming the Primary Node

18. Open another command prompt.

19. Use mongosh to communicate with the secondary server in port 27019.

```
mongosh --port 27019
```

Figure 8.17 shows the output of the command.

```
C:\Users\Linda>mongosh --port 27019
Current Mongosh Log ID: 6491b084eff9ea55022c71e5
Connecting to:      mongodb://127.0.0.1:27019/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1
.8.2
Using MongoDB:     6.0.5
Using Mongosh:     1.8.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-06-20T16:11:54.301+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2023-06-20T16:11:54.302+05:30: This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or with --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to disable this warning
-----

-----
Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
-----

replicaset: [direct: secondary] test>
```

Figure 8.17: Secondary Server at Port 27019

20. Confirm that the secondary server is not primary using the command:

```
rs.isMaster()
```

Figure 8.18 shows the output of the command. Note the phrase 'isMaster:false'.

```
replicaset [direct: secondary] test> rs.isMaster()
{
  topologyVersion: {
    processId: ObjectId("64918270f2b08fa94704052c"),
    counter: Long("6")
  },
  hosts: [ 'localhost:27017', 'localhost:27019', 'localhost:27020' ],
  setName: 'replicaset',
  setVersion: 5,
  ismaster: false,
  secondary: true,
  primary: 'localhost:27017',
  me: 'localhost:27019',
  lastWrite: {
    opTime: { ts: Timestamp({ t: 1687270995, i: 1 }), t: Long("1") },
    lastWriteDate: ISODate("2023-06-20T14:23:15.000Z"),
    majorityOpTime: { ts: Timestamp({ t: 1687270995, i: 1 }), t: Long("1") },
    majorityWriteDate: ISODate("2023-06-20T14:23:15.000Z")
  },
}
```

Figure 8.18: Confirming the Secondary Node

21. Open another command prompt.

22. Use mongosh to communicate with the secondary server in port 27020.

```
mongosh --port 27020
```

Figure 8.19 shows the output of the command.

```
C:\Users\Linda>mongosh --port 27020
Current MongoDB Log ID: 6491b1ef92312494d07fe149
Connecting to:          mongodb://127.0.0.1:27020/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.2
Using MongoDB:          6.0.5
Using Mongosh:          1.8.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2023-06-20T16:12:24.621+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2023-06-20T16:12:24.621+05:30: This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or with --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to disable this warning
-----

-----
Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
-----

replicaset [direct: secondary] test>
```

Figure 8.19: Secondary Server at Port 27020

Now that the replica set is set up, let us create a database collection and documents on the primary server and read from the secondary server.

Figure 8.20 shows the creation of database, collection, and document on the primary server.

```
replicaset [direct: primary] test> use sample_emp
switched to db sample_emp
replicaset [direct: primary] sample_emp> show collections
replicaset [direct: primary] sample_emp> db.empdetails.insertOne({Name:"David Jackson"})
{
  acknowledged: true,
  insertedId: ObjectId("6492c431ca833b9b775bc3f3")
}
replicaset [direct: primary] sample_emp> show collections
empdetails
replicaset [direct: primary] sample_emp> db.empdetails.find()
[
  { _id: ObjectId("6492c431ca833b9b775bc3f3"), Name: 'David Jackson' }
]
replicaset [direct: primary] sample_emp> |
```

Figure 8.20: Creation of a Collection with a Document in Primary Server

Even though the changes are replicated to the secondary server, the user will not be able to read the details inserted in primary from the secondary. The user can verify this using the series of commands:

```
show dbs
use sample_emp
show collections
db.empdetails.find()
```

Figure 8.21 shows the output of the commands.

```
replicaset [direct: secondary] test> show dbs
admin      80.00 KiB
config     288.00 KiB
local      552.00 KiB
sample_emp  40.00 KiB
replicaset [direct: secondary] test> use sample_emp
switched to db sample_emp
replicaset [direct: secondary] sample_emp> show collections
empdetails
replicaset [direct: secondary] sample_emp> db.empdetails.find()
MongoServerError: not primary and secondaryOk=false - consider using db.getMongo().setReadPref()
or readPreference in the connection string
```

Figure 8.21: Reading a Document from Secondary Server

Note that the `secondaryOk` must be set to true to allow the reading of collection contents from the secondary server. To do this, run the command:

```
rs.secondaryOk()
```

Figure 8.22 shows the output of this command.

```
replicaset [direct: secondary] sample_emp> rs.secondaryOk()
DeprecationWarning: .setSecondaryOk() is deprecated. Use .setReadPref("primaryPreferred") instead
Setting read preference from "primary" to "primaryPreferred"
```

Figure 8.22: Allowing the Secondary Server to Serve Collection Queries

Figure 8.23 shows that the document inserted in the primary server can be read now from the secondary server.

```
replicaset [direct: secondary] sample_emp> db.empdetails.find()
[
  { _id: ObjectId("6492c431ca833b9b775bc3f3"), Name: 'David Jackson' }
]
replicaset [direct: secondary] sample_emp> |
```

Figure 8.23: Reading Documents from the Secondary Server

Secondary servers can become primary servers as a result of an election. To force an election, the user can shut down the primary server using the command:

```
db.shutdownServer()
```

Figure 8.24 shows the result of shutting down the primary server.

```
replicaset [direct: primary] sample_emp> db.shutdownServer()
MongoNetworkError: connection 4 to 127.0.0.1:27017 closed
sample_emp> |
```

Figure 8.24: Shutting Down the Primary Server

Now, one of the secondary servers automatically becomes the primary. To verify this, let us insert another document from the secondary server running on port 27019 using the command:

```
db.empdetails.insertOne({Name:"Elana Thomas"})
```

Figure 8.25 shows the output of this command and the result of running the find command on the empdetails collection.

```
replicaset [direct: secondary] sample_emp> db.empdetails.insertOne({Name:"Elana Thomas"})
{
  acknowledged: true,
  insertedId: ObjectId("6492d0b4e873a7aadec6e22")
}
replicaset [direct: primary] sample_emp> db.empdetails.find()
[
  { _id: ObjectId("6492c431ca833b9b775bc3f3"), Name: 'David Jackson' },
  { _id: ObjectId("6492d0b4e873a7aadec6e22"), Name: 'Elana Thomas' }
]
replicaset [direct: primary] sample_emp> |
```

Figure 8.25: Secondary Server Elected as a Primary Server

Note that the server instance that was secondary has changed to primary and the document has been successfully inserted. Thus, the secondary server has automatically upgraded to be a primary server and has secured the write permissions as the original primary server is unavailable.

To return the elected primary server to its original state as secondary server, the user can use the stepDown command. However, the primary server that was shut down must be started using the command in a new command prompt:

```
mongod --replSet replicaset --logpath
\data\rs1\1.log --dbpath \data\rs1 --port 27017
```

In another new command prompt, the user can start the instance using the command:

```
mongosh --port 27017
```

Now, in the server instance of port 27019, the user can run the stepDown command as:

```
rs.stepDown()
```

Figure 8.26 shows the output of this command.

```
replicaset [direct: primary] sample_emp> rs.stepDown()
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687343939, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1687343939, i: 1 })
}
replicaset [direct: secondary] sample_emp> |
```

Figure 8.26: Secondary Server Steps Down from Primary Server Status

8.2 Sharding

Replication addresses the high availability of data servers by maintaining copies of the databases and ensuring that a secondary server takes over as a primary server when required. However, fast-growing large data sets may soon become challenging to handle for the primary server's CPU. The CPU, RAM, and Input/Output devices may be exhausted because of this. There are two ways of distributing data:

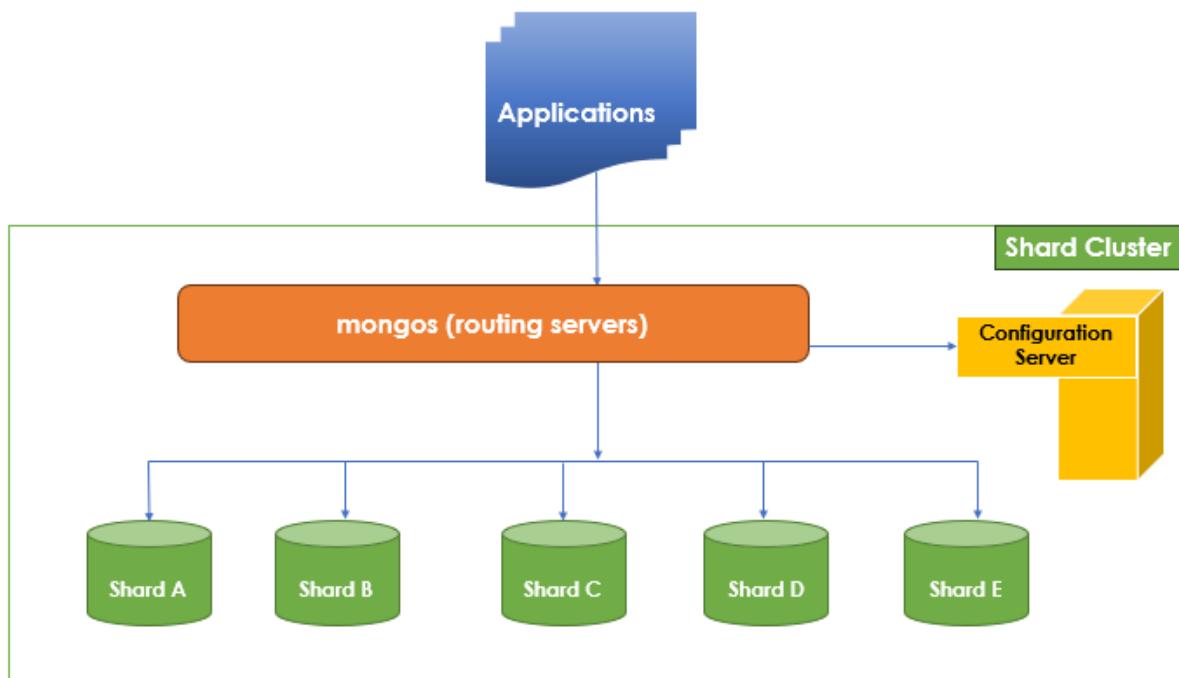
1. **Vertical scaling** involves increasing the RAM capacity, using a high-capacity processor, and increasing the storage space. Limitations in technology may prevent this set up from working efficiently for the given workload. Cloud-based providers have a vertical ceiling which prevents efficient processing of queries.
2. **Horizontal scaling** involves dividing the data across multiple servers. Each server is a high-end machine holding a subset of the large data set. Hence, the workload is divided and handled efficiently. In addition, the overall cost

is lower than investing on a single server. Sharding is the horizontal distribution of data across multiple servers.

When the load on a server increases, MongoDB supports distribution of data across different servers and thus supports horizontal scaling through sharding. In MongoDB, sharding is done at the collection level by distributing collections of data across multiple servers.

8.2.1 Shard Clusters

A shard cluster in MongoDB contains three components:



- **Shard Servers:** Data is distributed across the shard servers. Each shard server has a subset of sharded data and must be deployed as a replica set.
- **mongos:** The MongoDB instances `mongos` act as query routers. They provide an interface between the client application and shard cluster.
- **Configuration Servers:** Configuration information and metadata are stored in config servers. Each configuration server must also be deployed as a replica set.

8.2.2 Sharding Strategies

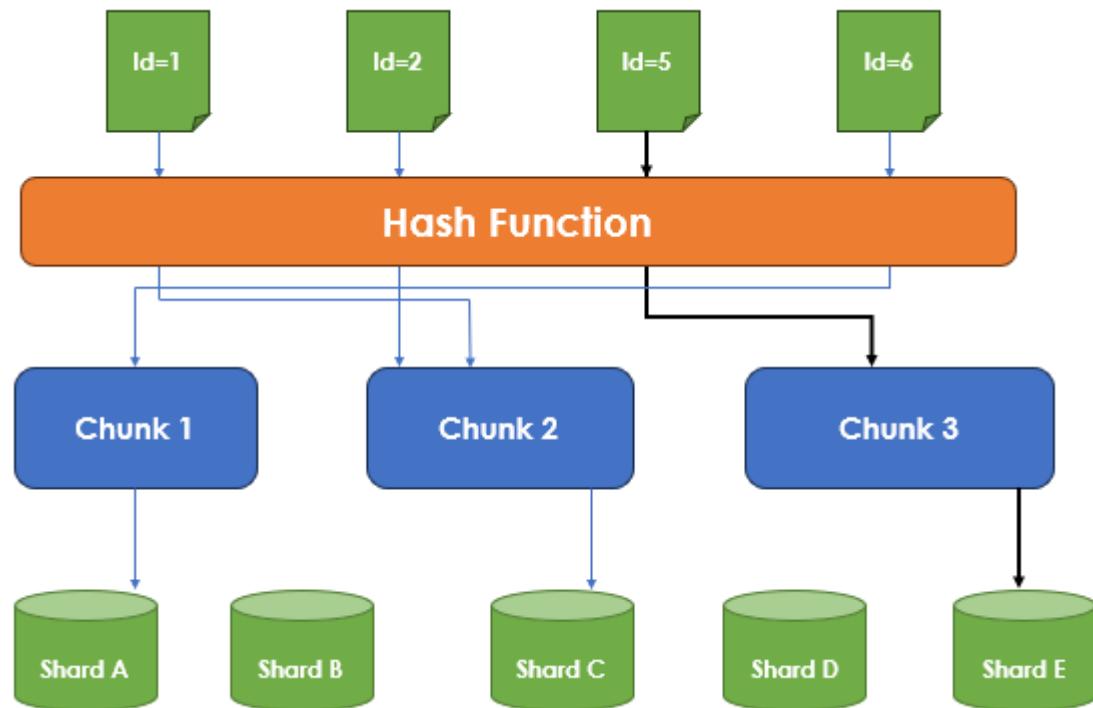
To distribute data across different shards, MongoDB uses shard keys. These shard keys are either single or compound index keys from the document which determine the distribution of data amongst the shard cluster. MongoDB divides the data in shards evenly into chunks. Each chunk has a span of index keys that are assigned to a range of shard key values.

The mapping of index keys to shard keys can be based on any of the two types of sharding strategies supported by MongoDB:

- **Hash-based Sharding**

Hashing refers to the use of a mathematical function that uses the shard key of the document to compute hash values. Each chunk is then assigned a range of shard key fields based on the hash values. A chunk has a default size of 64 MB. When a user queries for data, MongoDB automatically computes the hash value and locates the chunk that contains the required data.

Consider that the user has `Id` as the shard key for a collection.

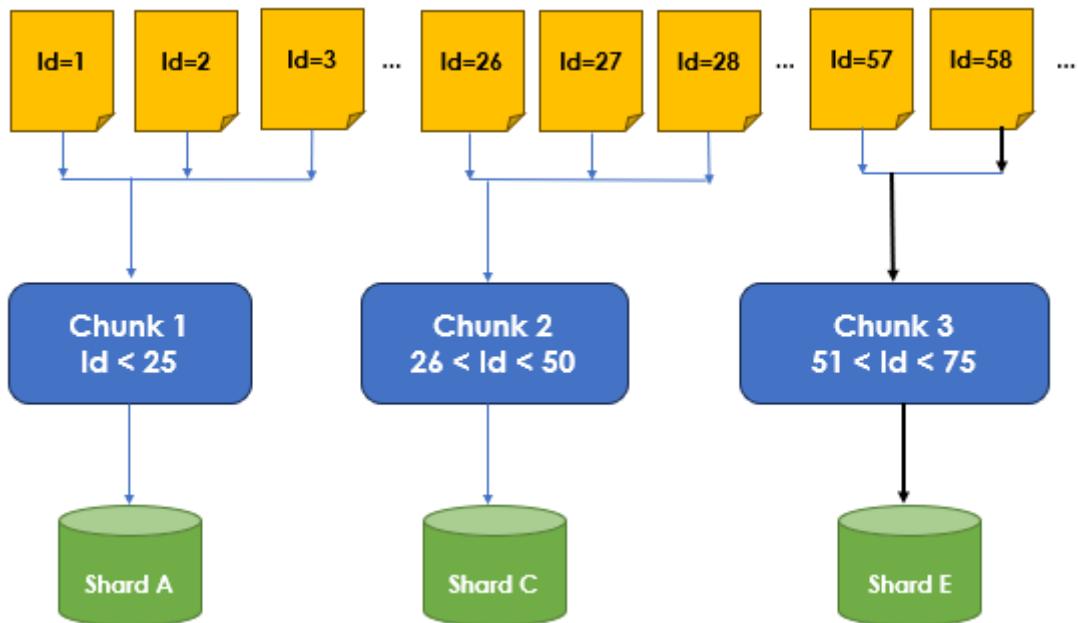


If the user wants to locate a document with `Id` as 5, then MongoDB uses the hash function to locate the chunk in which the document is stored. Here it is Chunk 3 which is in Shard E.

Note that documents with nearer shard key values may not be in the same shard or even in the adjacent shards. For example, document with `Id` as 5 is in Shard E while the document with `Id` as 6 is in Shard A.

- **Range-based Sharding**

In range-based sharding, indexes are arranged in order and continuous ranges are formed. Each index value falls under a range and the range is assigned to shards.



Documents with nearer shard values are likely to be in the same shard.

8.2.3 Sharding Methods

Table 8.2 lists the important methods employed to implement the sharding process. Here, `sh` denotes sharding.

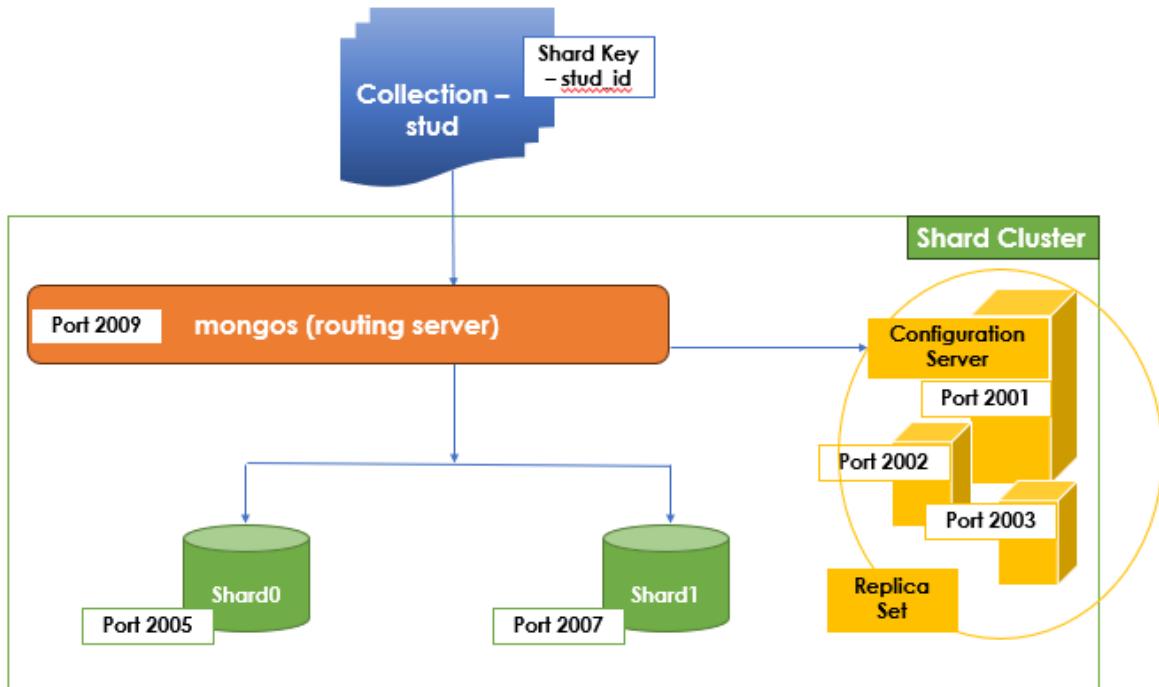
Name	Description
<code>sh.addShard()</code>	Adds a shard to the cluster
<code>sh.status()</code>	Reports on the status of a sharded cluster similar to <code>db.printShardingStatus()</code>
<code>sh.enableSharding()</code>	Creates a database
<code>sh.shardCollection()</code>	Enables sharding for a collection
<code>sh.moveChunk()</code>	Migrates a chunk to a cluster

Table 8.2: Sharding Methods

8.2.4 Implementing Sharding

Let us now see how to implement sharding with the help of an example. Consider that the user wants to store student details in the `stud` collection of the `studentdb` database. The data will be distributed across two shard servers `Shard0` and `Shard1` which will be configured to listen from the ports 2005 and 2007, respectively. A routing server enabled by `mongos` will be configured to listen from the port 2009. The configuration server replica set, primary `cs1`, and replicas, `cs2`, and `cs3`, will be configured to listen from ports 2001, 2002, and 2003, respectively.

Hundred documents which consist of fields `stud_id` and `class` will be routed through the query router from mongo shell. The query will distribute the documents across the two shard servers.



Building the Configuration Server Replica Set

1. Create three directories namely `cs1`, `cs2`, and `cs3` to hold the data for configuration replica sets in the, `c:\data` folder.
2. Open a new command prompt and create a `mongod` instance to act as a primary configuration server using the command:

```
mongod --configsvr --replSet rshard --logpath
\data\cs1\1.log --dbpath \data\cs1 --port 2001
```

Note that the replica set name is `rshard`.

3. Open a new command prompt and start communication with the configuration server, `cs1` through port 2001 using the command:

```
mongosh --port 2001
```

- Configure cs1 as the primary shard configuration server which will be part of a replica set using the commands:

```
shardconfig={_id:"rshard",members:[{_id:0,host:  
"localhost:2001"}]}  
  
rs.initiate(shardconfig)
```

Figure 8.27 shows the output of these commands.

```
test> shardconfig={_id:"rshard",members:[{_id:0,host:"localhost:2001"}]}  
{ _id: 'rshard', members: [ { _id: 0, host: 'localhost:2001' } ] }  
test> rs.initiate(shardconfig)  
{ ok: 1, lastCommittedOpTime: Timestamp({ t: 1687406954, i: 1 }) }  
rshard [direct: other] test> |
```

Figure 8.27: Configuring the Primary Configuration Server

- Configure the secondary configuration servers in the replica set using the commands in two new command windows:

```
mongod --configsvr --replSet rshard --logpath  
\data\cs2\2.log --dbpath \data\cs2 --port 2002  
  
mongod --configsvr --replSet rshard --logpath  
\data\cs3\3.log --dbpath \data\cs3 --port 2003
```

- From the primary configuration server instance, add the secondary servers to the replica set using the commands:

```
rs.add("localhost:2002")  
rs.add("localhost:2003")
```

- Check the status of the configuration servers using the command:

```
rs.status()
```

Figure 8.28 shows the status of the configuration servers.

```
name: 'localhost:2001',
health: 1,
state: 1,
stateStr: 'PRIMARY',
uptime: 2418,
optime: { ts: Timestamp({ t: 1687408977, i: 1 }), t: Long("1") },
optimeDate: ISODate("2023-06-22T04:42:57.000Z"),
lastAppliedWallTime: ISODate("2023-06-22T04:42:57.063Z"),
lastDurableWallTime: ISODate("2023-06-22T04:42:57.063Z"),
syncSourceHost: '',
syncSourceId: -1,
infoMessage: '',
electionTime: Timestamp({ t: 1687406954, i: 2 }),
electionDate: ISODate("2023-06-22T04:09:14.000Z"),
configVersion: 5,
configTerm: 1,
self: true,
lastHeartbeatMessage: ''
},
{
_id: 1,
name: 'localhost:2002',
health: 1,
state: 2,
stateStr: 'SECONDARY',
uptime: 46,
optime: { ts: Timestamp({ t: 1687408975, i: 1 }), t: Long("1") },
optimeDurable: { ts: Timestamp({ t: 1687408975, i: 1 }), t: Long("1") },
optimeDate: ISODate("2023-06-22T04:42:55.000Z"),
optimeDurableDate: ISODate("2023-06-22T04:42:55.000Z"),
```

Figure 8.28: Status of the Configuration Server Replica Sets

Creating the Shard Servers Shard0 and Shard1

8. Create two new folders `Shard0` and `Shard1` in the path, `C:\data` to hold data for sharding.
9. Create two log folders one each inside the folders `Shard0` and `Shard1`.
10. Create a shard server `Shard0` as replica set to listen to port 2005. To do this, open a new command prompt and execute the command:

```
mongod --shardsvr --repSet shardrep --logpath
\data\Shard0\log\Shard0.log --dbpath
\data\Shard0 --port 2005
```

Note that the replica set name for `Shard0` is `shardrep`.

11. Open a new command prompt and start communication with the shard server, Shard0 through port 2005 using the command:

```
mongosh --port 2005
```

12. Configure Shard0 as the primary shard server which will be part of a replica set using the commands:

```
sconfig={_id:"shardrep",members:[{_id:0,host:"localhost:2005"}]}  
rs.initiate(sconfig)
```

Figure 8.29 shows the output of the commands.

```
test> sconfig={_id:"shardrep",members:[{_id:0,host:"localhost:2005"}]}  
{ _id: 'shardrep', members: [ { _id: 0, host: 'localhost:2005' } ] }  
test> rs.initiate(sconfig)  
{ ok: 1 }  
shardrep [direct: other] test> |
```

Figure 8.29: Shard0 Server Configured to Listen from Port 2005

13. Similarly, create a shard server Shard1 as replica set to listen to port 2007 using the commands.

```
mongod --shardsvr --replSet shard0rep --  
logpath \data\Shard1\log\Shard1.log --dbpath  
\data\Shard1 --port 2007  
  
mongosh --port 2007  
  
s2config={_id:"shard0rep",members:[{_id:0,host:"localhost:2007"}]}  
  
rs.initiate(s2config)
```

14. Open a new command prompt and start the routing server to listen to port 2009 using the primary config server using the command:

```
mongos --port 2009 --configdb rshard/localhost:2001
```

Figure 8.30 shows the output of this command.

```
C:\Users\Linda>mongos --port 2009 --configdb rshard/localhost:2001
{"t": {"$date": "2023-06-22T14:21:27.918Z"}, "s": "W", "c": "SHARDING", "id": 24132, "ctx": "-", "msg": "Running a sharded cluster with fewer than 3 config servers should only be done for testing purposes and is not recommended for production."}
{"t": {"$date": "2023-06-22T19:51:27.930+05:30"}, "s": "I", "c": "CONTROL", "id": 23285, "ctx": "-", "msg": "Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}
{"t": {"$date": "2023-06-22T19:51:28.886+05:30"}, "s": "I", "c": "NETWORK", "id": 4915701, "ctx": "thread1", "msg": "Initialized wire specification", "attr": {"spec": {"incomingExternalClient": {"minWireVersion": 0, "maxWireVersion": 17}, "incomingInternalClient": {"minWireVersion": 0, "maxWireVersion": 17}, "outgoing": {"minWireVersion": 17, "maxWireVersion": 17}, "isInternalClient": true}}}
{"t": {"$date": "2023-06-22T19:51:28.895+05:30"}, "s": "I", "c": "NETWORK", "id": 4648602, "ctx": "thread1", "msg": "Implicit TCP FastOpen in use."}
{"t": {"$date": "2023-06-22T19:51:28.897+05:30"}, "s": "I", "c": "HEALTH", "id": 5936503, "ctx": "thread1", "msg": "Fault manager changed state", "attr": {"state": "StartupCheck"}}
{"t": {"$date": "2023-06-22T19:51:28.897+05:30"}, "s": "W", "c": "CONTROL", "id": 22120, "ctx": "thread1", "msg": "Access control is not enabled for the database. Read and write access to data and configuration is unrestricted", "tags": ["startupWarnings"]}
{"t": {"$date": "2023-06-22T19:51:28.898+05:30"}, "s": "W", "c": "CONTROL", "id": 22140, "ctx": "thread1", "msg": "This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or with --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to disable this warning", "tags": ["startupWarnings"]}
{"t": {"$date": "2023-06-22T19:51:28.899+05:30"}, "s": "I", "c": "CONTROL", "id": 23403, "ctx": "mongosMain", "msg": "Build Info", "attr": {"buildInfo": {"version": "6.0.5", "gitVersion": "c9a99c120371d4d4c52cbb15dac34a36ce8d3b1d", "modules": [], "allocator": "tcmalloc", "environment": {"distmod": "windows", "distarch": "x86_64", "target_arch": "x86_64"}}}}
{"t": {"$date": "2023-06-22T19:51:28.900+05:30"}, "s": "I", "c": "CONTROL", "id": 51765, "ctx": "mongosMain", "msg": "Operating System", "attr": {"os": {"name": "Microsoft Windows 10", "version": "10.0 (build 22621)"}}, "tags": ["osInfo"]}}
{"t": {"$date": "2023-06-22T19:51:28.900+05:30"}, "s": "I", "c": "CONTROL", "id": 21951, "ctx": "mongosMain", "msg": "Options set by command line", "attr": {"options": {"net": {"port": 2009}, "sharding": {"configDB": "rshard/localhost:2001"}}, "tags": ["options"]}}
{"t": {"$date": "2023-06-22T19:51:28.905+05:30"}, "s": "I", "c": "NETWORK", "id": 4603701, "ctx": "mongosMain", "msg": "Starting Replica Set Monitor", "attr": {"protocol": "streamable", "uri": "rshard/localhost:2001"}}, "tags": ["rsmon"]}}
{"t": {"$date": "2023-06-22T19:51:28.906+05:30"}, "s": "I", "c": "-", "id": 4333223, "ctx": "mongosMain", "msg": "RSM now active"}
```

Figure 8.30: Configuring the Routing Server

15. Open a new command prompt and establish communication with the routing server using the command:

```
mongosh --port 2009
```

16. Add the shard servers to the routing server using the commands:

```
sh.addShard("shardrep/localhost:2005")
sh.addShard("shard0rep/localhost:2007")
```

Figure 8.31 shows the output of this commands.

```
[direct: mongos] test> sh.addShard("shardrep/localhost:2005")
{
  shardAdded: 'shardrep',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687444154, i: 6 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1687444154, i: 6 })
}
[direct: mongos] test> sh.addShard("shardrep/localhost:2007")
{
  shardAdded: 'shardrep',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687444619, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1687444619, i: 1 })
}
[direct: mongos] test> |
```

Figure 8.31: Adding the Shards to the Router

17. Enable sharding for the studentdb database using the command:

```
sh.enableSharding("studentdb")
```

Figure 8.32 shows the output of the command.

```
[direct: mongos] test> sh.enableSharding("studentdb")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687452540, i: 2 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1687452540, i: 2 })
}
[direct: mongos] test> |
```

Figure 8.32: Enable Sharding for the studentdb Database

18. Enable sharding for the collection `stud` in the database `studentdb` using the command:

```
sh.shardCollection("studentdb.stud", {"stud_id": "hashed"})
```

Figure 8.33 shows the output of the command.

```
[direct: mongos] test> sh.shardCollection("studentdb.stud", {"stud_id": "hashed"})
{
  collectionsharded: 'studentdb.stud',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687452150, i: 34 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1687452150, i: 30 })
}
[direct: mongos] test> |
```

Figure 8.33: Sharding the `stud` Collection with `stud_id` as Hashed Index

19. Insert records into the collection using the command:

```
for (i = 1; i <= 100; i++) { db.stud.insertMany([{
  stud_id: i, class: "Grade-5"
}]);}
```

Figure 8.34 shows the output of the command.

```
[direct: mongos] test> for (i = 1; i <= 100; i++) { db.stud.insertMany([{
  stud_id: i, class: "Grade-5"
}]);}
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("64947c3c2572d3e93d716f17") }
}
[direct: mongos] test> |
```

Figure 8.34: Inserting Records into the Collection

20. Check if the documents are sharded using the command:

```
db.stud.getShardDistribution()
```

Figure 8.35 shows the output of the command.

```
[direct: mongos] studentdb> db.stud.getShardDistribution()
Shard shardrep at shardrep/localhost:2005
{
  data: '2KiB',
  docs: 56,
  chunks: 2,
  'estimated data per chunk': '1KiB',
  'estimated docs per chunk': 28
}
---
Shard shard0rep at shard0rep/localhost:2007
{
  data: '2KiB',
  docs: 44,
  chunks: 2,
  'estimated data per chunk': '1KiB',
  'estimated docs per chunk': 22
}
---
Totals
{
  data: '5KiB',
  docs: 100,
  chunks: 4,
  'Shard shardrep': [ '56 % data', '56 % docs in cluster', '54B avg obj size on shard' ],
  'Shard shard0rep': [ '44 % data', '44 % docs in cluster', '54B avg obj size on shard' ]
}
[direct: mongos] studentdb>
```

Figure 8.35: Sharded Data

Note that a total of 100 documents were inserted using a loop from the router server listening to port 2009. The Shard0 server listening to port 2005 has two chunks and a total of 56 documents and each chunk has 28 documents. The Shard1 server listening to port 2007 has 44 documents and two chunks with each chunk containing 22 documents.

8.3 Summary

- Replication is making the data available in more than one place which enables easy querying and high data availability.
- PSS and PSA are some of the ways to implement replication.
- If the primary server fails, the secondaries hold an election, and the elected member takes the place of the primary server.
- Sharding is distributing data between shard servers and making them available for the queries from the application.
- The query router helps in routing the data amongst the shards.
- Maintaining the shard servers as replica sets helps in ensuring high availability of data.

Test Your Knowledge

1. Which of the following statements are true about the three-member (P-S-S) replica set in MongoDB?
 - a. Only the primary server has all write operations.
 - b. Data from the primary servers are replicated to the secondary servers.
 - c. Any write or update operations on the primary server are recorded in the secondary server's log file.
 - d. If the primary server is down, then randomly any one of the secondary servers becomes the primary server.
2. Which of the following events will trigger an election among the replica sets in MongoDB?
 - a. When a secondary server tries to read the oplog file of the primary server
 - b. When a new server is added to the replica set
 - c. When the primary server fails
 - d. When the secondary servers lose connectivity with the primary server beyond the configured timeout
3. Consider that you have started the `mongod` instance for the primary server with the replica set name as `primereplica` which is listening to port number 27018. Which of the following option is used to configure this primary server?
 - a. `rsconfig={_id:"primereplica",primary:[{_id:0,port:"27018"}]}`
 - b. `rsconfig={_id:"primereplica",members:[{_id:0,port:"27018"}]}`
 - c. `rsconfig={_id:"primereplica",members:[{_id:0,host:"port:27018"}]}`
 - d. `rsconfig={_id:"primereplica",members:[{_id:0,host:"localhost:27018"}]}`
4. Which of the following statement is not true about Sharding in MongoDB?
 - a. It is a method of distributing a single dataset across multiple servers.
 - b. It implements horizontal scaling of data.
 - c. It implements vertical scaling of data.
 - d. `mongos` provides an interface between the client application and the shard cluster.

5. Which of the following method will print the data distribution statistics for a sharded collection?
- a. db.collection.getStatisticsDistribution()
 - b. db.collection.getShardDistribution()
 - c. db.collection.shardDistribution()
 - d. db.collection.statisticsDistribution()

Answers to Test Your Knowledge

1	a, b
2	b, c, d
3	d
4	c
5	b

Try it Yourself

1. Implement a three-member replica set with the P-S-S architecture by performing the following tasks:
 - a. Start the `mongod` instance for the primary server with port number 27010 and replica set name as `studreplica`.
 - b. Start the `mongod` instance for the two secondary servers with port numbers 27011, and 27012. Name the replica set as `studreplica`.
 - c. Configure the primary server and add the secondary servers to the replica set.
 - d. In the primary server, create a database with the name `Student_detail` that contains a collection named `Stud_mark`.
 - e. Insert the following four documents into the `Stud_mark` collection.

```
[  
    { name: "Adam",  
        gender:"M",  
        subjects:["Java", "C", "Python"],  
        marks:[89, 78, 90],  
        average:85.6  
    },  
    { name: "Franklin",  
        gender:"M",  
        subjects:["C", "VB", "Python"],  
        marks:[78, 85, 89],  
        average:84  
    },  
    { name: "Michael",  
        gender:"M",  
        subjects:["Java", "PHP"],  
        marks:[88, 89],  
        average:88.5  
    },  
    { name: "Amelia",  
        gender:"F",  
        subjects:["Ruby", "C++"],  
        marks:[86, 87],  
        average: 86.5  
    }  
]
```

- f. Check whether the `student_detail` database is replicated in the secondary servers which are listening to ports 27011 and 27012.
 - g. Force the primary server to shut down and check whether one of the secondary servers becomes the primary server.
 - h. Restore the role of the primary server to the server to listen to port 27010.
2. Create a sharded cluster with the following components:
 - a. Shard: Create two shard servers `shard1` and `shard2` where each shard server should be deployed as a replica set. Name the replica set name of the shard server as `empshardreplica`. The `shard1` and `shard2` servers will listen to ports 27013 and 27014, respectively.
 - b. Config servers: Create a configuration server with P-S-S architecture and name the replica set as `empconreplica`. The primary node of the config server group listens to port 27006 and the two secondary servers listen to ports 27007 and 27009.
 - c. `mongos`: Create a query routing server to listen to port 27005.
3. Implement the sharding concept by performing the following tasks:
 - a. Start the `mongos` routing server which listens to port 27010 using the configuration server 27006.
 - b. Add the shard servers `shard1` and `shard2` to the routing server.
 - c. Enable sharding for the `emp` database and `emp_detail` collection. The `emp_detail` has two fields, `emp_id` and `designation` where `emp_id` is used as the hashed key.
 - d. Insert 200 documents into the `emp_detail` collection using for loop. The `emp_id` value ranges from 1 to 200 and assign the designation as `software_engineer` for all the documents.
 - e. Check for the shard distribution of data in the shard servers `shard1` and `shard2`.



SESSION 9

TRANSACTION MANAGEMENT

Learning Objectives

In this session, students will learn to:

- Explain transactions in MongoDB
- Describe the transaction Application Programming Interfaces (API) in MongoDB
- Explain various error labels in MongoDB
- Describe properties of transactions in MongoDB
- Explain sessions and transactions within the sessions in MongoDB

Consider that a buyer wants to place an order on an e-commerce Website by making a payment using a credit card. The steps in this case will be:

- Calculation of the total amount to be paid for the items in the shopping cart
- Selection of mode of payment and entry of the details of the card by the buyer
- Initiation of payment request to the payment gateway

Only after the payment gateway approves the payment, the order is placed. Otherwise, the buyer is taken back to the shopping cart. For this process to work, all these steps should be performed as a single unit. Otherwise, the order will be placed even if the payment is declined, or the payment will be made,

and the order will not be placed. A set of operations that are performed as a single unit is known as a transaction.

This session will provide an overview of the transactions in MongoDB. It will also explain various transaction error labels in MongoDB. This session will explore the properties of transactions in MongoDB. It will also explain the sessions and transactions within the sessions in MongoDB.

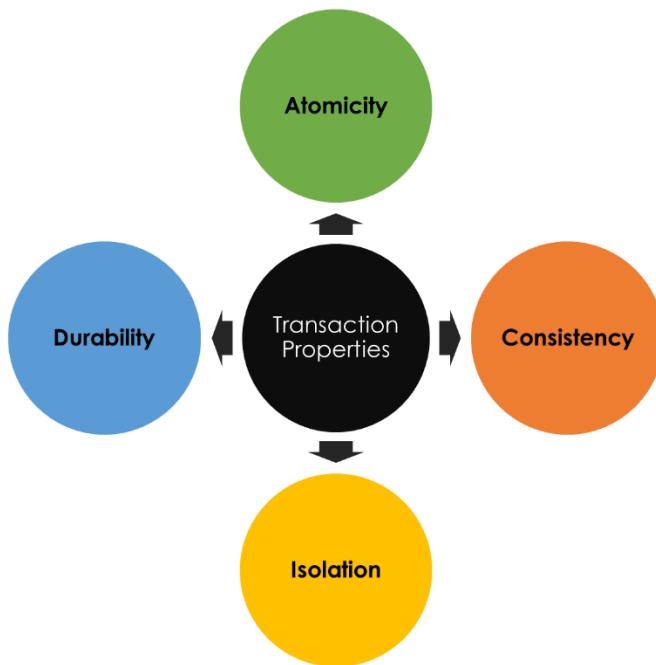
9.1 Introduction to MongoDB Transactions

Transaction is a set of database operations that are run against a database as a single unit of work. In a transaction either all the specified operations in the unit of work execute successfully or it does not execute at all. In case even one database operation in the set fails to execute successfully, all the changes made by other successful database operations are rolled back. The database then comes back to the state in which it was before the transaction was performed.

Transactions in MongoDB can involve operations on a single document or multiple documents.

9.1.1 Transaction Properties

Database transactions guarantee data integrity by complying with the four important properties of transactions:



These properties together are known as ACID properties.

Atomicity ensures that all operations specified in a transaction work as a single unit of work. All the operations in the transaction are executed and changes are made permanent. Otherwise, all the operations are not executed and the database returns to its original state prior to the transaction.

For example, when transferring money from Account A to Account B, if the amount is deducted from Account A and an error occurs when adding that amount to Account B. Atomicity ensures that the amount deducted from Account A is credited back into the account and the transaction fails.

Consistency ensures that the updates made by a transaction are consistent with the existing constraints in the database.

For example, consider an account that has a constraint that on a single day, only a maximum of \$5,000 can be withdrawn. If a transaction was initiated to withdraw \$10,000 from that account, then the transaction will fail because it is not consistent with the existing constraints.

Isolation ensures that in case of multiple transactions which concurrently update data in the same database, the updates from one transaction does not affect updates from the other.

For example, consider that a transaction has two operations—one operation adds \$300 to Account A and the other withdraws \$200 from Account A. Isolation prevents other transactions from accessing the details of Account A between these two operations.

Durability ensures that when a transaction is committed successfully, all the updates made by the transaction are saved permanently. Power outages or hardware failures will not impact the validity of the transaction.

For example, if the server of a bank crashes, the details of all the committed transactions will remain unaffected and can be recovered completely.

A transaction ends with a commit or a rollback. When a transaction is committed, all the changes made by the operations in the transaction are saved permanently and are visible outside the transaction. When a transaction is rolled back, all the changes made by the operations in the transaction are discarded. The database is then restored to the state in which it was before the beginning of the transaction.

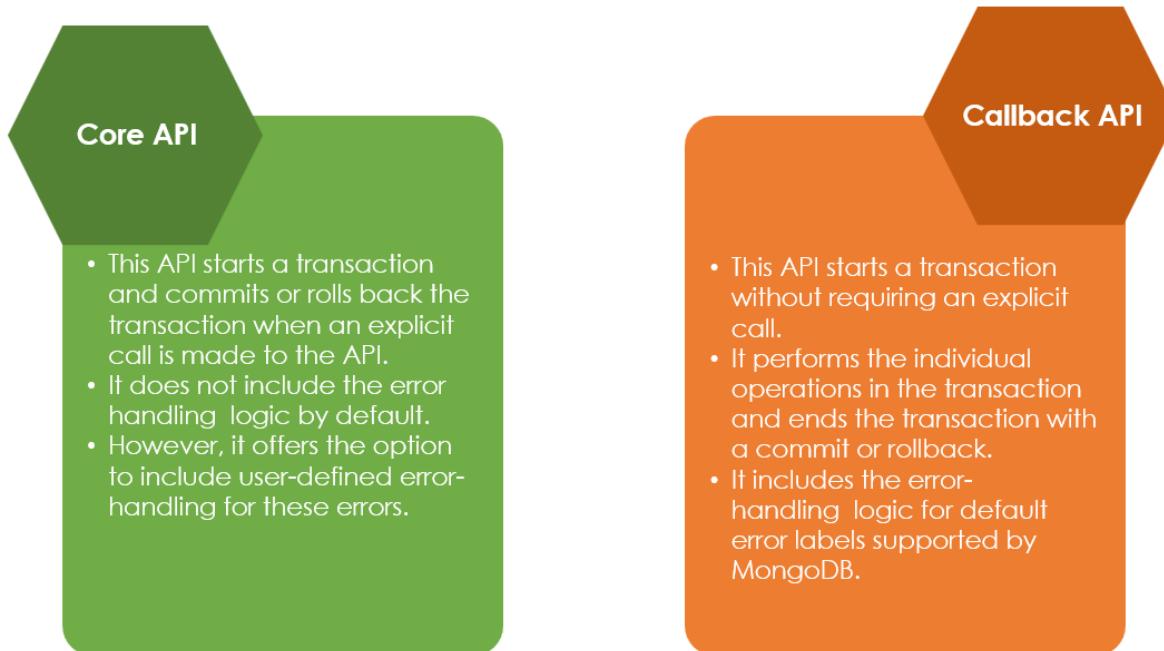
9.1.2 Replica Sets

MongoDB uses the concept of replica sets to support multi-document transactions. A replica set is a collection of `mongod` instances that provide data redundancy and availability by maintaining identical datasets.

The multiple copies of data in a replica set ensures that a multi-document transaction is committed or rolled back completely.

9.1.3 Transaction Application Programming Interface (API)

The two transaction APIs provided by MongoDB are:

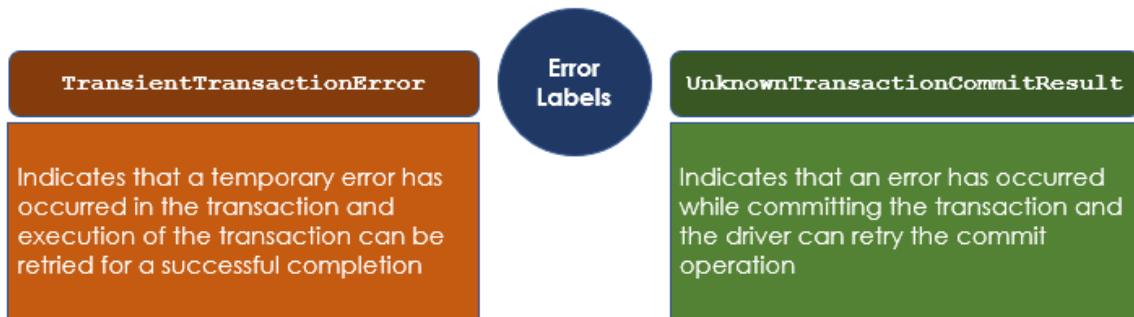


9.1.4 Transaction Error Handling

Consider a scenario where a user is using a credit card to make a payment for a purchase on a shopping Website. In this case, the payment will be successful after the user provides the correct card number, expiry date, Card Verification Value (CVV) number, and credentials. If any of these details are incorrect or if the specified card has expired the transaction is aborted with an error message stating that the payment could not be completed. This happens because these errors are captured in the transaction code and appropriate actions to be taken for each of these errors are also specified with the transaction code. For example, in this case, the transaction code would have specified if any of this incorrect information is provided, terminate the transaction, and display an error message.

Capturing the errors and specifying the actions to be taken in case of an error is referred to as error handling. Error handling is crucial in a transaction to ensure the consistency and integrity of data in the database. For example, in the scenario discussed earlier, consider that the card number is incorrect, but the card number exists for some other user. Then, the payment will be charged to the person who is holding that card instead of the person who made the payment.

The two major error labels in MongoDB are:



A temporary error can be temporary network error or conflict between two sessions that try to modify the same document. When the TransientTransactionError error is encountered, the entire transaction must be retried; individual write operations cannot be retried. When the UnknownTransactionCommitResult error is encountered, the commit operation can be retried; individual write operations in the transaction cannot be retried.

9.2 Transactions and Sessions

MongoDB supports the feature of sessions. A session is a logical grouping of related read and write operations that must be executed sequentially. Transactions exist and get executed within a session. A session can contain multiple transactions. These transactions within a session get executed sequentially. At a given time, only one transaction remains active in a session. If a session ends before a transaction completes, the transaction is aborted, and all changes are rolled back.

Session provides methods for the transactions to access the database. Let us learn about some of the important methods provided by MongoDB to create, commit, and abort transactions in a session.

9.2.1 Start Transaction

MongoDB provides the `Session.startTransaction` method to start a transaction within a session. The syntax for this method is:

```
Session.startTransaction(options)
```

This method takes an argument named `options` which is of type document. Table 9.1 describes the parameters of `options`.

Parameter	Description
<code>readConcern</code>	<ul style="list-style-type: none"> Specifies how the data should be read during a read operation Takes three values: <ul style="list-style-type: none"> <code>snapshot</code>: Reads the data that has been duplicated to majority of the members in the replica set at a specific point in time recently. <code>local</code>: Reads the most recent data that is available when the query is executed. It does not check if the data has been persisted and duplicated to the members of the replica set. There is a possibility that after the data is read, the changes are rolled back. This is the default setting that is applicable when no <code>readConcern</code> option is specified. <code>majority</code>: Reads the most recent data that is persisted and duplicated to the majority of the members in the replica set. This data will not be rolled back. Overrides the read concern levels specified in the operations
<code>writeConcern</code>	<ul style="list-style-type: none"> Specifies how MongoDB must acknowledge the success of write operation Takes two values: <ul style="list-style-type: none"> <code>majority</code>: Checks for acknowledgement from a majority of the members in the replica set <code><number></code>: Checks for acknowledgement from the specified number of members in the replica set

Table 9.1: Parameters of option

9.2.2 Commit Transaction

MongoDB provides the `Session.commitTransaction` method to permanently save the changes made by the operations in a transaction and to make the changes visible outside the transaction. The value for `writeConcern` specified in the method to start the transaction is used here.

The `Session.commitTransaction` transaction ensures atomicity of the transaction.

9.2.3 Abort Transaction

MongoDB provides the `Session.abortTransaction` method to rollback the changes made by the operations in a transaction. This method ensures that the changes made by the operations in the transaction are discarded and are not visible outside the transaction.

9.3 Working with Transactions

MongoDB supports transactions only on replica set or sharded cluster. It does not support transactions on standalone deployments.

Therefore, to work with transactions consider that the user has created a three-member replica set (Primary-Secondary-Secondary) named `replicaset`.

Now, consider that in `replicaset`, the user has the `Employee` database, and this database has the `EmpDetail` collection. The documents in this collection are shown in Figure 9.1.

```
replicaset [direct: primary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  }
]
```

Figure 9.1: Documents in `EmpDetail` Collection

To avoid duplicates in the `Emp_ID` field, the user has created a unique index using the command:

```
db.EmpDetail.createIndex( { "Emp_ID": 1 },
                           { "unique": true } )
```

To allow read operation on the secondary nodes, the user runs the command as:

```
rs.secondaryOk()
```

The command executes as shown in Figure 9.2.

```
replicaset [direct: secondary] test> rs.secondaryOk()
Leaving read preference unchanged (is already "primaryPreferred")
```

Figure 9.2: Allow Read Operation on Secondary

To verify that the documents reflect in the secondary node connected at port 27019, on the secondary node, run the commands as:

```
use Employee
```

```
db.EmpDetail.find()
```

The commands execute as shown in Figure 9.3.

```
replicaset [direct: secondary] Employee> use test
switched to db test
replicaset [direct: secondary] test> rs.secondaryOk()
Leaving read preference unchanged (is already "primaryPreferred")

replicaset [direct: secondary] test> use Employee
switched to db Employee
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  }
]
```

Figure 9.3: Documents in `EmpDetail` Collection on Secondary Node

Similarly, the user verifies that these documents are visible on the secondary node connected at port 27020.

9.3.1 Create a Transaction Within a Session and Commit the Transaction

Consider that the user wants to start a session in MongoDB. In this session, the user wants to execute a transaction to perform a single insert operation. Finally, the user wants to commit the transaction. To perform these operations:

1. To start a session named `session`, on the primary node, run the command as:

```
var session = db.getMongo().startSession()
```

The session is created.

2. To start a transaction, run the command as:

```
session.startTransaction({ "readConcern": { "level": "snapshot" }, "writeConcern": { "w": "majority" } })
```

The transaction starts.

3. To create a variable that represents the `EmpDetail` collection within the transaction, run the command as:

```
var empdetail =
session.getDatabase('Employee').getCollection('EmpDetail')
```

4. To insert a new document into `EmpDetail`, run the command as:

```
empdetail.insertOne({ Emp_ID:102, Emp_Name:"John Edward",
Gender:"Male", Designation:"Software Engineer"})
```

The command executes as shown in Figure 9.4.

```
replicaset [direct: primary] Employee> empdetail.insertOne({ Emp_ID:102,
Emp_Name:"John Edward", Gender:"Male", Designation:"Software Engineer" })
{
  acknowledged: true,
  insertedId: ObjectId("64782d25e5414053d2f443a0")
}
```

Figure 9.4: Document Inserted in `EmpDetail` Collection

- To view the newly inserted document in the primary node, on the primary node, run the command as:

```
empdetail.find()
```

The command executes as shown in Figure 9.5.

```
replicaset [direct: primary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.5: Documents on the Primary Node

The newly inserted document with value in the `Emp_ID` field 102 is displayed.

- To check if the newly inserted document is visible on the secondary node, on the secondary node, run the command as:

```
db.EmpDetail.find()
```

The command executes as shown in Figure 9.6.

```
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  }
]
```

Figure 9.6: Documents on the Secondary Node

The newly inserted document with value in the `Emp_ID` field 102 is not displayed. The collection in the secondary node is not yet updated with the newly inserted document because the transaction has not been committed.

7. To commit the transaction, on the primary node, run the command as:

```
session.commitTransaction()
```

The command executes as shown in Figure 9.7.

```
replicaset [direct: primary] Employee> session.commitTransaction()
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1685600175, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: 0
    }
  },
  operationTime: Timestamp({ t: 1685600175, i: 1 })
}
```

Figure 9.7: Committing the Transaction

8. Now to check if the newly inserted document is visible in the secondary node, repeat Step 6.

The command executes as shown in Figure 9.8.

```
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.8: Document on the Secondary Node After Commit

After the commit operation, the newly inserted document is visible in the secondary node.

9.3.2 Create a Transaction Within a Session and Abort the Transaction

Consider that the user wants to execute a few operations—update and delete—in a session. To do so, the user must perform these steps:

1. To start a session, a transaction, and create a variable that represents the EmpDetail collection, perform steps as learned earlier.
2. To update the Designation field of the employee with Emp_ID as 105 to Project Lead, on the primary node, run the command as:

```
empdetail.updateOne({Emp_ID:105}, {$set:{Designation:  
"Project Lead"})
```

The command executes as shown in Figure 9.9.

```
replicaset [direct: secondary] test> var session = db.getMongo().startSession()  
replicaset [direct: primary] test> session.startTransaction({"readConcern": { "level": "snapshot" }, "writeConcern":  
{ "w": "majority" }})  
replicaset [direct: primary] test> var empdetail = session.getDatabase('Employee').getCollection('EmpDetail')  
replicaset [direct: primary] test> empdetail.updateOne({Emp_ID:105}, {$set:{Designation:"Project Lead"})  
{  
 acknowledged: true,  
 insertedId: null,  
 matchedCount: 1,  
 modifiedCount: 1,  
 upsertedCount: 0  
}
```

Figure 9.9: Update a Document

3. To delete the document where Emp_ID is 103, on the primary node, run the command as:

```
empdetail.deleteOne({Emp_ID:103})
```

The command executes as shown in Figure 9.10.

```
replicaset [direct: primary] test> empdetail.deleteOne({Emp_ID:103})  
{ acknowledged: true, deletedCount: 1 }
```

Figure 9.10: Delete a Document

4. To check if the update and delete operations reflect on the primary node and the secondary node, on both the nodes, run the command as:

```
db.EmpDetail.find()
```

The command executes as shown in Figures 9.11 and 9.12.

```
replicaset [direct: primary] test> empdetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Project Lead'
  },
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.11: Documents on the Primary Node

```

replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]

```

Figure 9.12: Documents on the Secondary Node

Results of update and delete operations are seen on the primary node. However, the results of these operations are not reflected on the secondary node because the transaction is not yet committed.

Consider that the user has realized that document with `Emp_ID` as 102 must be deleted instead of document with `Emp_ID` as 103. Therefore, an abort action must be performed to cancel the delete operation. Since the transaction consists of two operations—update and delete, aborting the transaction will roll back both these operations.

To abort the transaction:

1. On the primary node, run the command as:

```
session.abortTransaction()
```

2. To verify if the changes are rolled back on the primary node, on the primary node, run the command as:

```
empdetail.find()
```

The command executes as shown in Figure 9.13.

```
replicaset [direct: primary] test> empdetail.find()
[
  {
    _id: ObjectId("647778437e0714d84f821a48"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("647778437e0714d84f821a49"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Accountant'
  },
  {
    _id: ObjectId("647778437e0714d84f821a47"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("647834aee5414053d2f443a1"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.13: Rolled Back Changes in the Collection

The changes made by the update and delete operations are rolled back.



By default, MongoDB aborts transactions that run for more than 60 seconds.

9.3.3 Create a Multi-Document Transaction Within a Session

Consider that the user has created a collection `DesignationDetail` with documents as shown in Figure 9.14.

```
[  
  {  
    _id: ObjectId("648ff438c48f7416b9f41ba0"),  
    Designation_ID: 10003,  
    Designation_Title: 'Accountant',  
    Department: 'Finance'  
  },  
  {  
    _id: ObjectId("648ff438c48f7416b9f41b9e"),  
    Designation_ID: 10002,  
    Designation_Title: 'HR Manager',  
    Department: 'Human Resources'  
  },  
  {  
    _id: ObjectId("648ff438c48f7416b9f41b9c"),  
    Designation_ID: 10001,  
    Designation_Title: 'Software Engineer',  
    Department: 'Product Development'  
  },  
  {  
    _id: ObjectId("648ff438c48f7416b9f41b9f"),  
    Designation_ID: 10012,  
    Designation_Title: 'HR Executive',  
    Department: 'Human Resources'  
  },  
  {  
    _id: ObjectId("648ff438c48f7416b9f41b9d"),  
    Designation_ID: 10011,  
    Designation_Title: 'Project Manager',  
    Department: 'Product Development'  
  },  
  {  
    _id: ObjectId("648ff438c48f7416b9f41ba1"),  
    Designation_ID: 10013,  
    Designation_Title: 'Finance Manager',  
    Department: 'Finance'  
  }  
]
```

Figure 9.14: Documents in the `DesignationDetail` Collection

Now, the user wants to change the designation of Accountant to Finance Executive. This change must be made in both the EmpDetail as well as the DesignationDetail collections. To do this, the user can employ a multi-document transaction.

To do so, the user must perform the following steps:

1. To start a session, a transaction, and create a variable that represents the EmpDetail collection, perform steps as learned earlier.
2. To create a variable that represents the DesignationDetail collection within the transaction, run the command as:

```
var designation =  
session.getDatabase('Employee').getCollection('Designation  
Detail')
```

3. To update the Designation field of the employee with Emp_ID as 105 to Finance Executive, on the primary node, run the command as:

```
empdetail.updateOne({Emp_ID:105}, {$set:{Designation:  
"Finance Executive"}})
```

4. To update the Designation_Title field of the designation with Designation_ID as 10003 to Finance Executive, on the primary node, run the command as:

```
designation.updateOne({Designation_ID:10003}, {$set:{  
Designation_Title:"Finance Executive"}})
```

5. To commit the transaction, on the primary node, run the command as:

```
session.commitTransaction()
```

The command executes as shown in Figure 9.15.

```
replicaset [direct: primary] Employee> var session = db.getMongo().startSession()
replicaset [direct: primary] Employee> session.startTransaction({ "readConcern": { "level": "snapshot" }, "writeConcern": { "w": "majority" } })
replicaset [direct: primary] Employee> var empdetail = session.getDatabase('Employee').getCollection('EmpDetail')
replicaset [direct: primary] Employee> var designation = session.getDatabase('Employee').getCollection('DesignationDetail')
replicaset [direct: primary] Employee> empdetail.updateOne({Emp_ID:105}, {$set:{Designation:"Finance Executive"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
replicaset [direct: primary] Employee> designation.updateOne({Designation_ID:10003}, {$set:{Designation_Title:"Finance Executive"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
replicaset [direct: primary] Employee> session.commitTransaction()
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1687162058, i: 2 }),
    signature: {
      hash: Binary(Buffer.from("00000000000000000000000000000000", "hex"), 0),
      keyId: 0
    }
  },
  operationTime: Timestamp({ t: 1687162058, i: 1 })
}
```

Figure 9.15: Update Documents in Two Collections

6. To view the documents in the EmpDetail collection on the primary, run the command as:

```
db.EmpDetail.find()
```

Figure 9.16 shows the output of this query.

```
replicaset [direct: primary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("6490093070d9fd0f3fb43b57"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("6490093070d9fd0f3fb43b58"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("6490093070d9fd0f3fb43b59"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Finance Executive'
  },
  {
    _id: ObjectId("6490093070d9fd0f3fb43b5a"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.16: Updated Document in the EmpDetail Collection

7. Similarly, to view the documents in the DesignationDetail collection on the primary, run the command as:

```
db.DesignationDetail.find()
```

Figure 9.17 shows the output of this query.

```
replicaset [direct: primary] Employee> db.DesignationDetail.find()
[
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b61"),
    Designation_ID: 10001,
    Designation_Title: 'Software Engineer',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b62"),
    Designation_ID: 10011,
    Designation_Title: 'Project Manager',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b63"),
    Designation_ID: 10002,
    Designation_Title: 'HR Manager',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b64"),
    Designation_ID: 10012,
    Designation_Title: 'HR Executive',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b65"),
    Designation_ID: 10003,
    Designation_Title: 'Finance Executive',
    Department: 'Finance'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b66"),
    Designation_ID: 10013,
    Designation_Title: 'Finance Manager',
    Department: 'Finance'
  }
]
```

Figure 9.17: Updated Document in the DesignationDetail Collection

8. To verify that the changes have been reflected on the secondary also, on the secondary node, run the command as:

```
db.EmpDetail.find()
```

Figure 9.18 shows the output of this query.

```
replicaset [direct: secondary] Employee> db.EmpDetail.find()
[
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b69"),
    Emp_ID: 105,
    Emp_Name: 'Linda Michael',
    Gender: 'Female',
    Designation: 'Finance Executive'
  },
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b68"),
    Emp_ID: 103,
    Emp_Name: 'Richard Franklin',
    Gender: 'Male',
    Designation: 'HR Manager'
  },
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b6a"),
    Emp_ID: 102,
    Emp_Name: 'John Edward',
    Gender: 'Male',
    Designation: 'Software Engineer'
  },
  {
    _id: ObjectId("64900bd570d9fd0f3fb43b67"),
    Emp_ID: 101,
    Emp_Name: 'Oliver Smith',
    Gender: 'Male',
    Designation: 'Software Engineer'
  }
]
```

Figure 9.18: Updated Document in the EmpDetail Collection

9. Similarly, to view the documents in the DesignationDetail collection on the secondary, run the command as:

```
db.DesignationDetail.find()
```

Figure 9.19 shows the output of this query.

```
replicaset [direct: secondary] Employee> db.DesignationDetail.find()
[
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b66"),
    Designation_ID: 10013,
    Designation_Title: 'Finance Manager',
    Department: 'Finance'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b64"),
    Designation_ID: 10012,
    Designation_Title: 'HR Executive',
    Department: 'Human Resources'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b61"),
    Designation_ID: 10001,
    Designation_Title: 'Software Engineer',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b65"),
    Designation_ID: 10003,
    Designation_Title: 'Finance Executive',
    Department: 'Finance'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b62"),
    Designation_ID: 10011,
    Designation_Title: 'Project Manager',
    Department: 'Product Development'
  },
  {
    _id: ObjectId("64900bbf70d9fd0f3fb43b63"),
    Designation_ID: 10002,
    Designation_Title: 'HR Manager',
    Department: 'Human Resources'
  }
]
```

Figure 9.19: Updated Document in the DesignationDetail Collection

9.4 Summary

- Transactions in MongoDB are a single logical unit of work that involves single or multiple operations.
- The integrity of data in the database is protected by the four properties of transactions—Atomicity, Consistency, Isolation, and Durability (ACID).
- MongoDB provides two transaction APIs—Callback API and Core API.
- MongoDB provides methods to create, commit, and abort transactions in a session.
- Two major error labels in MongoDB are `TransientTransactionError` and `UnknownTransactionCommitResult`.
- MongoDB sessions can contain multiple transactions. Only one of these transactions remains active at a time.

Test Your Knowledge

1. Which of the following statements are true about Callback API in MongoDB?
 - a. It starts a transaction without an explicit call.
 - b. It does not incorporate error handling logic
for `TransientTransactionError` and `UnknownTransactionCommitResult`.
 - c. It starts a transaction only upon an explicit call.
 - d. Automatically incorporates error handling logic
for `TransientTransactionError` and `UnknownTransactionCommitResult`.
2. Which of the following options is not true about aborting transactions in MongoDB?
 - a. By default, a transaction is aborted if it runs for more than 30 seconds.
 - b. The `Session.abortTransaction` method is used to roll back the changes made by the operations in a transaction.
 - c. If an error occurs during aborting a transaction, MongoDB drivers retry the abort method even when the `retryWrites` parameter in the connection string is set to `false`.
 - d. When a transaction is rolled back, all the changes made by the operations in the transaction are discarded.
3. Which of the following is the correct syntax to start a session and execute a transaction?
 - a. `var session = db.startSession()`
 - b. `var session = session.startSession()`
 - c. `var session = db.getMongo().startSession()`
 - d. `var session = db.getMongo().startTransaction()`

4. Which of the following is the correct syntax to create a variable Inventdetail that represents the Invdetail collection in the Inventory database within the session?
- a. var Inventdetail =
db.getDatabase('Inventory').getCollection('Invdetail')
 - b. var Inventdetail =
Startsession.getDatabase('Inventory').getCollection('Invdetail')
 - c. var Inventdetail = db.Inventory.Invdetail
 - d. var Inventdetail =
session.getDatabase('Inventory').getCollection('Invdetail')
5. Which of the following method is used to save the changes made by the operations in a transaction and make the changes visible outside the transaction?
- a. db.commitTransaction
 - b. Session.commitTransaction
 - c. Session.transactionCommit
 - d. db.transactionCommit

Answers to Test Your Knowledge

1	a, d
2	a
3	c
4	d
5	b

Try it Yourself

1. Implement a three-member replica set with the P-S-S architecture.
2. Create a database named `Inventory` which includes a collection named `sales_invent`.
3. In the primary server, insert the given three documents into the `sales_invent` collection.

```
[  
  {  
    cust_id:1002,  
    customername: "Richard",  
    gender:"M",  
    purchased_product:"cereals",  
    quantity:6,  
    price:60  
  },  
  {  
    cust_id:1005,  
    customername: "Williams",  
    gender:"M",  
    purchased_product:"Vegetables",  
    quantity:10,  
    price:150  
  },  
  {  
    cust_id:1007,  
    customername: "John",  
    gender:"M",  
    purchased_product:"Baby Food",  
    quantity:3,  
    price:300  
  }  
]
```

4. In the primary server, create an index for the field `cust_id` and view the documents in the `sales_invent` collection.
5. Start a session and start a transaction to perform the two given operations:
 - a. Update the `purchased_product` field of the customer with `cust_id` as 1007 to “Dairy product”.
 - b. Delete the document where `cust_id` is 1002.
6. In the primary server, commit the transaction given in question 5 and check whether the two operations that are executed in the transaction are visible in the secondary servers.

7. In the primary server, start another new session and start a transaction to insert one document into the `sales_invent` collection as:

```
{  
    cust_id:1009,  
    customername: "Smith",  
    gender:"M",  
    purchased_product:"Fruits",  
    quantity:5,  
    price:180  
}
```

8. In the primary server, abort the transaction given in question 7 and check whether the insert operation that is performed in the transaction is not reflected in the secondary servers.



SESSION 10

MONGODB TOOLS

Learning Objectives

In this session, students will learn to:

- Explain the method to connect MongoDB Compass with MongoDB deployment
- Describe how to use MongoDB Compass to perform the create, insert, update, select, and drop operations in a database
- Explain how to install MongoDB BI Connector
- Describe the process for integrating the MongoDB BI Connector with the MongoDB deployment
- Describe how to connect MongoDB database to BI tools using ODBC drivers

Mongo shell is a command-line interface that can be used to work with MongoDB databases. On the same lines, MongoDB provides MongoDB Compass, a Graphical User Interface (GUI)-based interface to work with databases. Compass can be used to perform all the functions that users can perform using mongo shell.

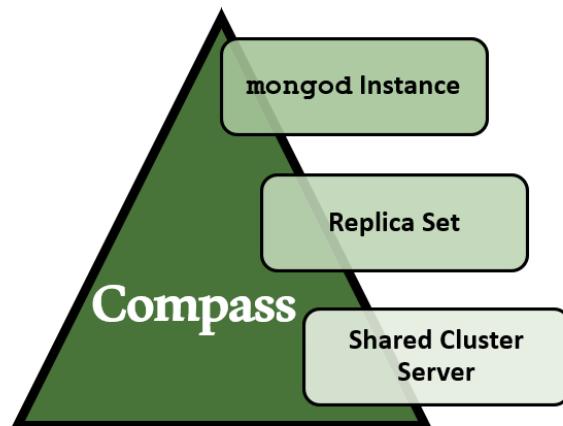
Data in MongoDB databases is often unstructured and expands faster. Consider a social networking application. The number of users and their posts (text, images, and videos) increase manifold in a short span of time. Data

analysts analyze data from these databases and create various reports. Tools used for such analysis can be traditional or modern. To share the data stored in the database to these tools, MongoDB uses Business Intelligence Connector (BI Connector) and Open Database Connectivity (ODBC) drivers.

This session will provide an overview of how to connect Compass with MongoDB deployment. It will explore the methods provided by Compass to perform the create, insert, update, select, and drop operations on database collections. The session will then explain how to install MongoDB BI Connector and connect it with MongoDB deployment. It will also explain how to create a system Data Source Name (DSN) and access MongoDB collection data from within Microsoft Excel.

10.1 Connect MongoDB Compass with MongoDB Deployment

MongoDB Compass is a visual interactive tool that helps the user to manage the documents in the database efficiently. Large volumes of data in MongoDB databases can be grouped and analyzed using Compass. Users can connect to Compass from:

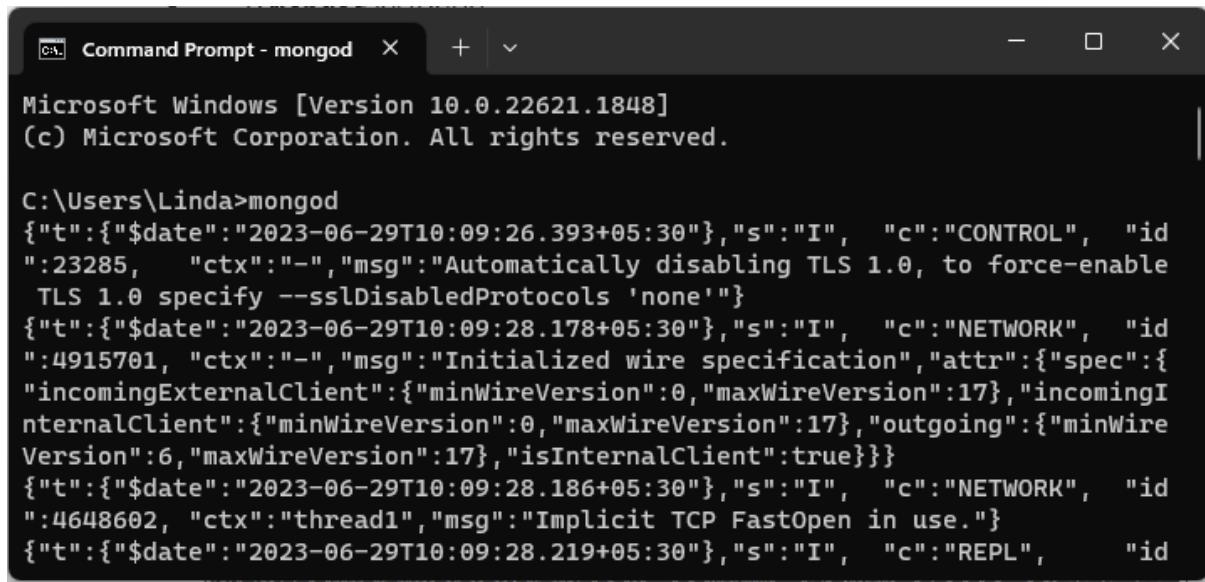


Compass is a free-to-use tool that can run on Linux, Mac, or Windows. To connect Compass with a `mongod` instance:

1. Open the command prompt window.
2. Start the `mongod` instance using the command:

```
mongod
```

Figure 10.1 shows the output of the command.



```
Microsoft Windows [Version 10.0.22621.1848]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Linda>mongod
{"t":{"$date":"2023-06-29T10:09:26.393+05:30"},"s":"I", "c":"CONTROL", "id":23285, "ctx":"-", "msg":"Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2023-06-29T10:09:28.178+05:30"},"s":"I", "c":"NETWORK", "id":4915701, "ctx":"-", "msg":"Initialized wire specification", "attr":{"spec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":17}, "incomingInternalClient":{"minWireVersion":0,"maxWireVersion":17}, "outgoing":{"minWireVersion":6,"maxWireVersion":17}, "isInternalClient":true}}}
{"t":{"$date":"2023-06-29T10:09:28.186+05:30"},"s":"I", "c":"NETWORK", "id":4648602, "ctx":"thread1", "msg":"Implicit TCP FastOpen in use."}
{"t":{"$date":"2023-06-29T10:09:28.219+05:30"},"s":"I", "c":"REPL", "id":4648603, "ctx":"-", "msg":"Sharding configuration loaded from file"}
```

Figure 10.1: Starting a mongod Instance

3. To open Compass, click **Start** → **All apps** →.
4. In the list of apps, scroll down and double-click **MongoDBCompass**.

The MongoDB Compass Setup wizard opens as shown in Figure 10.2.

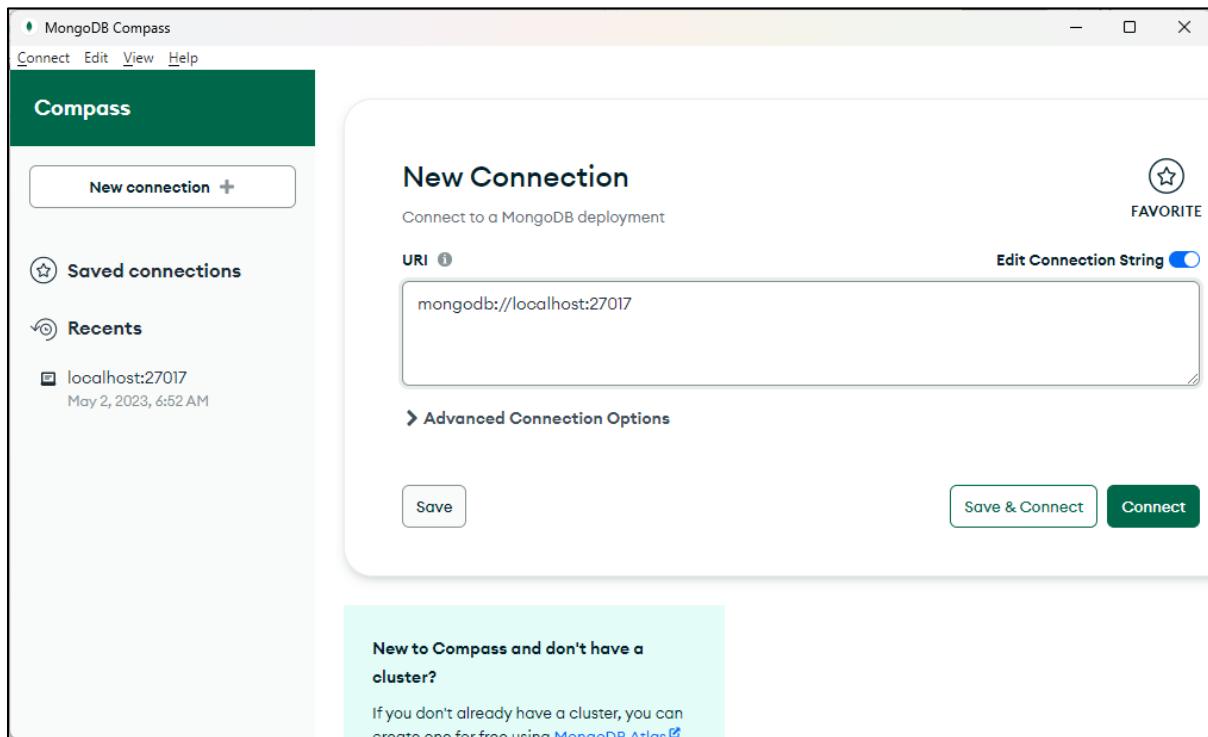


Figure 10.2: MongoDB Compass Setup

5. Click **Connect** or **Save and Connect** option.

On clicking this option, the user is redirected to the home page of Compass.

As shown in Figure 10.3, Compass is now connected to MongoDB that is listening to port 27017.

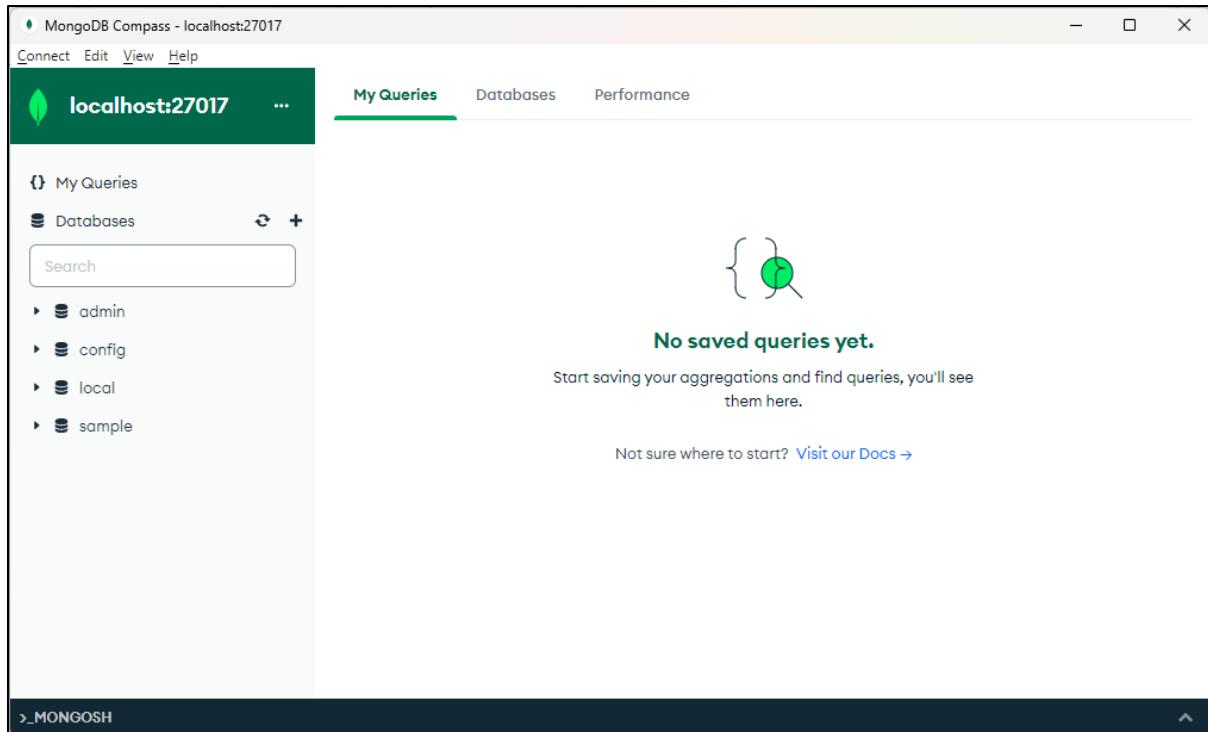


Figure 10.3: Compass Connected to MongoDB

10.2 Manage Database Using Compass

MongoDB makes it easier to create, manage, and optimize databases and collections to meet specific user requirements. Despite the size of the datasets, the user must be able to:

- Create collections
- Insert documents
- Update documents
- Query documents
- Delete documents
- Drop collections

All these operations can be easily done using Compass. To manage databases using Compass:

1. Open the **MongoDB Compass** home page.

2. Click the **Databases** tab.

The **Databases** tab lists the existing databases in your MongoDB deployment as shown in Figure 10.4.

The screenshot shows the MongoDB Compass interface with the title "MongoDB Compass - localhost:27017". The left sidebar has "My Queries" and "Databases" selected. The main area is titled "Databases" and shows four databases: "admin", "config", "local", and "sample". Each database card displays its storage size, number of collections, and number of indexes. A "Create database" button is at the top, and a "MONGOSH" prompt is at the bottom.

Database	Storage size	Collections	Indexes
admin	20.48 kB	1	1
config	24.58 kB	1	2
local	36.86 kB	1	1
sample			

Figure 10.4: Database Tab in Compass

To view information about the existing databases:

1. Click the database name.
2. Click the collection name to view the documents in that collection.
3. Click **Databases** on the left pane to go back to the **Databases** tab page.

Let us now explore the various functions used to manage databases using Compass.

10.2.1 Create Database

To create a new database:

1. On the **Database** tab, click **Create database**.

The **Create Database** dialogue box appears as shown in Figure 10.5.

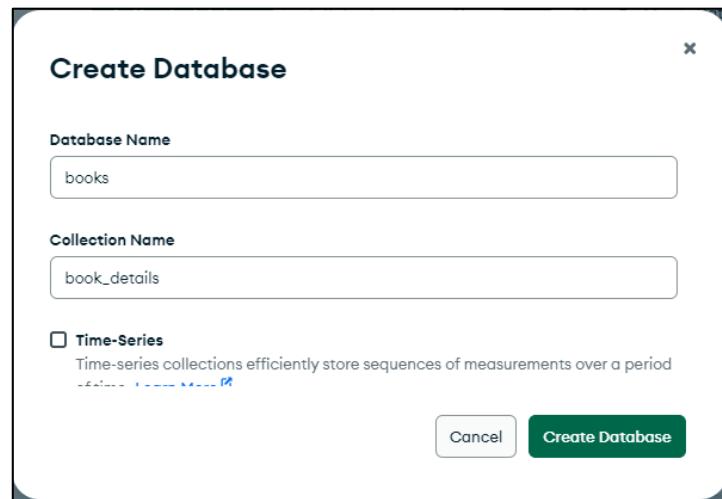


Figure 10.5: Create Database Dialogue Box

2. Type the database name as `books` and the collection name as `book_details`.
3. Click **Create Database**.

The database details appear as shown in Figure 10.6.

A screenshot of the MongoDB Compass interface. The top bar shows 'MongoDB Compass - localhost:27017/books.book_details'. The left sidebar shows databases like admin, books (which is selected), config, local, and sample. Under 'books', the 'book_details' collection is selected. The main panel shows the 'books.book_details' collection with 0 documents and 1 index. It has tabs for 'Documents', 'Aggregations', 'Schema', 'Explain Plan', 'Indexes', and 'Validation'. Below the tabs is a search bar and a 'Find' button. A large 'ADD DATA' button is at the bottom left. The status bar at the bottom says 'This collection has no data' and 'It only takes a few seconds to import data from a JSON or CSV'.

Figure 10.6: books Database

10.2.2 Insert Documents

To insert documents into the `books_details` collection:

1. Click the **Add Data** drop-down and select **Insert document** as shown in Figure 10.7.

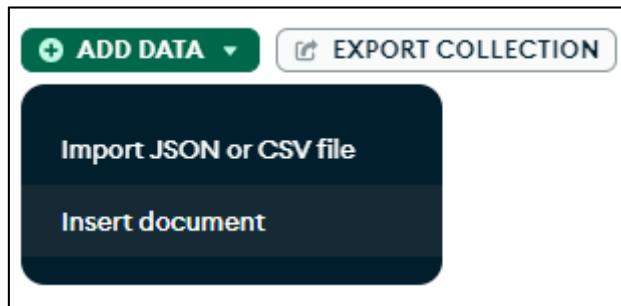


Figure 10.7: Add Data

The default view for inserting documents in a database appears as shown in Figure 10.8.

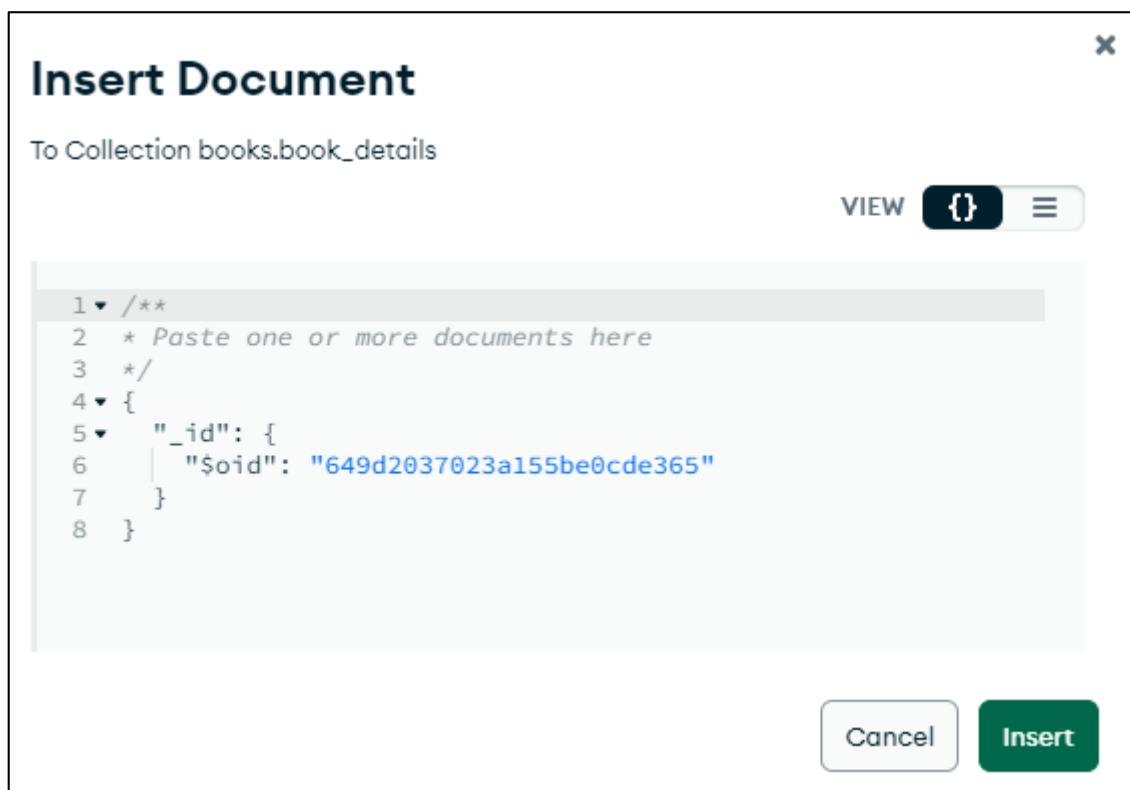


Figure 10.8: Default View for Inserting Documents in a Database

2. To import JSON files, ensure that the `{ }` brackets on the top-right is selected and delete the existing command.

3. To insert four documents into the `book_details` collection in the `books` database, enter the command as:

```
[{"_id" : 101,"book_title":"NoSQL  
Distilled","book_description": "A Brief Guide to the  
Emerging World of Polyglot  
Persistence","author":"Martin Fowler","edition":2},  
 {"_id" : 102,"book_title": "Python Crash  
Course","book_description": "A Hands-On, Project-Based  
Introduction to Programming","author": "Eric  
Matthes","edition":1},  
 {"_id" : 103,"book_title": "Seven Databases in Seven  
Weeks","book_description": "A Guide to Modern Databases  
and the NOSQL Movement","author": "Luc  
Perkins","edition":4},  
 {"_id" : 104,"book_title": "Next Generation  
Databases","book_description": "NoSQL and Big  
Data","author": "Harrison","edition":3}]
```

Figure 10.9 shows the command window for inserting four documents into the collection, `book_details`.

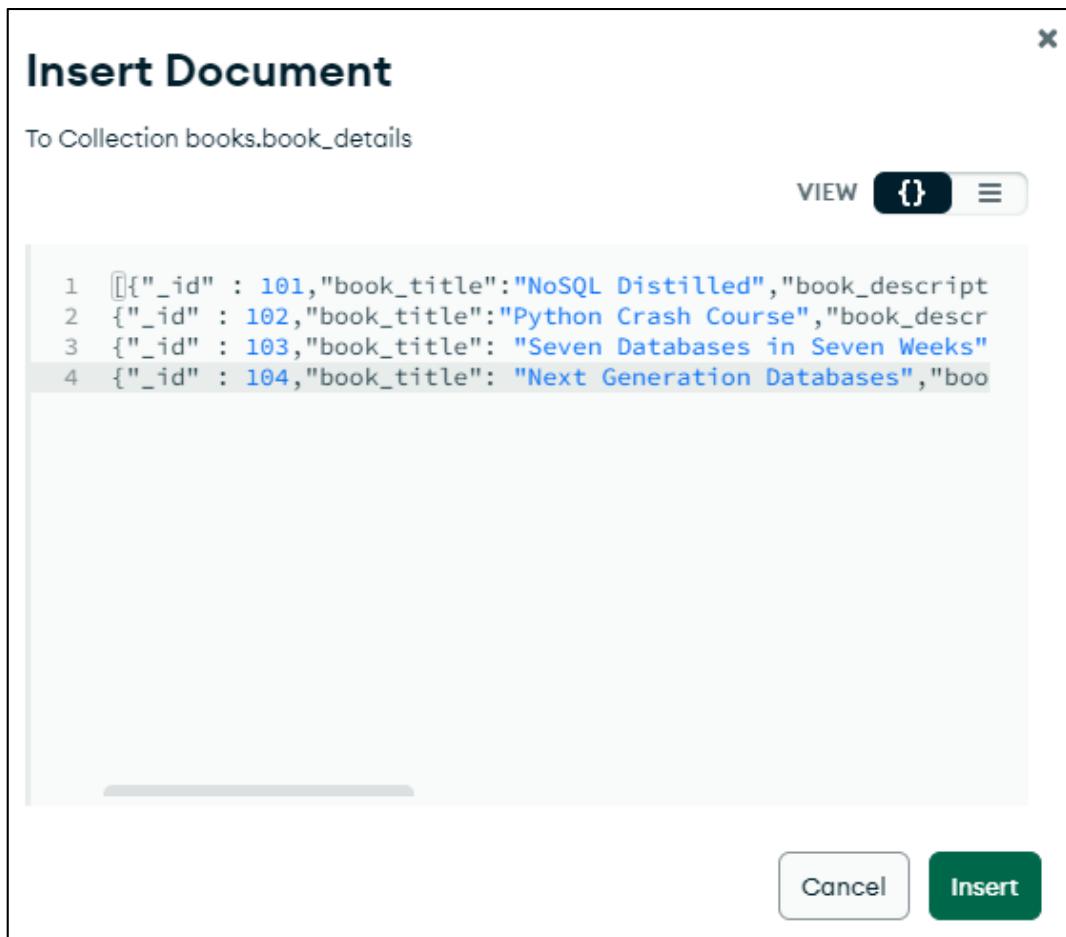


Figure 10.9: Inserting Four Documents

- Click **Insert**. Figure 10.10 shows the four documents inserted into the collection.

Document	Content
_id: 101	book_title: "NoSQL Distilled" book_description: "A Brief Guide to the Emerging World of Polyglot Persistence" author: "Martin Fowler" edition: 2
_id: 102	book_title: "Python Crash Course" book_description: "A Hands-On, Project-Based Introduction to Programming" author: "Eric Matthes" edition: 1
_id: 103	book_title: "Seven Databases in Seven Weeks" book_description: "A Guide to Modern Databases and the NOSQL Movement"

Figure 10.10: Inserted Documents

10.2.3 Modify Documents

After adding the documents to the database collection, the user can edit and modify specific documents in Compass. Existing fields can be removed or edited, and new fields can be added. Changes made to the collection can also be reverted back by the user.

Let us edit the document with `_id` as 101. To modify the document:

1. Hover the cursor over the document to be modified.
2. Select the **pencil icon** that appears on the right top side of the document as shown Figure 10.11.



Figure 10.11: Document Showing Pencil Icon

Change the `edition` of document to 3 as shown in Figure 10.12.

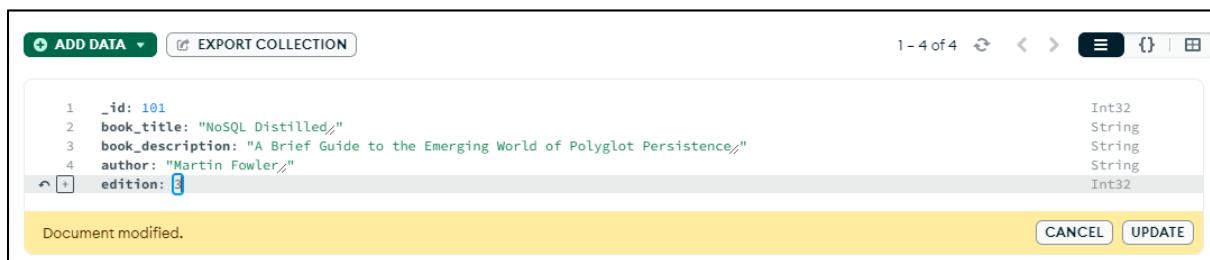


Figure 10.12: Editing the edition of Document with `_id` as 101

3. Click **UPDATE**. The value for the `edition` field is updated.

10.2.4 Query Data

The user can query documents using the filter option. To find a document with `book_title` as "NoSQL Distilled":

1. To filter the documents, in the **Filter** box, enter the command as:

```
{book_title: "NoSQL Distilled"}
```

2. Click **Find**. Figure 10.13 shows the filtered document.

The screenshot shows the MongoDB Compass interface for the collection 'books.book_details'. At the top, it displays '4 DOCUMENTS' and '1 INDEXES'. Below the header, there are tabs for 'Documents', 'Aggregations', 'Schema', 'Explain Plan', 'Indexes', and 'Validation'. The 'Documents' tab is selected. A filter bar at the top has the query '{book_title: "NoSQL Distilled"}'. To the right of the filter are buttons for 'Reset', 'Find', and 'More Options'. Below the filter, there are buttons for '+ ADD DATA' and 'EXPORT COLLECTION'. On the far right, there are navigation icons and a status bar showing '1-1 of 1'. The main content area shows a single document with the following fields and values:
_id: 101
book_title: "NoSQL Distilled"
book_description: "A Brief Guide to the Emerging World of Polyglot Persistence"
author: "Martin Fowler"
edition: 3

Figure 10.13: Filtered Document



In the filter, users can use all of the MongoDB query operators except the \$text and \$expr operators.

3. To view the query aggregation pipeline stages, expand **More Options** dropdown at the top-right section of the document window. Figure 10.14 shows all the query aggregation pipeline stages available for the user:
 - **Project** the fields in the document
 - **Sort** the documents
 - **Skip** the number of documents
 - **Limit** the number of documents

The screenshot shows the 'More Options' dropdown expanded. The dropdown includes sections for 'Project', 'Sort', 'Collation', and 'Skip' and 'Limit' settings. The 'Project' section shows the query '{ field: 0 }'. The 'Sort' section shows the query '{ field: -1 } or [['field', -1]]. The 'Collation' section shows the query '{ locale: 'simple' }'. To the right of the dropdown, there are fields for 'MaxTimeMS' (set to 60000) and 'Less Options'.

Figure 10.14: More Options

4. To reset the filter, click **Reset**.
5. Click **Find**. The collection shows all the documents.

10.2.5 Drop a Document

Users can drop or delete documents from the collections using Compass. To

drop a document with `_id` as 102:

1. Hover the cursor over the document to be dropped.
2. Click the **Remove** icon as shown in Figure 10.15.

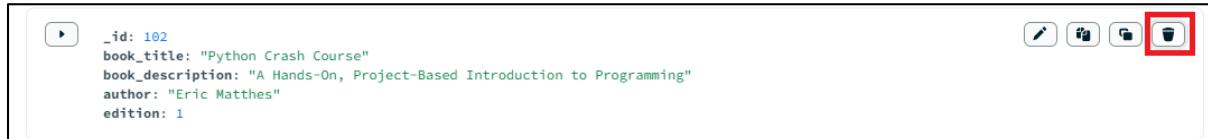


Figure 10.15: Remove Icon

A dialog box appears as shown in Figure 10.16.

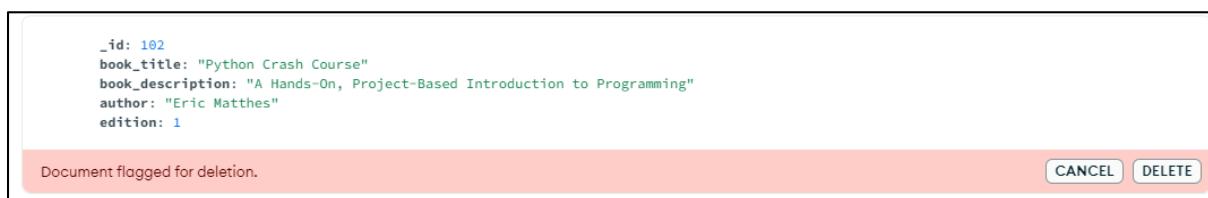


Figure 10.16: Removing the Document with `_id` as 102

3. Click **DELETE**.

Figure 10.17 shows the documents window after deletion of the document.

A screenshot of the MongoDB "books.book_details" collection interface. The top right shows "3 DOCUMENTS" and "1 INDEXES". The "Documents" tab is selected. A search bar at the top says "Type a query: { field: 'value' }". Below it are "ADD DATA" and "EXPORT COLLECTION" buttons. The main area lists three documents:

- `_id: 101`
`book_title: "NoSQL Distilled"`
`book_description: "A Brief Guide to the Emerging World of Polyglot Persistence"`
`author: "Martin Fowler"`
`edition: 3`
- `_id: 103`
`book_title: "Seven Databases in Seven Weeks"`
`book_description: "A Guide to Modern Databases and the NOSQL Movement"`
`author: "Luc Perkins"`
`edition: 4`
- `_id: 104`
`book_title: "Next Generation Databases"`
`book_description: "NoSQL and Big Data"`

Figure 10.17: Collection After Document Deletion

Thus, Compass can be used to manage MongoDB databases easily. Next, the user would want to analyze the data in the databases to get insights into the data, such as trends, what customers like, what they do not like, and so on. For this analysis, users can use any traditional or modern Business Intelligence (BI) tools. However, the tools must be able to connect with MongoDB to access the data in the databases. This is where MongoDB BI Connector will be useful.

10.3 Install MongoDB BI Connector

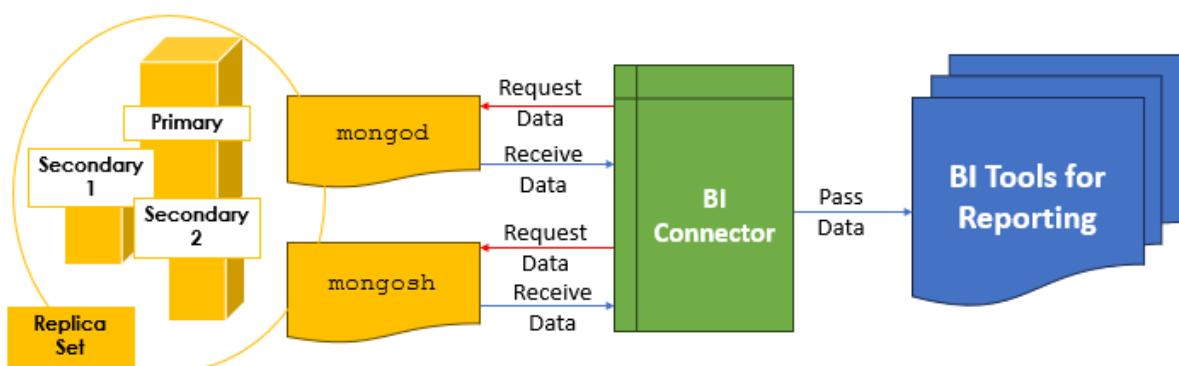
MongoDB databases store large volumes of three-dimensional data. Traditional business intelligence tools that deal with two-dimensional data are not equipped to handle the data stored in MongoDB databases.

The BI Connector tool offered by MongoDB allows the user to create SQL queries based on the data stored in MongoDB databases. The queried data can then be used to visualize and report through graphs and charts using powerful relational BI tools such as Tableau and Power BI.



Alternatively, users can use MongoDB Charts to visualize data directly from the MongoDB collections.

BI Connector acts as a layer between the MongoDB data clusters and the BI tools.



BI Connector does not store data. It only facilitates the BI tools to query data residing in the MongoDB databases.

The BI Connector deployment comprises four components:

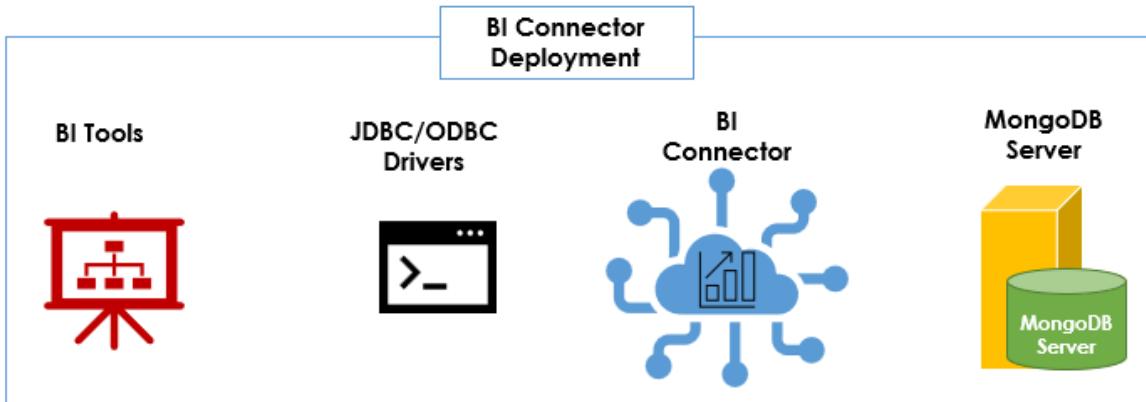


Table 10.1 describes some of the components included in the BI Connector deployment.

System Component	Description
BI Tool	It is a visualization and reporting tool, such as Tableau, PowerBI, or Microsoft Excel.
JDBC or ODBC driver	They provide methods to help the BI tools to connect the MongoDB instances through the BI Connector.
BI Connector	It is a tool that provides a relational schema for MongoDB databases. It translates SQL queries between the BI tool and the MongoDB instances.
MongoDB Database	It may be a self-managed MongoDB server with replica sets or a MongoDB Atlas cluster.

Table 10.1: BI Connector System Components

To install MongoDB BI Connector for Windows:

1. Open the browser and visit the URL:
<https://www.mongodb.com/download-center/bi-connector/releases>
2. On this page, under **2.14.7**, scroll down to the **Windows x64** section and download the file [mongodb-bi-win32-x86_64-v2.14.7.msi](#)
3. After the download is complete, run the installer. The **MongoDB BI Connector Setup** wizard opens as shown in Figure 10.18.



Figure 10.18: MongoDB BI Connector Setup wizard

2. Click **Next**.
The **End-User License Agreement** page opens.

3. Select **I accept the terms in the License Agreement** check box as shown in Figure 10.19.

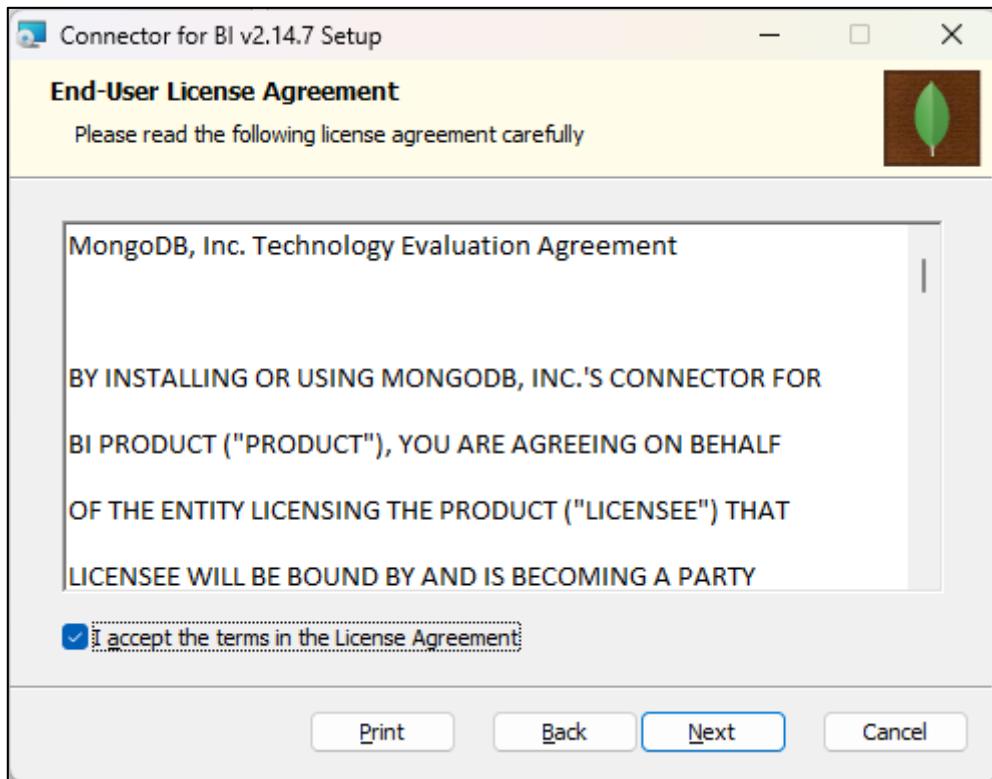


Figure 10.19: End-User License Agreement

4. Click **Next**.

The **Custom Setup** page opens as shown in Figure 10.20.

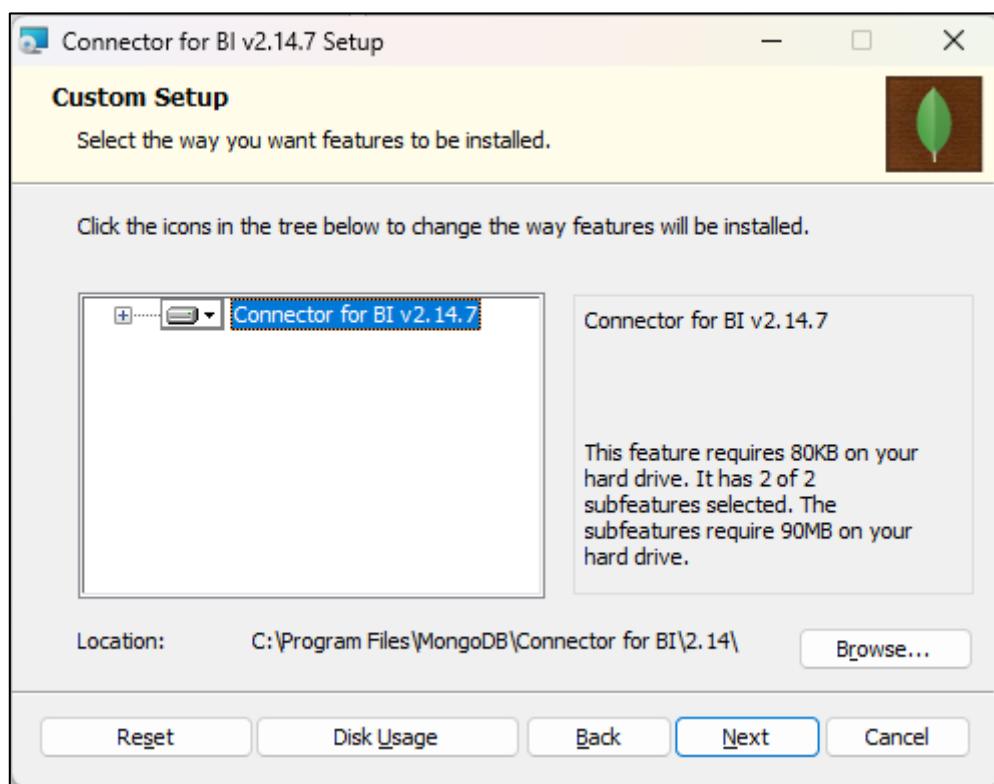


Figure 10.20: Custom Setup Page

5. Click **Next**.

The **Ready to Install Connector for BI** page opens as shown in Figure 10.21.

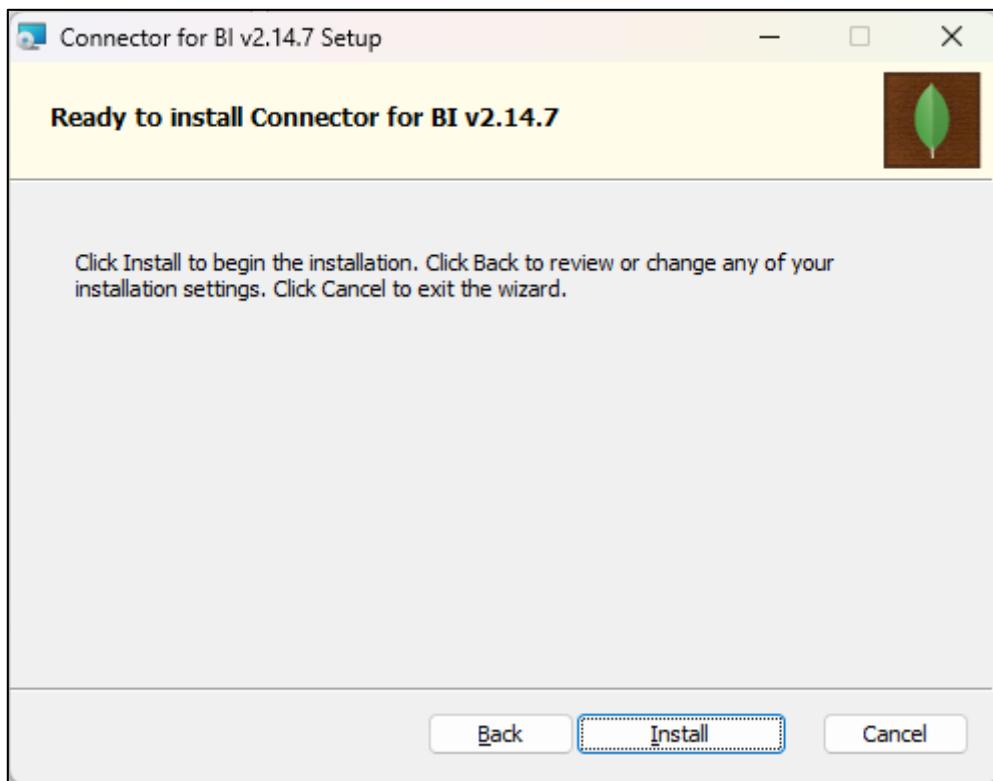


Figure 10.21: Ready to Install Connector for BI

6. Click **Install**.

After the installation is complete, the **Completed the Connector for BI** page opens as shown in Figure 10.22.



Figure 10.22: Completed the Connector for BI Setup Wizard

7. Click **Finish**.

Now, MongoDB BI Connector is installed.

10.4 Connect MongoDB BI Connector with MongoDB Deployment

A MongoDB instance can be connected to the BI tool using the BI Connector program, `mongosqld`. BI tools require a database schema to map the MongoDB databases and collections in the current `mongod` instance to a relational data model. Therefore, the user must either specify a schema file using `--schema` option or create a database schema by directly launching the `mongosqld` program.

To launch `mongosqld`:

1. Open the command prompt.
2. Change the drive to the path specified in the command:

```
cd C:\Program Files\MongoDB\Connector for BI\2.14\bin
```

3. Start mongosqld from the command line using the command:

```
mongosqld.exe
```

Figure 10.23 shows the output of this command.

```
C:\Program Files\MongoDB\Connector for BI\2.14\bin>mongosqld.exe
2023-06-29T14:35:25.470+0530 I CONTROL      [initandlisten] mongosqld starting: version=v2.14.7 p
id=10080 host=LAPTOP-KGV0C49M
2023-06-29T14:35:25.471+0530 I CONTROL      [initandlisten] git version: 3eb4fd411c7a1dc1776da62e
6a2d30e48b9366ab
2023-06-29T14:35:25.471+0530 I CONTROL      [initandlisten] OpenSSL version OpenSSL 1.0.2n-fips
7 Dec 2017 (built with OpenSSL 1.0.2s 28 May 2019)
2023-06-29T14:35:25.471+0530 I CONTROL      [initandlisten] options: {}
2023-06-29T14:35:25.471+0530 I CONTROL      [initandlisten] ** WARNING: Access control is not ena
bled for mongosqld.
2023-06-29T14:35:25.471+0530 I CONTROL      [initandlisten]
2023-06-29T14:35:25.652+0530 I NETWORK      [initandlisten] waiting for connections at 127.0.0.1:
3307
2023-06-29T14:35:26.131+0530 I SCHEMA       [sampler] sampling MongoDB for schema...
2023-06-29T14:35:26.164+0530 I SCHEMA       [sampler] mapped schema for 3 namespaces: "books" (1)
: ["book_details"]; "empl_det" (1): ["empl"]; "sample" (1): ["emp"]
```

Figure 10.23: mongosqld program

Now that the MongoDB databases are mapped to a schema using the mongosqld program, users can use the drivers to connect to the BI tool.

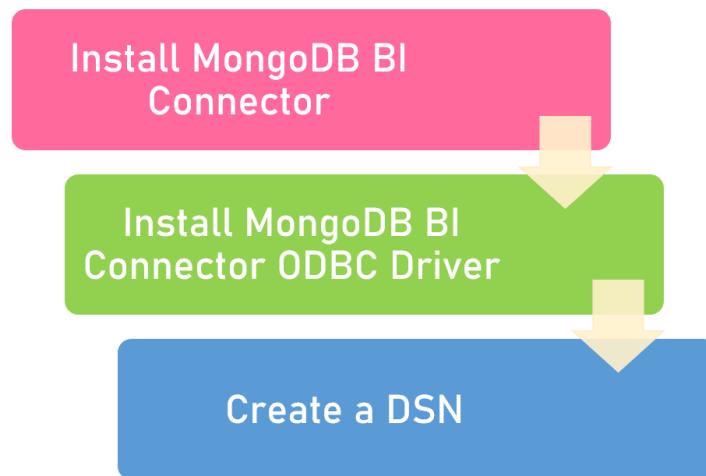
10.5 Connect MongoDB Database to BI Tools Using ODBC

Data from various databases can be exported from MongoDB by establishing a connection between MongoDB and those databases. The MongoDB BI Connector ODBC driver facilitates SQL clients to connect to MongoDB Connector for BI.



A DSN is a file that stores details about the database to which a connection must be established using the ODBC driver.

Steps required to establish a connection with an external database are:



MongoDB BI Connector is already installed (Refer Section 10.3).

For installing MongoDB BI Connector ODBC driver:

1. Open the browser and navigate to the URL:

<https://github.com/mongodb/mongo-bi-connector-odbc-driver/releases/>

2. To download the MongoDB BI Connector ODBC driver, scroll down and select:

[mongodb-connector-odbc-1.4.3-win-64-bit.msi](#)

Figure 10.24 shows the **MongoDB BI ODBC** driver download page.

The screenshot shows the MongoDB BI ODBC v1.4.3 download page. At the top, it says "v1.4.3" and "Latest". Below that, a note says "Version 1.4.3 adds support for macOS 12." and a link to "Please refer to the README and to the BI Connector reference documentation for usage instructions.". Under the heading "Assets" (with a count of 8), there is a list of files:

File	Size	Last Modified
mongodb-connector-odbc-1.4.3-macos-64-bit.dmg	28.7 MB	Jul 14, 2022
mongodb-connector-odbc-1.4.3-rhel-7.0-64.tar.gz	26 MB	Jul 14, 2022
mongodb-connector-odbc-1.4.3-ubuntu-14.04-64.tar.gz	25.9 MB	Jul 14, 2022
mongodb-connector-odbc-1.4.3-ubuntu-16.04-64.tar.gz	26 MB	Jul 14, 2022
mongodb-connector-odbc-1.4.3-win-32-bit.msi	26.7 MB	Jul 14, 2022
mongodb-connector-odbc-1.4.3-win-64-bit.msi	27.4 MB	Jul 14, 2022
Source code (zip)		Jul 13, 2022
Source code (tar.gz)		Jul 13, 2022

Figure 10.24: MongoDB BI ODBC Driver Download

3. After the download is complete, run the installer.

The **MongoDB ODBC 1.4.3 Setup** wizard opens as shown in Figure 10.25.



Figure 10.25: MongoDB ODBC Setup wizard

4. Click **Next**.

The **End-User License Agreement** page opens.

5. Select **I accept the terms in the License Agreement** check box as shown in Figure 10.26.

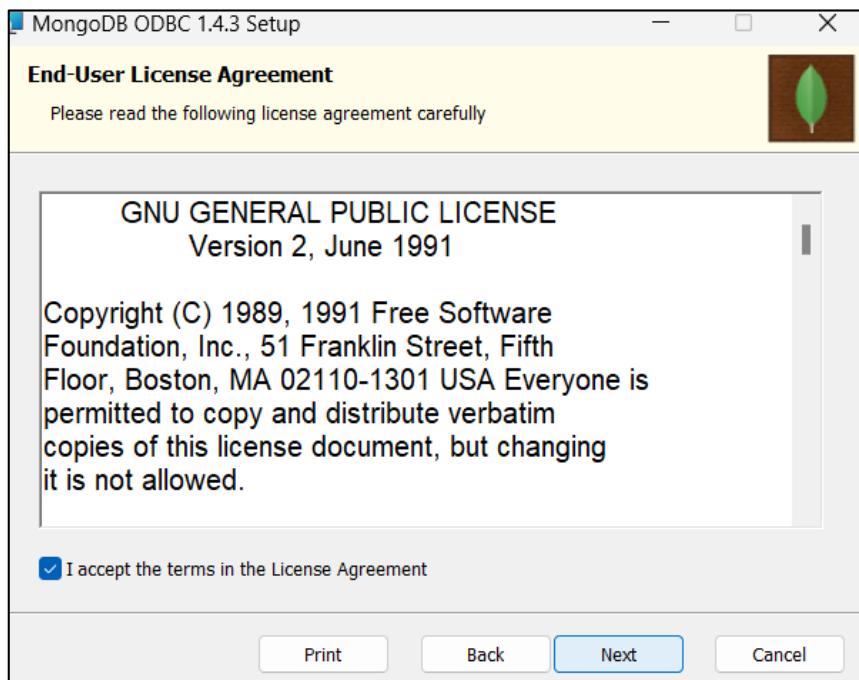


Figure 10.26: End-User License Agreement

6. Click **Next**.

The **Custom Setup** page opens as shown in Figure 10.27.

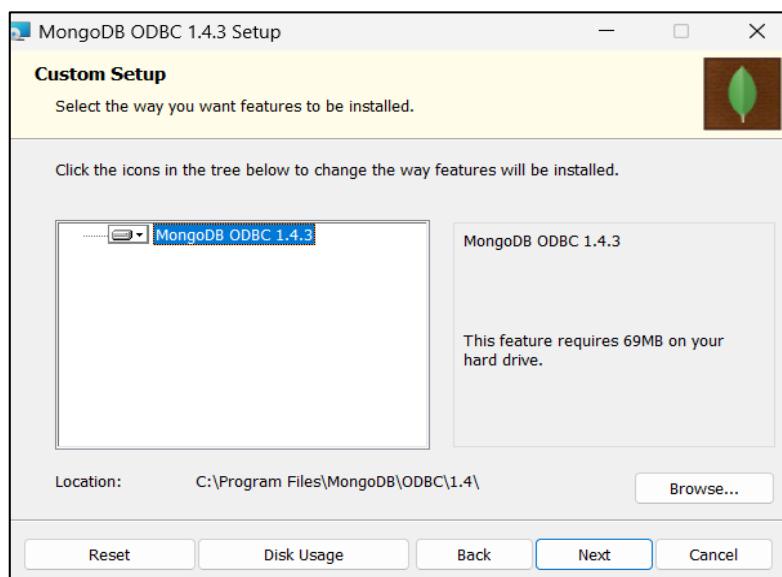


Figure 10.27: Custom Setup Page

7. Click **Next**.

The **Ready to Install MongoDB ODBC** page opens as shown in Figure 10.28.

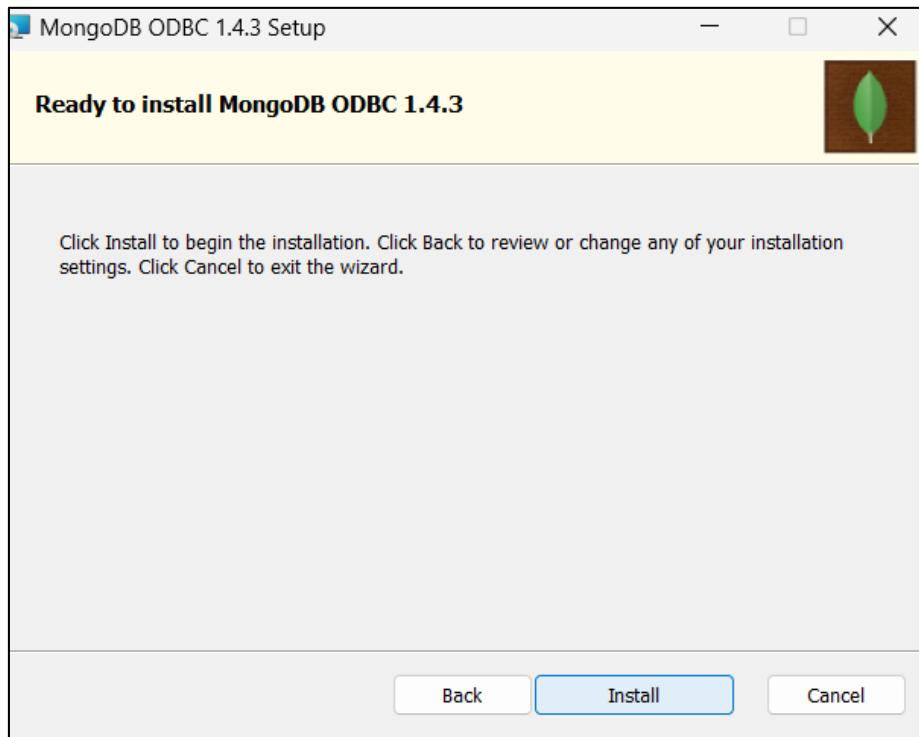


Figure 10.28: Ready to Install MongoDB ODBC

8. Click **Install**.

After the installation is complete, the **Completed the MongoDB ODBC** page opens as shown in Figure 10.29.



Figure 10.29: Completed the MongoDB ODBC Setup Wizard

9. Click **Finish**.

The MongoDB BI Connector ODBC driver is installed.

10.5.1 Create a Data Source Name (DSN)

The ODBC driver must know the database to which it must connect and the credentials to connect to the database. A DSN provides this required information to the ODBC driver. To create a DSN for Windows:

1. Open a command prompt window.
2. Start the `mongod` instance using the command:

```
mongod
```

3. Start `mongosqld` from the command line using the command:

```
mongosqld.exe
```

4. To open the **Microsoft ODBC Data Sources** Program:

- Open the **Control Panel** window.
- Select the **System and Security** option.
- Select the **Windows Tools** option as shown in Figure 10.30.

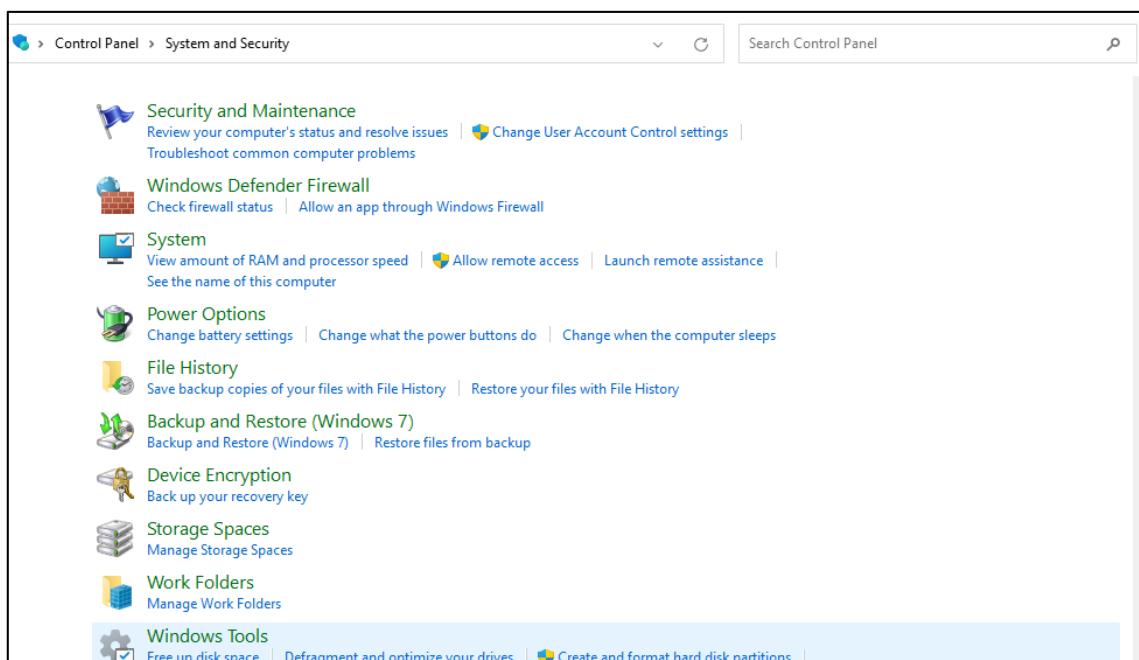


Figure 10.30: Select Windows Tools

- Choose the Program Version (**64-bit or 32-bit**) that is appropriate for the system and ODBC driver version. For the demonstration purpose, the program version is selected as 64-bit.

The **ODBC Data Source Administrator (64 bit)** wizard opens.

- On this page, click the **System DSN** tab.
Click the **Add** button as shown in Figure 10.31.

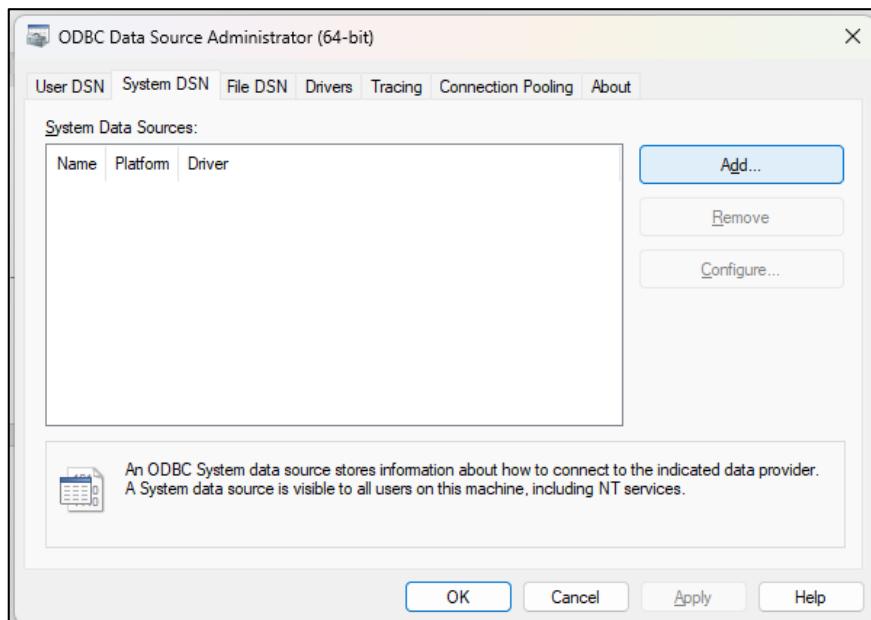


Figure 10.31: ODBC Data Source Administrator (64 bit) Setup Wizard

The **Create New Data Source** window opens.

- Select either the **MongoDB ODBC ANSI Driver** or the **MongoDB ODBC Unicode Driver** from the list of available drivers as shown in Figure 10.32. For the demonstration purpose, the MongoDB ODBC Unicode Driver is selected.

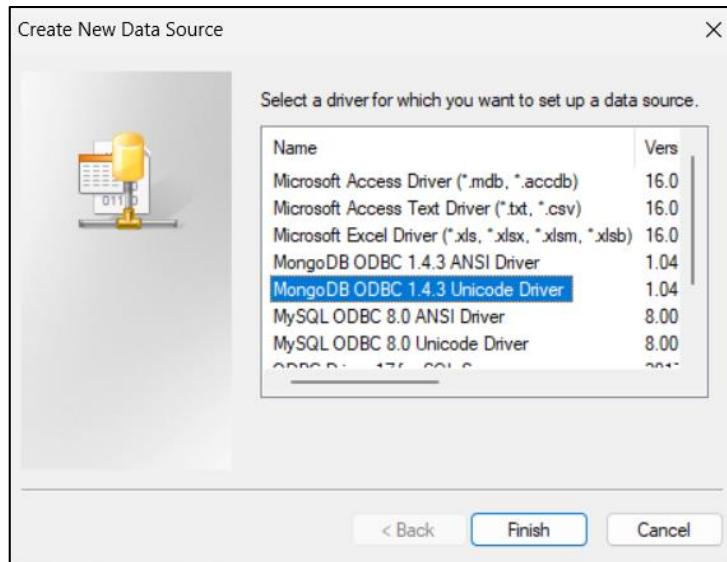


Figure 10.32: Create New Data Source

7. Click **Finish**.

The **MongoDB ODBC Data Source Configuration** dialog box opens.

8. In the dialog box:

- In the **Data Source Name** box, enter the name for the data source. For example, `excelBIconnector`.
- In the **Port** box, specify the port to be used to connect to the data source. For example, **3307**.
- In the **Database** drop-down, select the database to connect to. For example, `books`.
- Click **Test**.

The **Test Result** message box appears as shown in Figure 10.33.

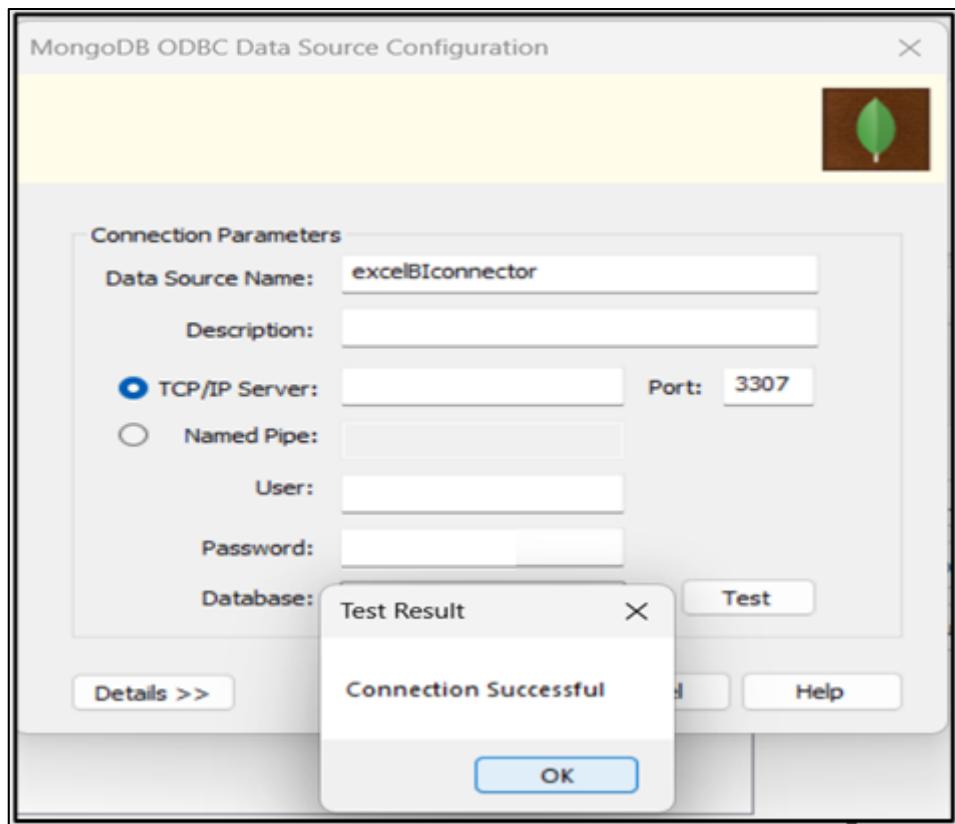


Figure 10.33: MongoDB ODBC Data Source Configuration dialogue box

9. Click **OK**.

The connection is successfully established.

After creating the DSN, the users can use it to import data from MongoDB into various BI tools.

10.5.2 Import Data from MongoDB to Microsoft Excel

To import data from the `book_details` collection into Microsoft Excel:

1. Open a command prompt window.
2. Start the `mongod` instance using the command:

```
mongod
```

3. Start `mongosqld` from the command line using the command:

```
mongosqld.exe
```

4. Open an Excel workbook into which data from the `book_details` MongoDB collection must be imported.
5. In the Excel workbook, on the **Data** tab, in the **Get & Transform** group,

select the **Get Data** option as shown in Figure 10.34.

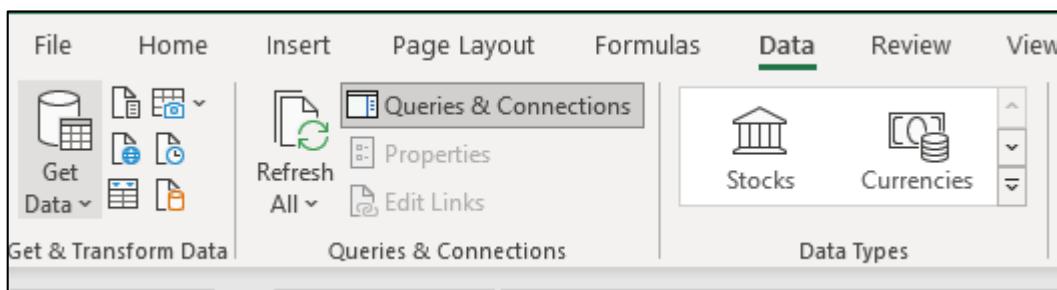


Figure 10.34 Get Data Option

6. In the drop-down menu that appears, select **From Other Sources**, and in the context menu that appears, select **From ODBC** as shown in Figure 10.35.

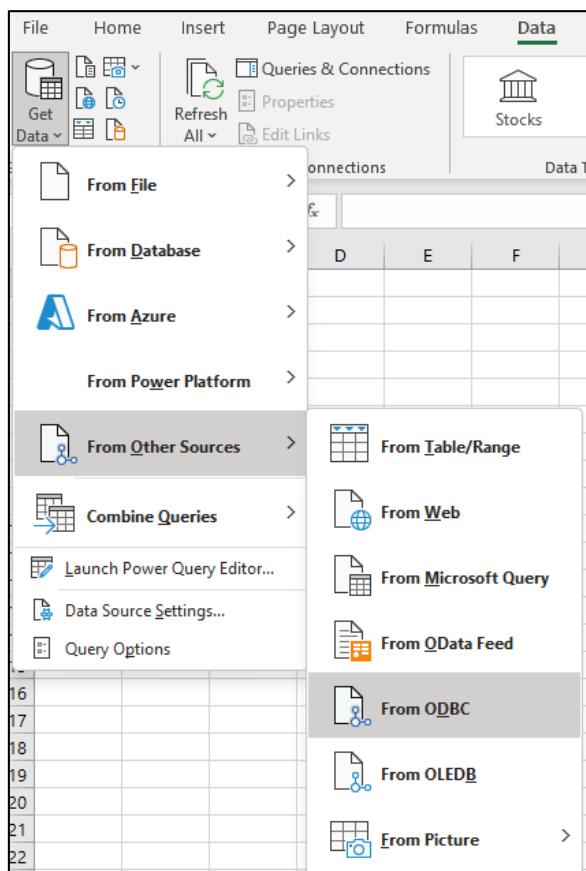


Figure 10.35: Select From ODBC option

The **From ODBC** window appears.

7. In the **Data source name (DSN)** drop-down menu, select **excelBIconnector** as shown in Figure 10.36.

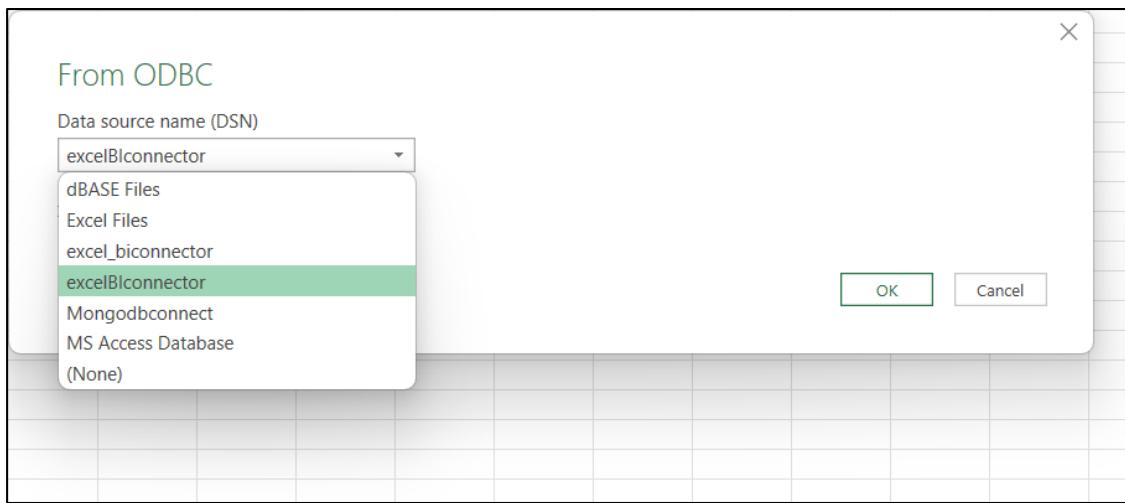


Figure 10.36: Local BI Connector mongosqld Setup Wizard

3. Click **OK**.

The **Navigator** window opens as shown in Figure 10.37.

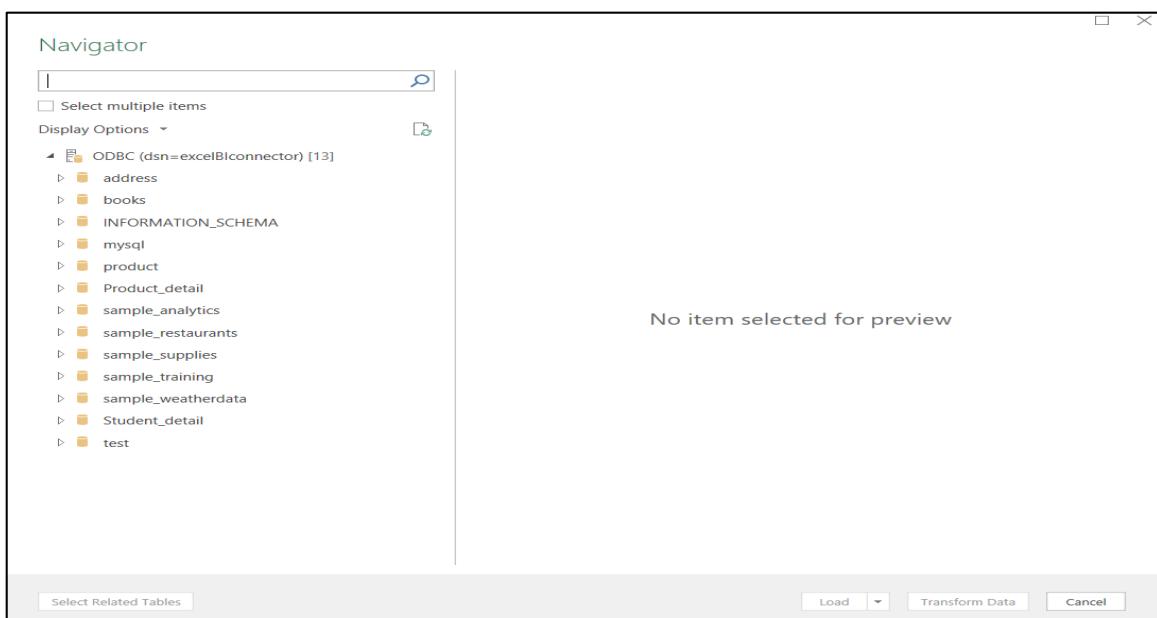


Figure 10.37: Navigator Window

4. In the **Navigator** window, under **books** select **book_details** as shown in Figure 10.38.

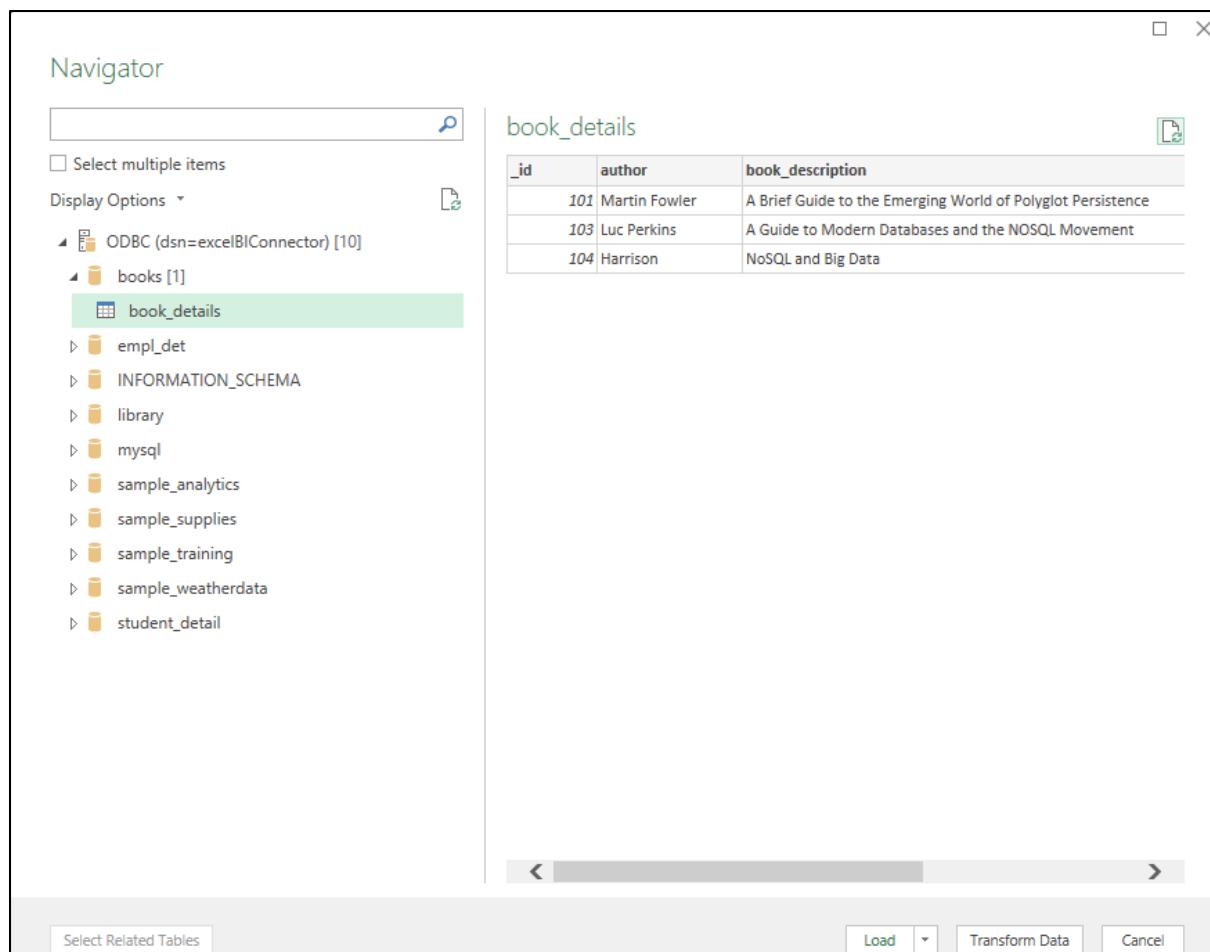


Figure 10.38: Navigator Window

5. Click **Load**.

The data from the `book_details` MongoDB collection is now imported into the Microsoft Excel spreadsheet.

The imported data is shown in Figure 10.39.

	A	B	C	D	E	F
1	<code>id</code>	<code>author</code>	<code>book_description</code>	<code>book_title</code>	<code>edition</code>	
2	101	Martin Fowler	A Brief Guide to the Emerging World of Polyglot Persistence	NoSQL Distilled	3	
3	103	Luc Perkins	A Guide to Modern Databases and the NOSQL Movement	Seven Databases in Seven Weeks	4	
4	104	Harrison	NoSQL and Big Data	Next Generation Databases	3	
5						

Figure 10.39: Imported Data from book_details

10.6 Summary

- MongoDB Compass is a visual interactive tool that helps users to manage documents in the database efficiently.
- Users can connect to Compass from a `mongod` instance, any of the replica set, or any of the sharded cluster servers.
- Relational business intelligence tools such as Tableau and Power BI can be used to envision, chart, and review three-dimensional MongoDB data by connecting them with the help of the MongoDB BI Connector.
- A `mongod` instance and BI tool are connected by the BI Connector program, `mongosqld`. MongoDB collections and databases must be mapped to a data schema by `mongosqld`.
- The MongoDB BI Connector ODBC driver allows SQL clients to connect to MongoDB Connector for BI.

Test Your Knowledge

1. Which of the query operators of MongoDB given here cannot be used in the `Filter` field of MongoDB Compass?
 - a. `$text`
 - b. `$slice`
 - C. `$expr`
 - d. `$all`
2. Which of the options when used in the query bar of MongoDB Compass specify the fields to return in the resulting data?
 - a. Match
 - b. Project
 - c. Collation
 - d. Limit
3. Which of the statement given here is not true about the BI connector?
 - a. It provides a relational schema.
 - b. It only acts as a conduit between the MongoDB cluster and business intelligence tools.
 - c. It stores data in the form of documents.
 - d. It translates SQL queries between the BI tool and MongoDB.
4. From the given options, which BI Connector program connects `mongod` instance to BI tool?
 - a. `mongostat`
 - b. `mongodump`
 - c. `mongodrql`
 - d. `mongosqld`
5. Consider a file, `stud.csv`, has been saved in the location, `C:\stud_det`. Which command imports `stud.csv` into MongoDB?
 - a. `mongoimport --db stud_det --collection stud --type=csv --headerline --file C:\stud_det\stud.csv`
 - b. `mongoimport --db stud_det --collection stud --type=csv --file C:\stud_det\stud.csv`
 - c. `mongoimport --db stud_det --collection stud --type=json --headerline --file C:\stud_det\stud.csv`
 - d. `mongoimport --db stud_det --collection stud --type csv --headerline --file C:\stud_det\stud.csv`

Answers to Test Your Knowledge

1	a, c
2	b
3	c
4	d
5	a

Try it Yourself

1. Connect Mongodb Compass with the mongod instance which is listening to port 27017.
2. Create a database named Student_detail, and a collection named Stud_personal using MongoDB Compass.
3. Insert the given three documents into the stud_personal collection using MongoDB Compass.

```
[  
  {  
    stud_id:10401  
    name: "Adam",  
    gender:"M",  
    hobbies:["Singing","Gardening","Playing Chess"],  
    blood_group:"AB+ve"  
  },  
  {  
    stud_id:10405  
    name: "Franklin",  
    gender:"M",  
    hobbies:["Dancing","Cooking"],  
    blood_group:"B+ve"  
  },  
  {  
    stud_id:10408  
    name: "Michael",  
    gender:"M",  
    hobbies:["Photography"],  
    Blood_group:"B+ve"  
  }  
]
```

Using the collection stud_personal, perform the tasks:

- a. Filter the details of the students who have the "B+ve" blood group.
 - b. Exclude the _id field and display the first two documents of the stud_personal collection which includes only the fields stud_id, name, and hobbies.
 - c. Remove a document having stud_id = 10405 from the stud_personal collection.
4. Install MongoDB BI Connector for Windows.
 5. Install MongoDB BI Connector ODBC driver.
 6. Connect MongoDB BI Connector with MongoDB.

7. Create a Data Source Name (DSN) for Windows.
8. Connect MongoDB to Microsoft Excel and import the `stud_personal` collection into Microsoft Excel using the MongoDB Connector for BI.



SESSION 11

MONGODB CLOUD

Learning Objectives

In this session, students will learn to:

- Explain how to create an Atlas account and set up a cluster
- Describe how to access the MongoDB Atlas cluster
- Explain how to import data into the Atlas cluster from a MongoDB instance and manage the imported data
- Explain how to export data from the Atlas cluster into a MongoDB instance and manage the exported data
- Describe how to perform administrative tasks in the MongoDB cluster

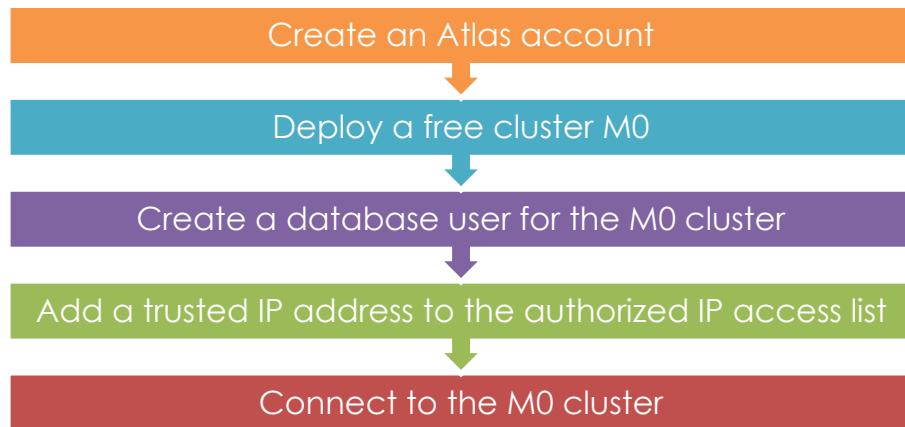
MongoDB Atlas is a cloud-based MongoDB service that provides a reliable and convenient platform for storing and accessing data. It offers a user-friendly interface and eliminates the requirement for manual infrastructure management. MongoDB Atlas is a popular cloud-based database used by organizations to store large amounts of data, such as user data for social media, e-commerce, or gaming. For example, Netflix and Spotify use MongoDB Cloud to store user profiles, watch history, recommendations, songs, artists, and playlists.

This session will walk through the entire process of setting up MongoDB Atlas, from creating an account to accessing the cluster. In addition, this session will

explore the methods for importing and exporting data between a local MongoDB instance and the MongoDB Atlas cluster. It will also cover how to perform essential administrative tasks such as pausing, resuming, and terminating the MongoDB cluster.

11.1 Get Started with MongoDB Atlas

Steps to set up MongoDB Atlas are:



11.1.1 Create an Atlas Account

MongoDB Atlas is a cloud-based service that requires an account to access its servers. With an Atlas account, the Atlas dashboard can be accessed to deploy MongoDB clusters, create database users, and manage databases.

To create an Atlas account, use any one of the three options:



Google account is the preferred method. However, email address can also be used.

To create the Atlas account:

1. Open the browser and navigate to the URL:
<https://www.mongodb.com/cloud/atlas/register>

The **MongoDB Atlas Landing** page opens as shown in Figure 11.1.

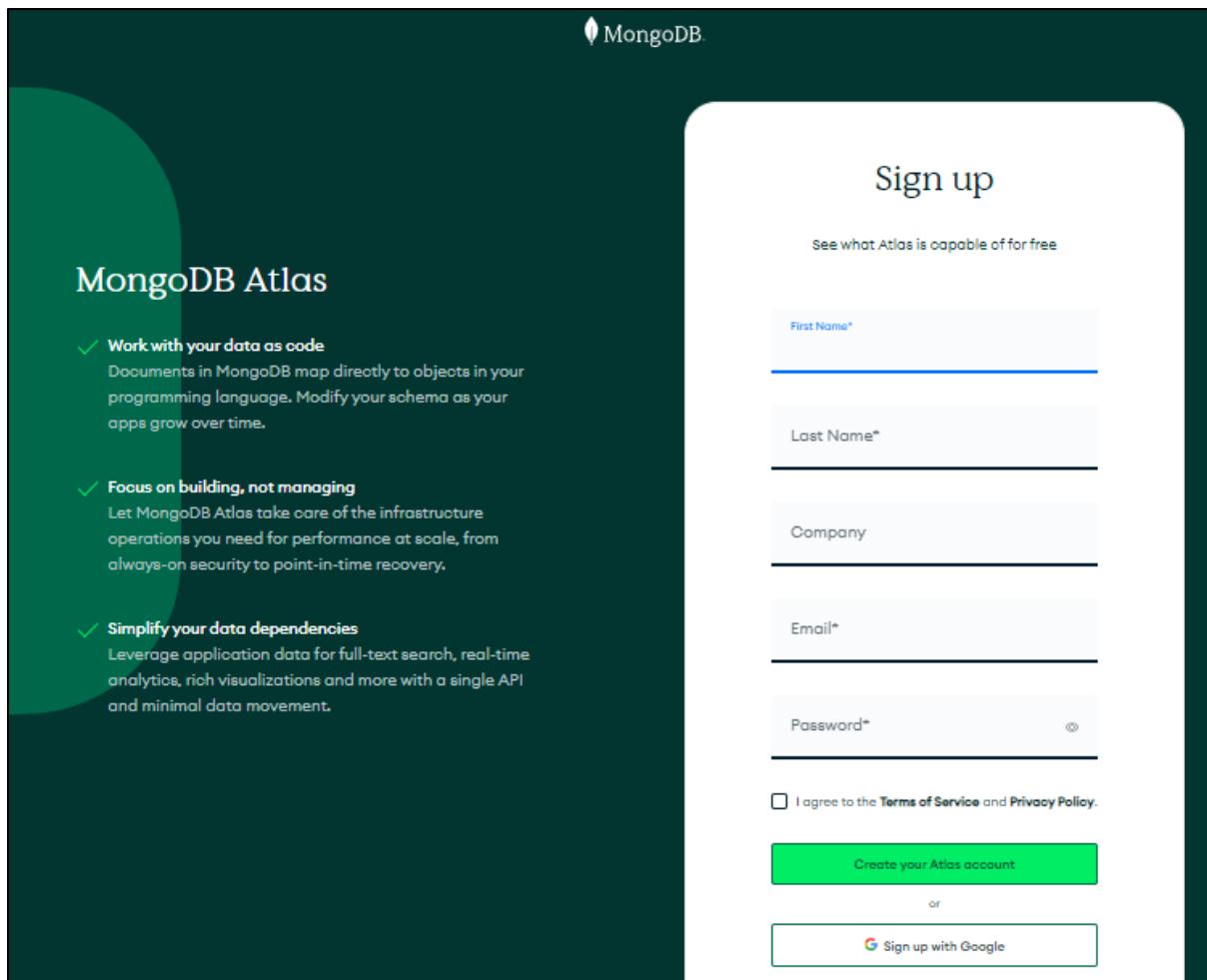


Figure 11.1: MongoDB Atlas Landing Page

2. To sign up using the preferred method or Google account, on this page, scroll down and click **Sign up with Google**.

The **Accept Privacy Policy and Terms of Service** page opens as shown in Figure 11.2.

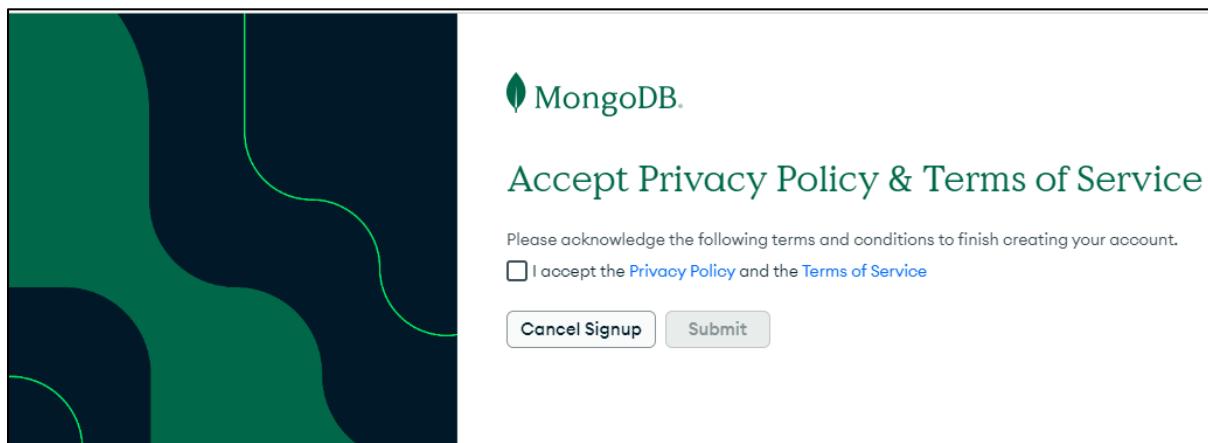


Figure 11.2: Accept Privacy Policy and Terms of Service Page

3. To understand how the data is collected, used, and shared, review the privacy policy and terms of service. To proceed, select the **I accept the Privacy Policy and Terms of Service** check box.
4. Click **Submit**.

This completes the sign-up process and welcomes the user to the account as shown in Figure 11.3.

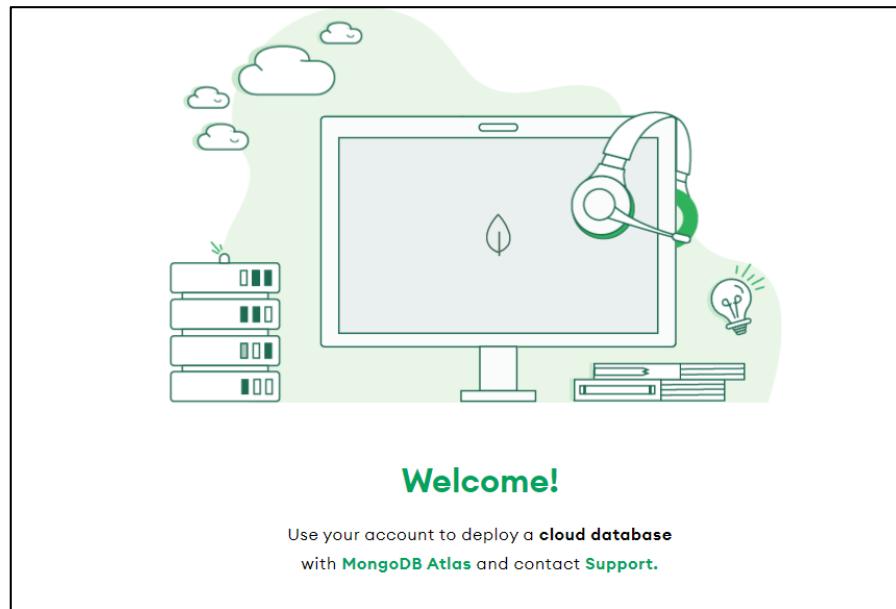


Figure 11.3: Welcome to Account Page

Atlas automatically creates a default organization and a project. For first time users, Atlas prompts to respond to a few queries related to the user and the project as shown in Figure 11.4.

Welcome to Atlas!

Tell us a few things about yourself and your project.

What is your goal today?

Your answer will help us guide you to successfully getting started with MongoDB Atlas.

Build a new application
 Migrate an existing application
 Explore what I can build
 Learn MongoDB

What type of application are you building?

I'm just exploring ▾

What is your preferred language?

We'll use this to customize code samples and content we share with you. You can always change this later.

Other ▾

Finish

Figure 11.4: Welcome to Atlas Page

5. On this page, respond to all the queries appropriately and click **Finish**. This completes the Atlas account creation process. The next step is to create a cluster.

11.1.2 Deploy a Free Cluster

After creating an Atlas account, a cluster can be deployed. A cluster is a group of servers that work together to store and manage data. To create a MongoDB cluster in Atlas, the user must specify the required size and configuration, and Atlas takes care of the rest.

M0 clusters are a free, entry-level option for MongoDB users who are learning the database or developing small, proof-of-concept applications. These clusters are limited in terms of storage and features, but they are a great way to get started with MongoDB without any upfront costs. These clusters are small-scale, but they never expire, and they provide access to a subset of Atlas features.

To deploy M0 or the free cluster:

1. From the **Deploy your database** page shown in Figure 11.5, select the **M0** option.

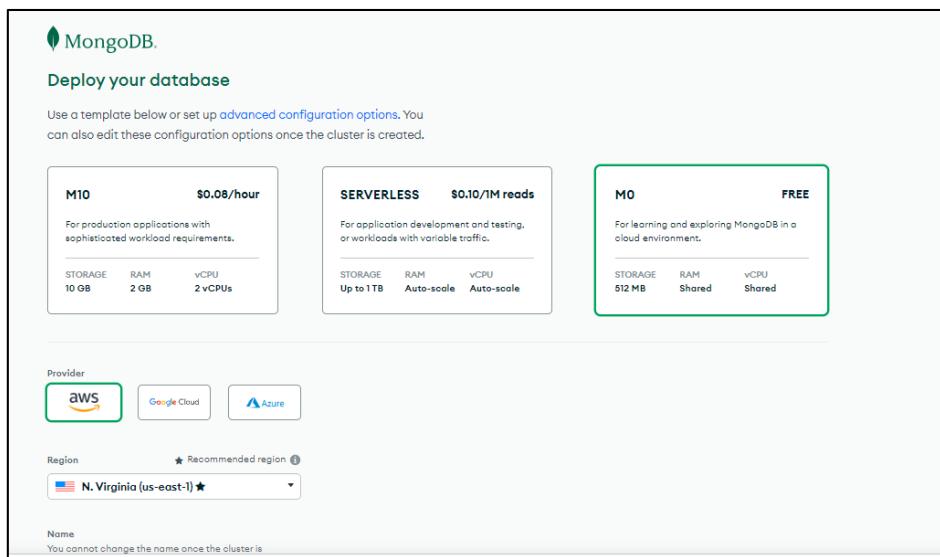


Figure 11.5: Deploy Your Database Page

2. Scroll down to make more selections as shown in Figure 11.6.
 - Select the preferred **Provider**.
Atlas supports M0 free clusters on Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. In this case, the **AWS** option is selected.
 - Select the preferred **Region**.
Atlas displays only the cloud provider regions that support M0 free clusters. Here, **N. Virginia (us-east-1)** is selected.
 - Enter a name for the cluster in the **Name** box.
Users can specify any name for the cluster, as long as it contains American Standard Code for Information Interchange or ASCII letters, numbers, and hyphens. Retain **Cluster0** as given here.

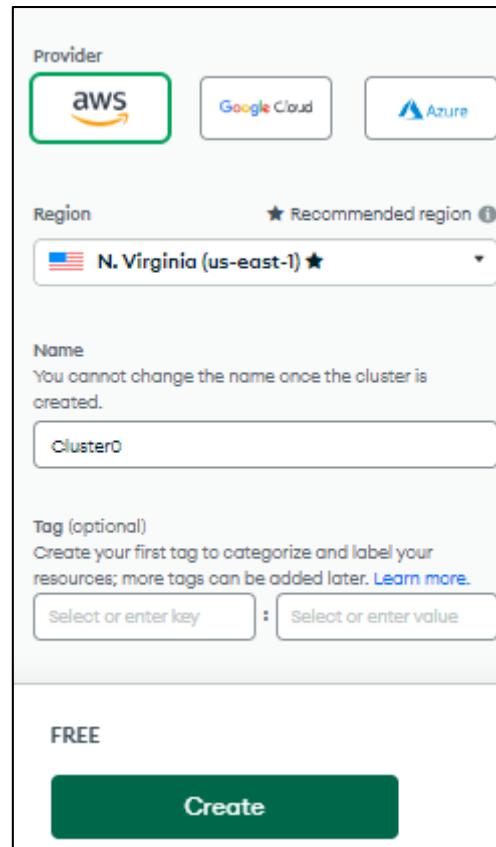


Figure 11.6: Deploy Your Database Page

6. Click **Create**.

The **Security Quickstart** page opens as shown in Figure 11.7 with the success message.

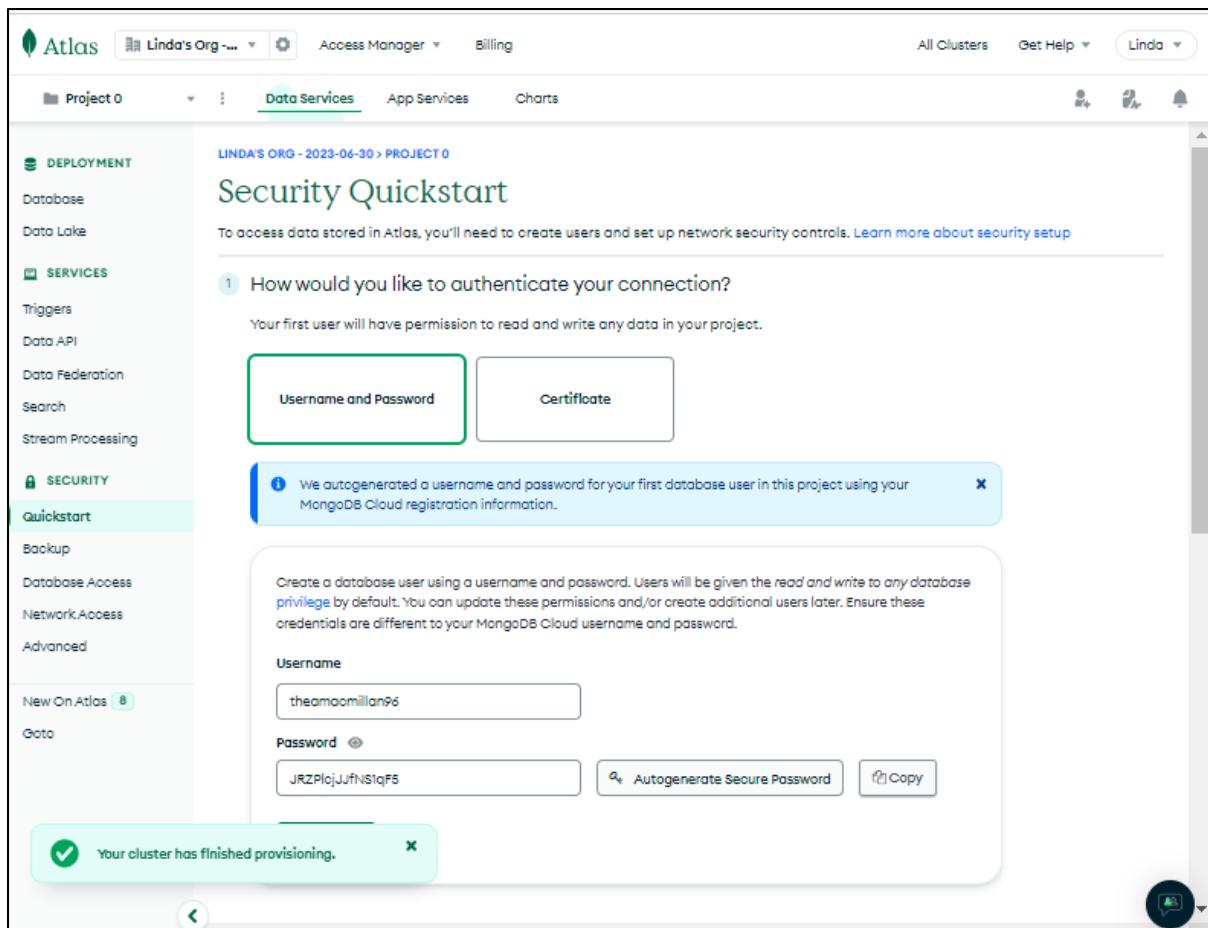


Figure 11.7: Success Message on the Security Quickstart Page

This completes the deployment of free cluster M0.

11.1.3 Create a Database User for the M0 Cluster

To protect data, Atlas requires users to authenticate as valid MongoDB database users before they can access the cluster. To allow users to access the MongoDB database on the cluster, a database user must be created after deploying the cluster. The database user can then be granted permission to access the cluster and the database hosted on the cluster. The database user accounts can be created for different users and roles, and each account can be granted different permissions.

There is a difference between database users and Atlas users in MongoDB Atlas:

Database Users

Can access databases hosted in Atlas.

Atlas Users

Can log in to Atlas but cannot access the MongoDB databases.



A database user account can ONLY be created by a user who has either Organization Owner or Project Owner permission.

MongoDB Atlas autogenerates a username and password for the first database user in the project using the already submitted MongoDB Cloud registration information.

To manually add a user account:

1. On the **Security Quickstart** page, as shown in Figure 11.8, enter the new username and credentials.

Security Quickstart

To access data stored in Atlas, you'll need to create users and set up network security controls. [Learn more about security setup](#)

- How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

[Username and Password](#)

[Certificate](#)

ⓘ We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information. **X**

Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username

theo

Password ⓘ

JRZPlcJJfNS1qF5

[Autogenerate Secure Password](#)

[Copy](#)

[Create User](#)

Figure 11.8: Security Quickstart Page

This username and password combination is used to grant a user access to databases and collections in the cluster within the Atlas project.

- To use a password that is auto generated by Atlas, click **Autogenerate Secure Password**.
- Click **Create User**.

This creates an Atlas user.

11.1.4 Add a Trusted IP Address to the Authorized IP Access List

After creating a database user, the deployed cluster is ready to be connected. Each device that connects to a network is assigned a unique identifier called an Internet Protocol or IP address. It is used to identify and locate devices on the network, and to route data between devices.

In MongoDB Atlas, an IP access list is a list of trusted IP addresses that are allowed to connect to a MongoDB Atlas cluster. By limiting access to specific IPs, unauthorized connections can be prevented so as to ensure that only trusted sources can connect to the cluster.

To secure the MongoDB Atlas cluster, access should be restricted only to specified IP addresses as shown in Figure 11.9. The current IP address is automatically added. If not added, the user can add it to the IP Access List.

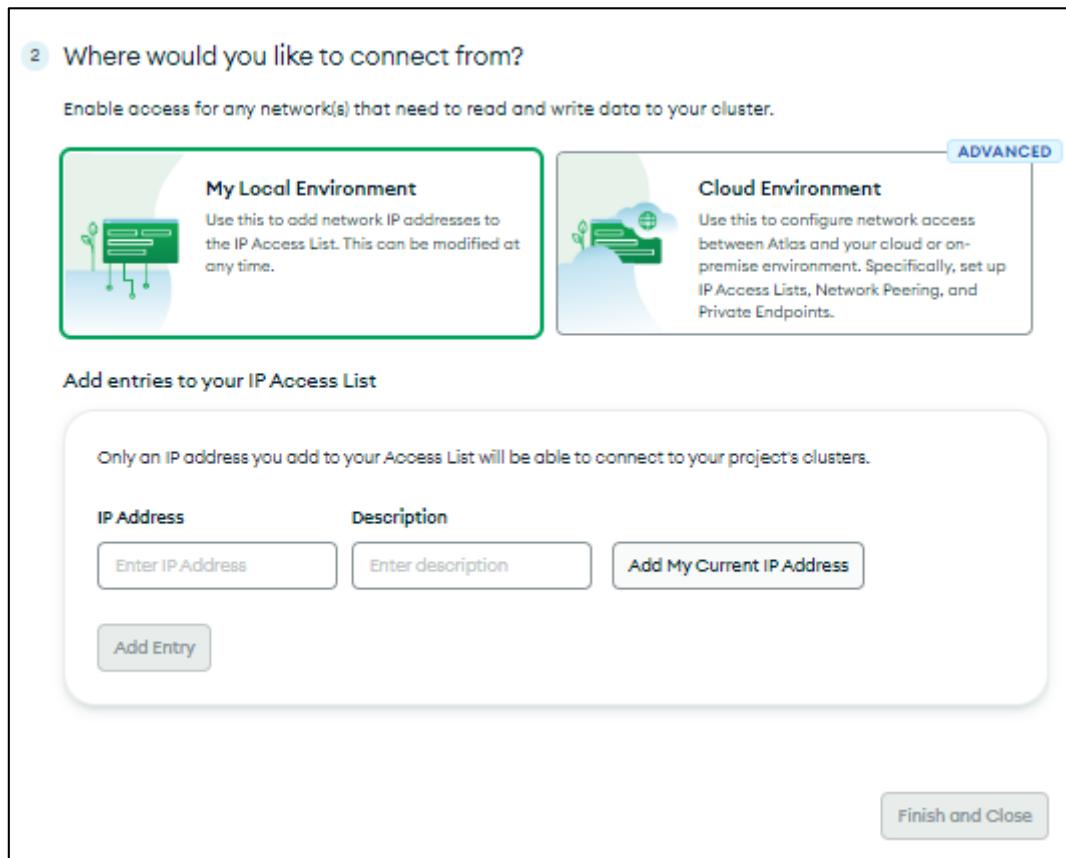


Figure 11.9: Add entries to your IP Access List Page

1. To add the current IP address, on this page, click **Add My Current IP Address**.
The IP Access List is updated with the current IP address.
2. Click **Finish and Close**.
A success message appears as shown in Figure 11.10.

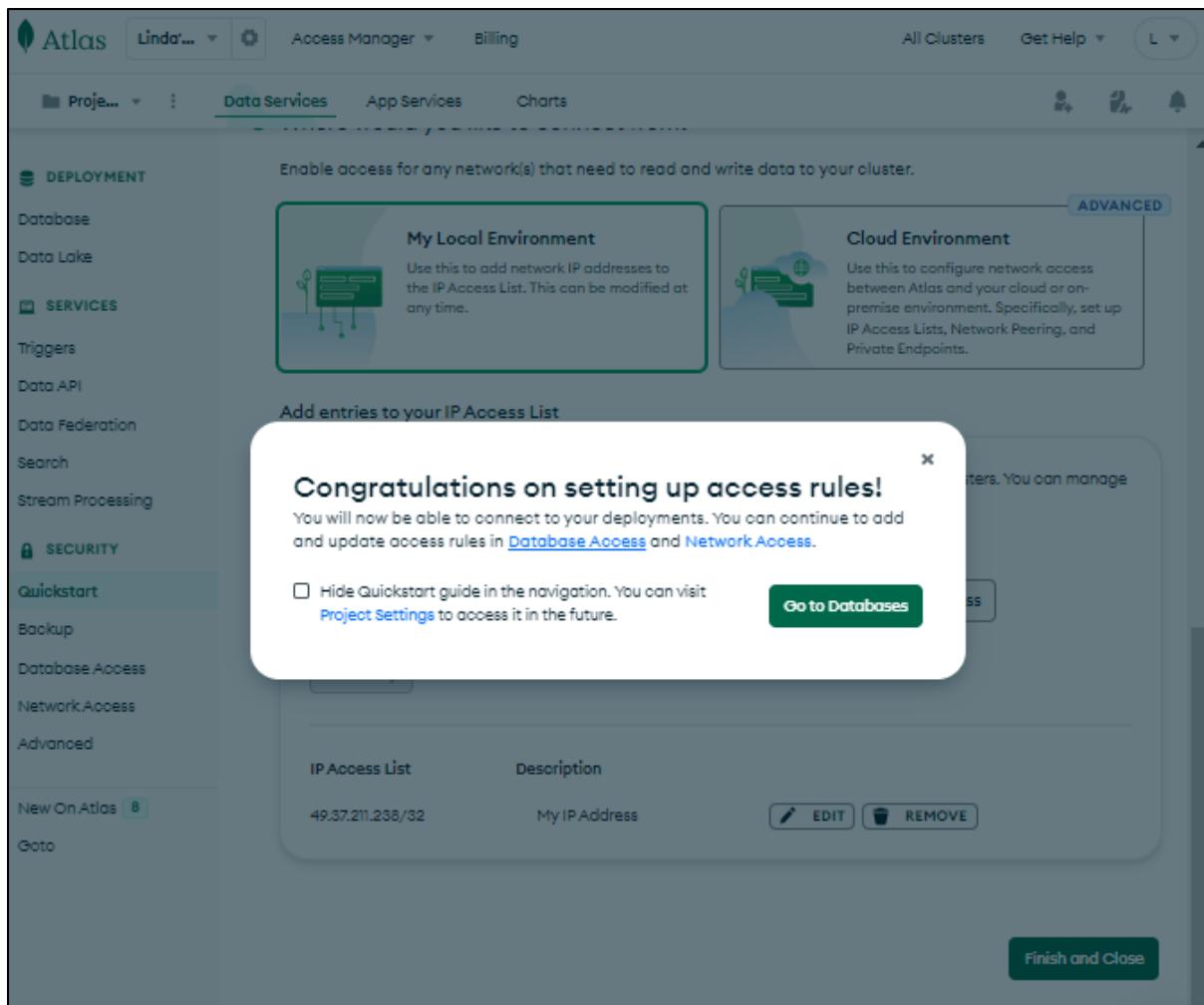


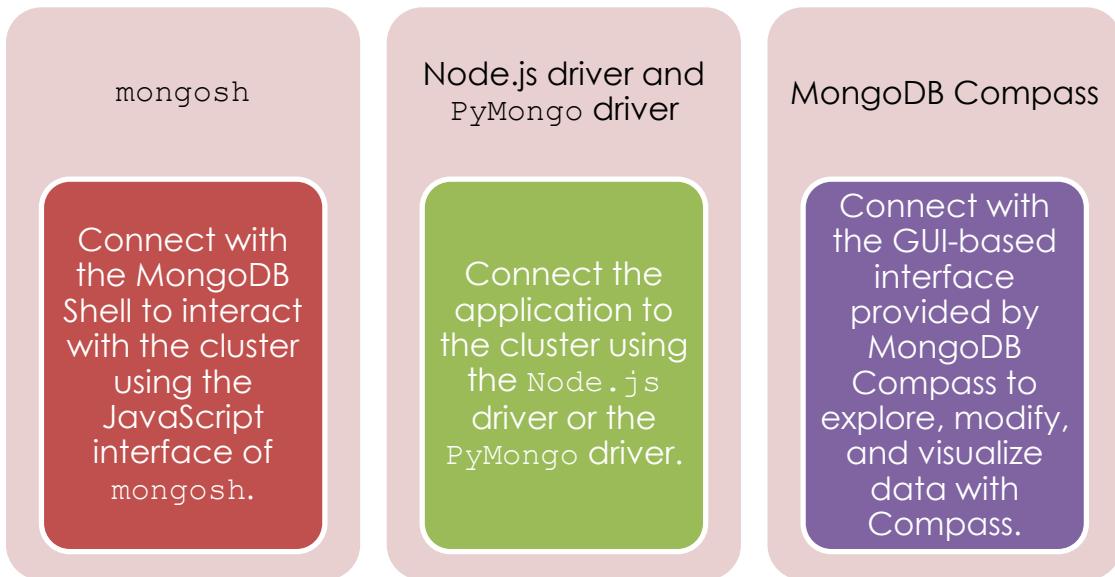
Figure 11.10: Cluster Provisioning Success Message Page

3. Click **Go to Databases**.

Cluster provisioning is completed.

11.1.5 Connect Mongo Shell to M0 Cluster

Finally, the user must connect to the cluster using one of these methods:



There are a few prerequisites for connecting to the cluster:

- An Atlas account must exist.
- An active cluster must be created in this account.
- An organization with a project must exist.
- An IP address must be added to the IP access list.
- A database user must exist on the cluster.

To connect to the M0 cluster using `mongosh`:

1. Click **Database** in the upper-left corner of the Atlas screen.
The **Database Deployments** page opens as shown in Figure 11.11.

LINDA'S ORG - 2023-06-30 > PROJECT 0

Database Deployments

Find a database deployment...

 Load sample datasets to Cluster0.
Atlas provides sample data you can load into your Atlas clusters. You can use this data to quickly get started exploring with data in MongoDB.

Cluster0 Connect View Monitoring Browse Collections ...

Enhance Your Experience
For better performance and full access to Atlas features, upgrade your cluster now!
 Upgrade
R: 0 W: 0 Last 3 hours 100.0%
Connections: 0 Last 3 hours 100.0%
In: 0 Out: 0 Last 3 hours 100.0%
VERSION REGION CLUSTER TIER TYPE BACKUPS LINKED APP SERVICES ATLAS SQL ATLAS SEARCH
6.0.6 AWS / N. Virginia (us-east-1) MO Sandbox (General) Replica Set - 3 nodes Inactive None Linked Connect Create Index
+ Add Tag

Figure 11.11: Database Deployments Page

2. To deploy the desired deployment, on this page, click **Connect** for **Cluster0**.

The **Choose a Connection Method** page opens as shown in Figure 11.12.

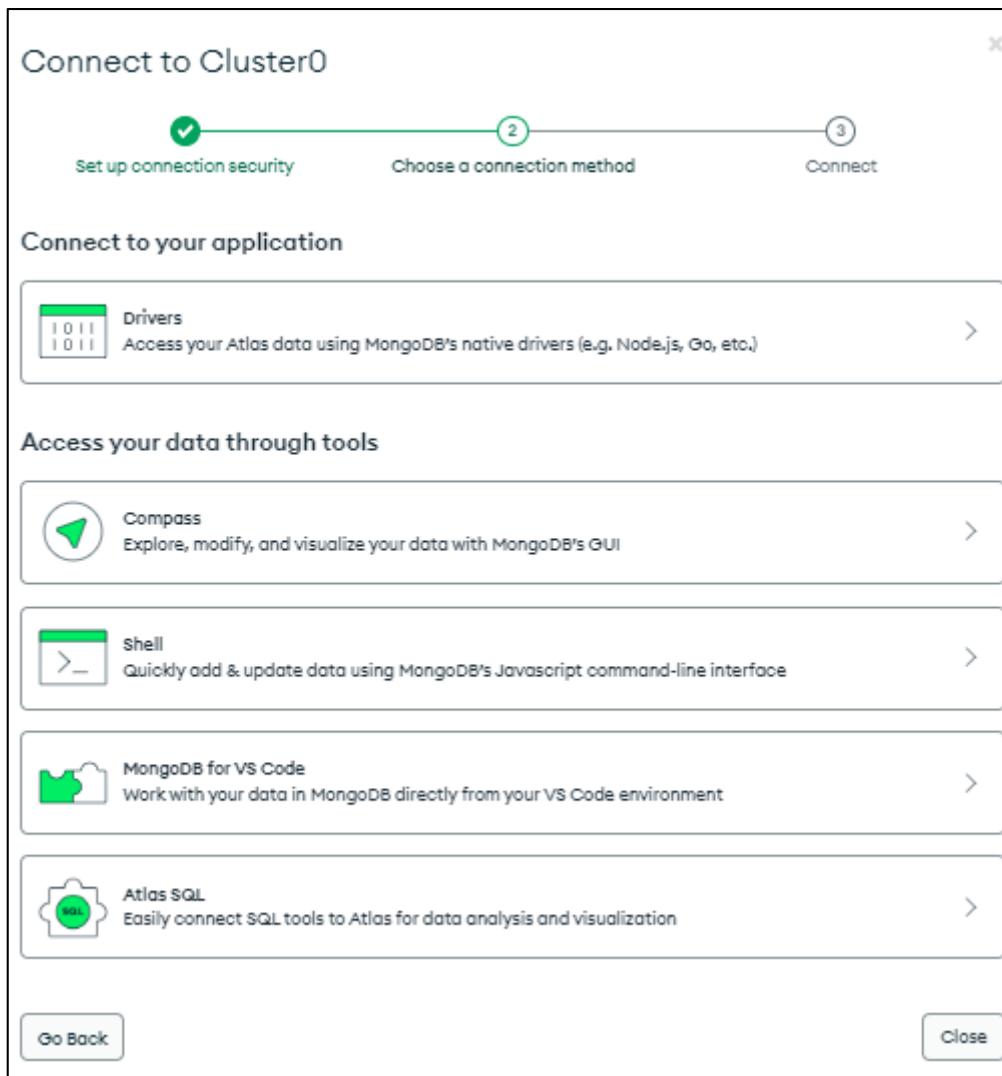


Figure 11.12: Choose a Connection Method Page

3. To use MongoDB Shell, on this page, under **Access your data through tools** section, click **Shell**.
The **Connect Page** opens as shown in Figure 11.13.
4. As the MongoDB shell is already installed, click **I have the MongoDB Shell installed**.

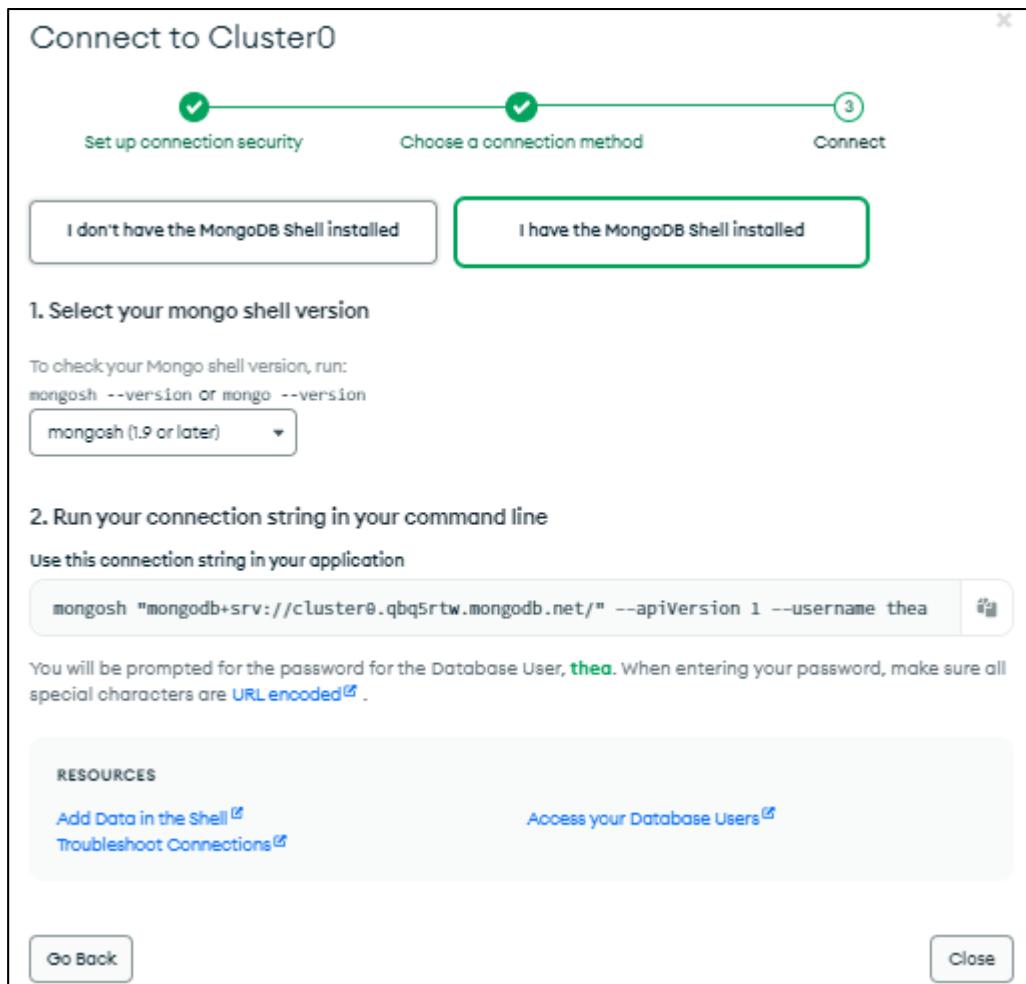


Figure 11.13: Connect Page

5. To use `mongosh`, under **Select your mongo shell version** section, from the drop-down list, select the `mongosh` version. The connection string is created automatically.
6. To use this connection string, from the **Run your connection string in your command line** box, copy the connection string and click **Close** to close the **Connect to Cluster0** dialog box.
7. Open the command prompt. Paste and run the copied connection string as shown in Figure 11.14.

```
C:\Users\Linda>mongosh "mongodb+srv://cluster0.qbq5rtw.mongodb.net/" --apiVersion 1 --username thea
Enter password: *****
```

Figure 11.14: Starting the Connection

When prompted, enter the password as specified when creating the Atlas database user.

The command is executed as shown in Figure 11.15.

```
C:\Users\Linda>mongosh "mongodb+srv://cluster0.qbq5rtw.mongodb.net/" --apiVersion 1 --username thea
Enter password: *****
Current Mongosh Log ID: 649ffbd0abfb793124fe7987
Connecting to:      mongodb+srv://<credentials>@cluster0.qbq5rtw.mongodb.net/?appName=mongosh+1.8.2
Using MongoDB:      6.0.6 (API Version 1)
Using Mongosh:      1.8.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

Atlas atlas-xihz70-shard-0 [primary] test> |
```

Figure 11.15: Connected to the Atlas Cluster

Now, cluster0 is connected within the mongosh. Keep this command prompt open.



In the same way, MongoDB Atlas can be connected with MongoDB Compass using the connection string.

11.2 Access MongoDB Atlas Cluster

After successfully setting up MongoDB, it is time to start using the database. Atlas provides sample data in a variety of formats, including JavaScript Object Notation (JSON), Comma-Separated Values (CSV), and Tab-Separated Values (TSV). As a beginner to MongoDB, sample data is a great way to learn about MongoDB and see how it can be used to store and manage data.

11.2.1 Load and View Sample Data in MongoDB Atlas

Currently, the cluster0 deployed has no databases or collections.

To load a sample dataset into cluster0:

1. If not there already, navigate to the **Database Deployments** page shown in Figure 11.11.
2. To load the available sample datasets, on this page, click **Load sample dataset**. A confirmation message appears when the sample dataset is loaded into cluster0 as shown in Figure 11.16.

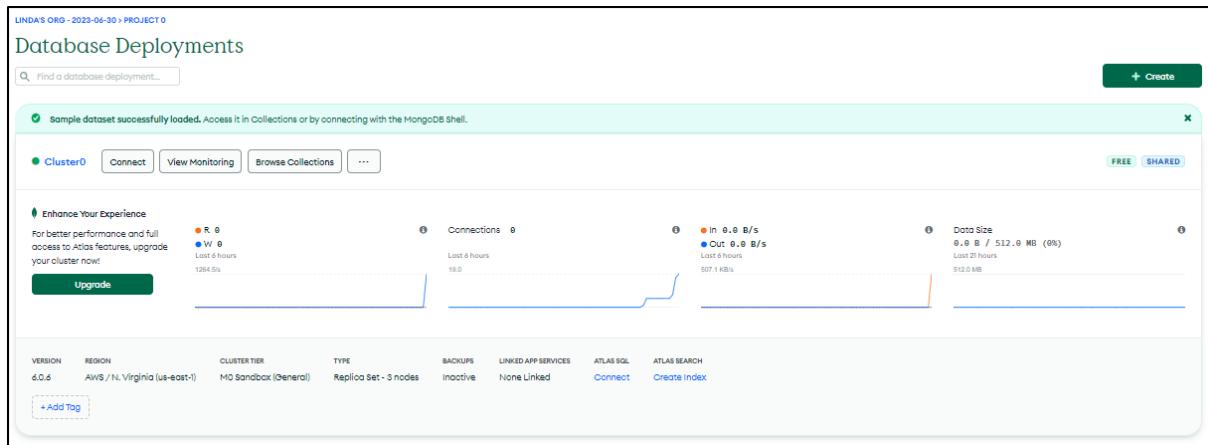


Figure 11.16: Loaded Sample Dataset Confirmation Message Page

3. To view the sample dataset, on this page, click **Browse Collections**.
The **Loaded Sample Dataset Lists** page opens as shown in Figure 11.17.

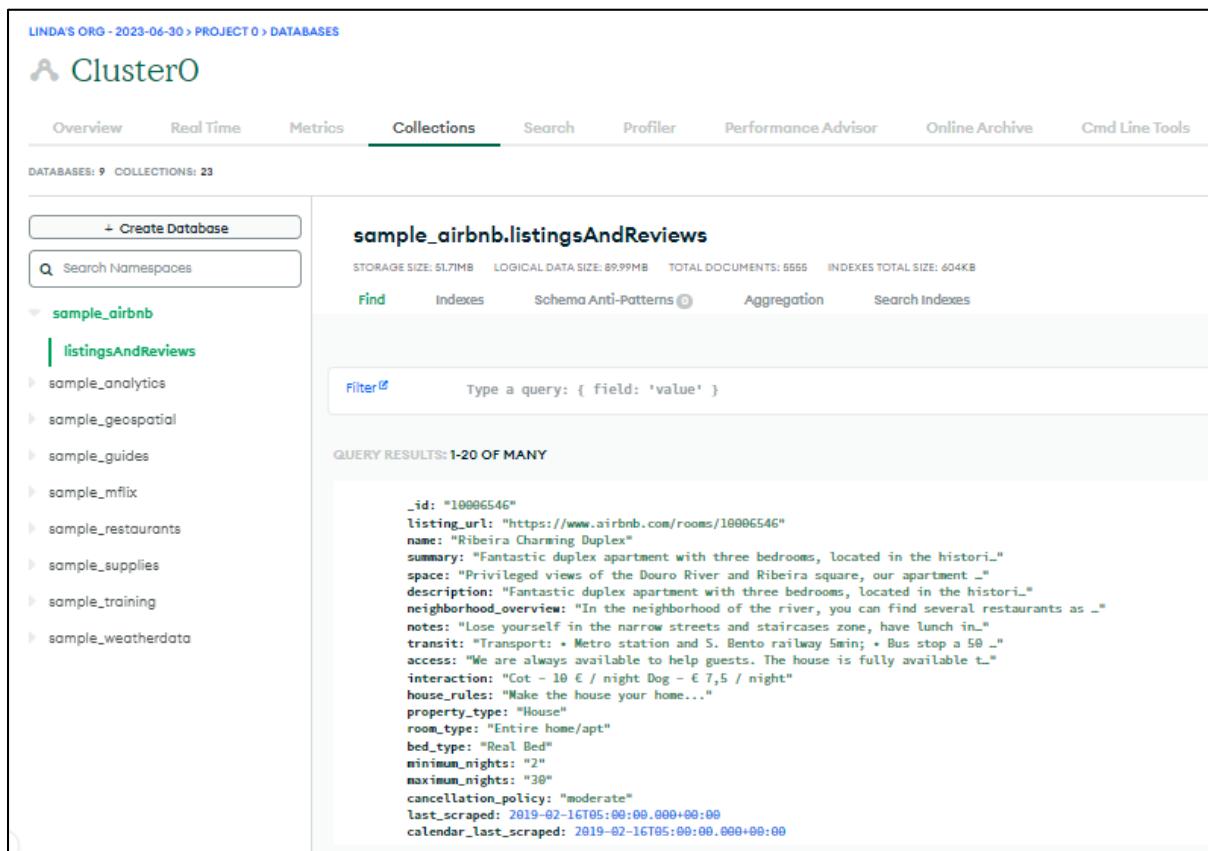


Figure 11.17: Loaded Sample Dataset Lists Page

11.2.2 View the Sample Data Using Mongo Shell

The loaded sample datasets can be viewed in Mongo Shell.

To view the databases loaded on cluster0 using Mongo Shell:

1. In the command prompt window that runs mongosh, execute the command as:

```
show dbs
```

All sample databases available on the MongoDB Atlas are displayed as shown in Figure 11.18.

```
Atlas atlas-xihz70-shard-0 [primary] test> show dbs
sample_airbnb      52.30 MiB
sample_analytics   9.14 MiB
sample_geospatial  1.35 MiB
sample_guides      40.00 KiB
sample_mflix       120.20 MiB
sample_restaurants 6.57 MiB
sample_supplies    1.12 MiB
sample_training    51.76 MiB
sample_weatherdata 2.75 MiB
admin              372.00 KiB
local              6.17 GiB
Atlas atlas-xihz70-shard-0 [primary] test> |
```

Figure 11.18: Sample Dataset Page

All Create, Read, Update, and Delete (CRUD) operations can be performed on the databases, and the changes will be reflected in the databases in cluster0.

11.2.3 Import Data From the Local System to MongoDB Atlas

Any database from the local system can be imported to MongoDB Atlas. This is very useful to migrate data from a previous database platform or to import backed up data. Let us import a database named sales which contains the product_sales collection from the C:\Sample_dataset folder to MongoDB Atlas.

To import data from a local system:

1. Navigate to the **Database Deployments** page and choose **Command Line Tools** for cluster0 as shown in Figure 11.19.

The screenshot shows the MongoDB Atlas Database Deployments page for cluster0. At the top, there's a success message: "Sample dataset successfully loaded. Access it in Collections or by connecting with the MongoDB Shell." Below this, there are buttons for "Connect", "View Monitoring", "Browse Collections", and a "..." menu. The "Cluster0" button is highlighted with a green dot. A dropdown menu from the "... menu" is open, showing options: "Edit Configuration", "Command Line Tools" (which is selected and highlighted in grey), "Load Sample Dataset", and "Terminate". There's also a "Connections" section showing 0 connections over the last 6 hours. On the left, there's an "Enhance Your Experience" section with a "Upgrade" button. In the center, there are two line charts: one for "Latency (ms)" and one for "Throughput (ops/s)". At the bottom, there's a table with cluster details and a "+ Add Tag" button.

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED APP SERVICES	ATLAS SQL
6.0.6	AWS / N. Virginia (us-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Connect

Figure 11.19: Database Deployments Page

The **Cmd Line Tools** page opens as shown in Figure 11.20.

The screenshot shows the MongoDB Atlas interface for Cluster0. The top navigation bar includes links for Overview, Real Time, Metrics, Collections, Search, Profiler, Performance Advisor, Online Archive, and Cmd Line Tools (which is underlined, indicating it's the active tab). Below the navigation, there are three main sections: 'Connect To Your Cluster', 'Manage Your Cluster From the Command Line Interface', and 'Manage Your Cluster From the Command Line'. Each section contains descriptive text and a 'Install' button. The 'Binary Import and Export Tools' section at the bottom provides examples of mongorestore and mongodump commands with their respective parameters.

LINDA'S ORG - 2023-06-30 > PROJECT 0 > DATABASES

Cluster0

Overview Real Time Metrics Collections Search Profiler Performance Advisor Online Archive **Cmd Line Tools**

Connect To Your Cluster

Methods to connect your application to your cluster via MongoShell, URI, or Compass can be found in the connect modal.

[Connect Instructions](#)

Manage Your Cluster From the Command Line Interface

Create and manage MongoDB Atlas resources from your command line and easily automate them using scripts. [Learn more](#)

[Install Atlas CLI](#)

Manage Your Cluster From the Command Line

Use command line utilities to import and export data, restore backups, and view diagnostics

[Install MongoDB Database Tools](#)

Binary Import and Export Tools

Replace `PASSWORD` with the password for the admin user and `DATABASE` with the name of the database you wish to import/export to your cluster.

`mongorestore` | creates a new database or adds data to an existing database. By default, `mongorestore` reads the database dump in the `dump` sub-directory of the current directory; to restore from a different directory, pass in the path to the directory as a final argument.

```
mongorestore --uri mongodb+srv://thea:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/
```

`mongodump` | creates a binary export of the contents of a database

```
mongodump --uri mongodb+srv://thea:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/<DATABASE>
```

Figure 11.20: Cmd Line Tools Page

2. Scroll down to the **Data Import and Export Tools** section of the page.

The **Data Import and Export Tools** section appears as shown in Figure 11.21.

Data Import and Export Tools

Replace **PASSWORD** with the password for the admin user, **DATABASE** with the name of the database you wish to import/export to your cluster, and **COLLECTION** with the name of the collection you wish to import/export to your cluster. Replace **FILETYPE** with "json" or "csv" to specify the file type. Where applicable, replace **FILENAME** with the location and name of the output file (for export) or data source (for import).

NOTE: When exporting or importing CSV data, an additional --fields flag is often required. See documentation for the specific tool for additional details.

[mongoimport](#) | imports content from an Extended JSON, CSV, or TSV export

```
mongoimport --uri mongodb+srv://thea:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/<DATABASE> --collection <COLLECTION> --type <FILETYPE> --file <FILENAME>
```

[mongoexport](#) | produces a JSON or CSV export of data stored in a MongoDB instance

```
mongoexport --uri mongodb+srv://thea:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/<DATABASE> --collection <COLLECTION> --type <FILETYPE> --out <FILENAME>
```

Figure 11.21: Data Import and Export Tools Section

3. From the **Data Import and Export Tools** section, copy the mongoimport connection string template to the clipboard.

```
mongoimport --uri  
mongodb+srv://<USERNAME>:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/<DATABASE> --collection <COLLECTION> --type  
<FILETYPE> --file <FILENAME>
```

The connection string template includes placeholder values for certain options. These placeholders must be replaced with the appropriate values for the Atlas cluster as described in Table 11.1.

Placeholder	Values
<PASSWORD>	<p>Replace with the password for the user specified in --username</p> <p>The template includes the database user for the project as the --username. To authenticate as a different user, replace the value of --username and specify the password for that user in --password.</p> <p>Enclose the password within double or single quotes. For example, if the password is @bc123, it must be surrounded within quotes, such as "@bc123".</p>

Placeholder	Values
<DATABASE>	Replace with the name of the database to which data is being imported.
<COLLECTION>	Replace with the name of the collection to which data is being imported.
<FILETYPE>	Replace with the file type of the data source from which data is being imported.
<FILENAME>	Replace with the name of the data source from which data is being imported.

Table 11.1: Connection String Template Values

Table 11.1 does not include <USERNAME> as the connection string will automatically contain the username with which you logged in to Atlas.

4. To import the sales database with product_sales collection, open a new command prompt.
5. Run the connection string command as:

```
mongoimport --uri
mongodb+srv://thea:tea99@cluster0.qbq5rtw.mongodb.net/sales
--collection product_sales --type json --file
C:\Sample_dataset\sales\product_sales.json
```

The command executes as shown in Figure 11.22.

```
C:\Users\Linda>mongoimport --uri mongodb+srv://thea:tea99@cluster0.qbq5rtw.
mongodb.net/sales --collection product_sales --type json --file C:\Sample_d
ataset\sales\product_sales.json
2023-07-01T21:58:14.135+0530      connected to: mongodb+srv://[**REDACTED**]@
cluster0.qbq5rtw.mongodb.net/sales
2023-07-01T21:58:14.712+0530      3 document(s) imported successfully. 0 docu
ment(s) failed to import.
```

Figure 11.22: Imported Sales Data

A message shows that the three documents from the local system are imported to the MongoDB Atlas successfully.

6. To view the imported sales database, open the command prompt window that runs mongosh and execute the command as:

```
show dbs
```

The sales database is displayed as shown in Figure 11.23.

```
Atlas atlas-xihz70-shard-0 [primary] test> show dbs
sales          40.00 KiB
sample_airbnb   52.36 MiB
sample_analytics 9.14 MiB
sample_geospatial 1.35 MiB
sample_guides    40.00 KiB
sample_mflix     120.28 MiB
sample_restaurants 6.57 MiB
sample_supplies   1.12 MiB
sample_training   51.78 MiB
sample_weatherdata 2.75 MiB
admin           372.00 KiB
local            6.17 GiB
Atlas atlas-xihz70-shard-0 [primary] test> |
```

Figure 11.23: Imported Sales Database

7. To view more details, switch to sales database by executing the command:

```
use sales
```

The command is executed as shown in Figure 11.24.

```
Atlas atlas-xihz70-shard-0 [primary] test> use sales
switched to db sales
Atlas atlas-xihz70-shard-0 [primary] sales>
```

Figure 11.24: Switched to Sales Database

8. To view the documents of product_sales collection, execute the command as:

```
db.product_sales.find()
```

The command is executed as shown in Figure 11.25.

```
Atlas atlas-xihz70-shard-0 [primary] sales> db.product_sales.find()
[
  {
    _id: ObjectId("64a0541e43840751334bbc80"),
    product_id: 338,
    sales_person: 'Amelia',
    products_sold: [ 4, 8, 9, 6 ]
  },
  {
    _id: ObjectId("64a0541e43840751334bbc81"),
    product_id: 302,
    sales_person: 'Smith',
    products_sold: [ 5, 3, 7, 2 ]
  },
  {
    _id: ObjectId("64a0541e43840751334bbc82"),
    product_id: 371,
    sales_person: 'Robert',
    products_sold: [ 3, 8, 6, 5 ]
  }
]
Atlas atlas-xihz70-shard-0 [primary] sales> |
```

Figure 11.25: product_sales Collection Database

11.2.4 Export Data from Atlas Cluster to the Local System

Data from the Atlas cluster can also be exported to the local system for backup, offline analysis, sharing, or troubleshooting. Let us export the sample_airbnb database from MongoDB Atlas to the C:\Sample_dataset folder for offline analysis.

To export data to the local system:

1. Navigate to the **Database Deployments** page.
2. For **Cluster0**, click the three dots, and then click **Command Line Tools**.
3. Scroll down to the **Data Import and Export Tools** section as shown in Figure 11.26.

Data Import and Export Tools

Replace `PASSWORD` with the password for the admin user, `DATABASE` with the name of the database you wish to import/export to your cluster, and `COLLECTION` with the name of the collection you wish to import/export to your cluster. Replace `FILETYPE` with "json" or "csv" to specify the file type. Where applicable, replace `FILENAME` with the location and name of the output file (for export) or data source (for import).

NOTE: When exporting or importing CSV data, an additional `--fields` flag is often required. See documentation for the specific tool for additional details.

`mongoimport` | imports content from an Extended JSON, CSV, or TSV export

```
mongoimport --uri mongodb+srv://thea:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/<DATABASE> --collection <COLLECTION> --type <FILETYPE> --file <FILENAME>
```

`mongoexport` | produces a JSON or CSV export of data stored in a MongoDB instance

```
mongoexport --uri mongodb+srv://thea:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/<DATABASE> --collection <COLLECTION> --type <FILETYPE> --out <FILENAME>
```

Figure 11.26: Data Import and Export Tools Section

1. From the **Data Import and Export Tools** section, copy the `mongoexport` connection string to the clipboard.

```
mongoexport --uri  
mongodb+srv://thea:<PASSWORD>@cluster0.qbq5rtw.mongodb.net/<DATABASE> --collection <COLLECTION> --type <FILETYPE> --  
out <FILENAME>
```

The connection string template includes placeholder values for certain options. These placeholders must be replaced with the appropriate values for the Atlas cluster as described in Table 11.1.

2. To export the `sample_airbnb` database with `listingsAndReviews` collection, open a new command prompt and execute the command:

```
mongoexport --uri  
mongodb+srv://thea:tea99@cluster0.qbq5rtw.mongodb.net/sample_airbnb --collection listingsAndReviews --type json --out  
C:\Sample_dataset\airbnb_listings.json
```

The command executes as shown in Figure 11.27.

```
C:\Users\Linda>mongoexport --uri mongodb+srv://thea:tea99@cluster0.qbq5rtw.mongodb.net/sample_airbnb --collection listingsAndReviews --type json --out C:\Sample_dataset\airbnb_listings.json
2023-07-01T22:24:32.076+0530      connected to: mongodb+srv://[**REDACTED**]@cluster0.qbq5rtw.mongodb.net/sample_airbnb
2023-07-01T22:24:33.303+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:34.305+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:35.297+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:36.298+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:37.304+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:38.292+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:39.302+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:40.303+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:41.305+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:42.299+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:43.298+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:44.302+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:45.304+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:46.295+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:47.293+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:48.299+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:49.292+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:50.292+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:51.305+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:52.297+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:53.293+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:54.292+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:55.292+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:56.294+0530      [.....] sample_airbnb.listingsAndReviews 0/5555 (0.0%)
2023-07-01T22:24:57.244+0530      [#####] sample_airbnb.listingsAndReviews 5555/5555 (100.0%)
2023-07-01T22:24:57.244+0530      exported 5555 records
```

Figure 11.27: Exported sample_airbnb Database

A total of 5555 records from the sample_airbnb database is exported into the local system successfully. The file will be displayed in the local system as shown in Figure 11.28.

Name	Date modified	Type
sales	7/1/2023 7:30 PM	File folder
sample_analytics	5/30/2023 11:14 AM	File folder
sample_geospatial	5/30/2023 11:14 AM	File folder
sample_guides	5/30/2023 11:14 AM	File folder
sample_mflix	5/30/2023 11:14 AM	File folder
sample_restaurants	5/30/2023 11:14 AM	File folder
sample_supplies	5/30/2023 11:14 AM	File folder
sample_training	5/30/2023 11:14 AM	File folder
sample_weatherdata	5/30/2023 11:14 AM	File folder
airbnb_listings	7/1/2023 10:24 PM	JSON Source File

Figure 11.28: Exported `airbnb_listings.json` in the Local System

11.3 Administering MongoDB Cluster

MongoDB clusters are complex systems that require regular administration to ensure that they are running smoothly and efficiently. The clusters can be:

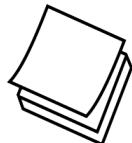
Paused	Resumed	Terminated
<ul style="list-style-type: none"> Temporarily disable the cluster and prevent users from accessing the data Useful for maintenance or troubleshooting purposes 	<ul style="list-style-type: none"> Bring the cluster back online and make the data available to users 	<ul style="list-style-type: none"> Permanently delete the cluster and all of its data Should only be done as a last resort, such as when the cluster is no longer required or when there is a serious problem with the cluster

11.3.1 Pause a Cluster

MongoDB Atlas automatically pauses all inactive M0, M2, and M5 clusters after 60 days of inactivity. For clusters other than M0, M2, and M5 type, users can manually initiate a pause.

When a cluster is paused in Atlas, the user cannot modify the cluster configuration and cannot read or write data to the cluster. Atlas stops collecting monitoring information for these clusters completely and does not allow any connections to the cluster until the cluster is resumed. Configured alerts will no longer trigger. All backups will be stopped, but any existing snapshots will be retained until they expire.

Prior to pausing, Atlas sends an email notification seven days in advance and another email after the pause.



Atlas does not allow the user to initiate a pause on M0 cluster. However, an automatically paused M0 cluster can be resumed or terminated at any time.

Backup Compliance Policy

A Backup Compliance Policy is a set of rules that govern how backups are created and managed for MongoDB Atlas clusters. This policy specifies details such as frequency of backups, the retention period for backups, and the encryption used for backups.



If a paused cluster does not have the **Encryption at Rest** option enabled, the **Require Encryption at Rest using Customer Key Management for all clusters** option cannot be toggled to **ON** in a Backup Compliance Policy.

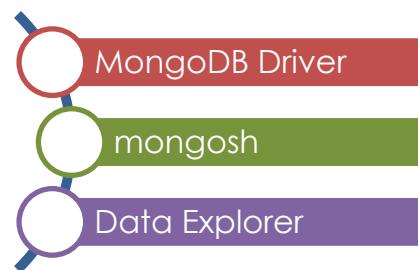
In general, pausing a cluster will NOT affect the existing backups that have been created for the cluster. However, if the cluster is resumed and the Backup Compliance Policy has the Continuous Cloud Backup option enabled, then Atlas will create a new backup for the cluster.

11.3.2 Resume a Cluster with Backup Compliance Policy

When **Backup Compliance Policy** is enabled, resuming a cluster in Atlas automatically enables Cloud backup. If the **Require Point in Time Restore to all clusters** option is set to **ON**, Atlas will enable Continuous Cloud Backup and adjust the restore window according to the Backup Compliance Policy.

Additionally, Atlas will automatically modify the backup to meet the minimum requirements specified by the Backup Compliance Policy.

To resume collecting monitoring information for an Atlas M0 cluster that has been paused for monitoring, connections can be made to the cluster using one of the three options:



To resume a paused cluster, navigate to the **Database Deployments** page and choose **Resume** for the desired cluster. Note that this option will be enabled only when the cluster is paused.

11.3.3 Terminate a Cluster

To terminate a cluster:

1. Navigate to the **Database Deployments** page and choose **Terminate** for the desired cluster as shown in Figure 11.29.

Database Deployments

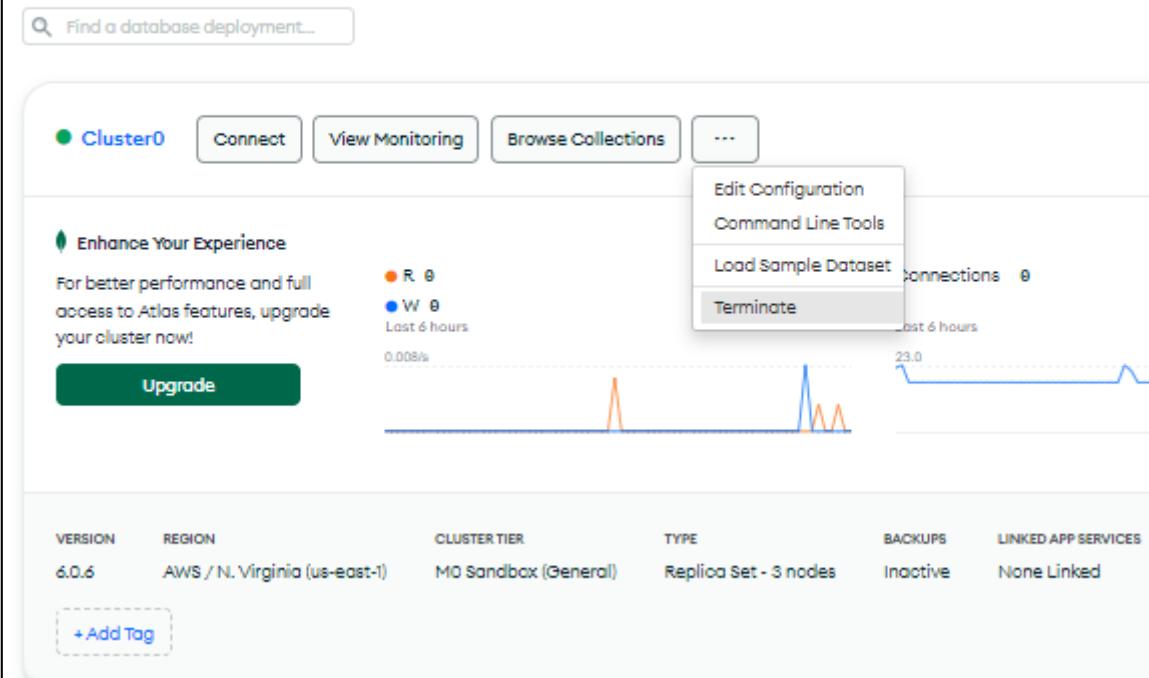


Figure 11.29: Database Deployments Page



Terminating a cluster will also delete all backup snapshots for that cluster.

11.4 Summary

- To get started with MongoDB Atlas, begin to create an Atlas account, set up a cluster and then, create a user for that cluster.
- MongoDB Atlas provides sample datasets to load into the Atlas database deployments.
- A dataset can be imported from the local system into the MongoDB Atlas using the `mongoimport` command.
- A dataset can be exported to the local system from the MongoDB Atlas Database using the `mongoexport` command.
- As part of administration tasks, the MongoDB clusters can be paused, resumed, and terminated.

Test Your Knowledge

1. Which of the following connection methods does MongoDB Atlas provide to connect your clusters with deployment?
 - a. Connect via Compass
 - b. Connect via mongosh
 - c. Connect via Application
 - d. Connect via VS Code
2. Which of the following service providers are supported by Atlas for M0-free clusters?
 - a. Amazon Web Services
 - b. Google Cloud Platform
 - c. IBM Cloud
 - d. Microsoft Azure
3. Which of the following cluster is a free cluster used for learning and exploring MongoDB in a cloud environment?
 - a. M10
 - b. Serverless
 - c. M0
 - d. M9
4. Which of the following statements is not true about database users for clusters in MongoDB Atlas?
 - a. It is necessary to create a database user to access the cluster.
 - b. Atlas allows you to create a database user only if you are an **Organization Owner** or a **Project Owner**.
 - c. Database users are separate from Atlas users.
 - d. Atlas users can access MongoDB databases.
5. Which of the following statements is true about pausing a cluster in MongoDB Atlas?
 - a. It is not possible to initiate a pause for M0 clusters.
 - b. It is not possible to resume or terminate an automatically paused cluster at any time.
 - c. After 60 days of inactivity in a cluster, Atlas automatically pauses that cluster.
 - d. It is possible to pause M10 or larger clusters.

Answers to Test Your Knowledge

1	a, b, c, d
2	a, b, d
3	C
4	D
5	C

Try it Yourself

1. In MongoDB Atlas, perform the given tasks:
 - a. Create an Atlas account.
 - b. Deploy a free M0 cluster.
 - c. Add the current IP address to the cluster IP access list.
 - d. Create a database user for your cluster.
 - e. Connect MongoDB mongosh to your Atlas cluster using the connection string.
2. Load the sample dataset in MongoDB Atlas cluster and view the loaded datasets.
3. Create a database named Customer_detail and a collection named Customer_personal in your local system. The documents in the collection are:

```
[  
  {  
    Customer_id:94608  
    Customer_name:Richard David  
    Address:5858 Horton Street, USA  
    Gender:Male  
  },  
  {  
    Customer_id:94590  
    Customer_name:Rita Edward  
    Address:401 Quarry Road, Stanford,USA  
    Gender:Female  
  },  
  {  
    Customer_id:94305  
    Customer_name:Adam Smith  
    Address:#170, Emeryville valley, USA  
    Gender:Male  
  }  
]
```

4. Import the Customer_detail database and the Customer_personal collection from the local system to the MongoDB Atlas cluster.
5. Export the sample_mflix database which is loaded as part of the sample dataset from the MongoDB Atlas cluster to your Local system.



SESSION 12

MONGODB DATABASE CONNECTIVITY WITH PYTHON

Learning Objectives

In this session, students will learn to:

- Describe how to connect Python with MongoDB
- Explain how to install PyMongo Driver
- Describe how to create a database and collection in MongoDB using Python
- Describe the ways to query, sort, update, and delete documents in MongoDB using Python

Python is a popular computer programming language used to automate processes, analyze data, and create Websites. It is a general-purpose programming language that can be used in various application domains. Python can be integrated with MongoDB to create robust and versatile database applications. These database applications use Python as the application interface and MongoDB as the database server.

This session will provide an overview of connecting Python with MongoDB. It will also explain how to install the PyMongo Driver. This session will explore the methods to create a database and collection in MongoDB using Python. It will also explain the methods to query, sort, update, and delete documents in MongoDB using Python.

12.1 Overview of Connecting Python with MongoDB

As discussed in the previous sessions, MongoDB is a popular choice to build extremely scalable and reliable databases. Python is a versatile programming language with an extensive standard library. Therefore, combination of Python and MongoDB has proven to be excellent for developing database applications.

PyMongo is MongoDB's official Python driver. PyMongo helps to create a connection between Python and MongoDB. It also provides a wide range of methods that can be used to insert, query, update, and delete data from the database. The documents retrieved using PyMongo can be accessed using data structures in Python.

12.2 Install PyMongo Driver

Before installing PyMongo, users must download and install the latest version of Python from the URL: <https://www.python.org/downloads/>

To install PyMongo:

1. Open the Command Prompt.
2. Navigate to the path where Python is installed on the local system.
3. Execute the command as:

```
pip install pymongo
```

The command is executed as shown in Figure 12.1.

```
c:\>pip install pymongo
Collecting pymongo
  Downloading pymongo-4.4.0-cp311-cp311-win_amd64.whl (453 kB)
    ━━━━━━━━━━━━━━━━ 453.6/453.6 kB 2.8 MB/s eta 0:00:00
Collecting dnspython<3.0.0,>=1.16.0 (from pymongo)
  Downloading dnspython-2.3.0-py3-none-any.whl (283 kB)
    ━━━━━━━━━━━━━━ 283.7/283.7 kB 17.1 MB/s eta 0:00:00
Installing collected packages: dnspython, pymongo
Successfully installed dnspython-2.3.0 pymongo-4.4.0
```

Figure 12.1: Install PyMongo

The PyMongo driver is now installed. The user can now use the driver to connect Python with MongoDB.

12.3 Create a Database and Collection Using Python

To create a database and collection in MongoDB using Python, the user must write Python code for:

1. Establishing a connection with MongoDB
2. Creating a database in MongoDB
3. Creating a collection in the database
4. Inserting documents into the collection

This code must be entered in a .py(Python) file and executed.

To facilitate connection to MongoDB from Python, PyMongo includes the MongoClient class. This class acts as a client to the MongoDB server. To connect to a MongoDB database, users must first create a MongoClient instance by importing it. The command to import MongoClient is:

```
from pymongo import MongoClient  
client = MongoClient()
```

The command `client = MongoClient()` will create a connection to the default host and default port.

Now, the output from Python applications when used with large databases such as MongoDB can be huge. Also, MongoDB provides the output data in JavaScript Object Notation (JSON) format, which can be difficult to read. To make the output readable, Python offers the `pprint` module. `pprint` stands for 'pretty print'. So, the next step is to import `pprint` using the command as:

```
import pprint
```

Consider that the user wants to create a database named `library`. To do so, the user can use the command as:

```
db = client.library
```

Consider that the user wants to create a collection named `library_details`. To do so, the user can use the command as:

```
lib_collection = db.library_details
```

Consider that the user wants to insert some documents into this collection. Code Snippet 1 lists the code to perform this action:

Code Snippet 1:

```
lib = [
    {"book_id": "1010", "book_name": "Python
Programming", "book_author": "John M Zelle",
"volume":3},
    {"book_id": "1019", "book_name": "Python for Data
Analysis", "book_author": "Wes Mckinney",
"volume":1},
    {"book_id": "1009", "book_name": "Python
Cookbook", "book_author": "David Beazley",
"volume":3}
]
library_details.insert_many(lib)
pprint.pprint("Documents Inserted successfully")
```

This code will insert three documents into the `library_details` collection. The `pprint` function in this code will print the message on the screen indicating successful insertion of documents.

As discussed earlier, the code to establish the connection, create a database, create a collection, and insert documents into the collection must be placed in a Python file. For example, let us enter the code into a file named `python_library.py` and save the file to the `C:\Python` folder.

To execute the code in the `python_library.py` file, at the command prompt, run the command as:

```
python C:\Python\python_library.py
```

The command executes as shown in Figure 12.2.

```
C:\Users\linda>python C:\Python\python_library.py
'Documents Inserted successfully'
```

Figure 12.2: Execution of Python File

To view the created database, connect to Mongo Shell and execute the command as:

```
show dbs
```

The command executes as shown in Figure 12.3.

```
test> show dbs
admin              232.00 KiB
books              72.00 KiB
config              72.00 KiB
empl_det            40.00 KiB
library              40.00 KiB
local              72.00 KiB
sample_analytics     9.45 MiB
sample_supplies      968.00 KiB
sample_training       8.39 MiB
sample_weatherdata    4.55 MiB
student_detail        40.00 KiB
```

Figure 12.3: Show Databases

To verify if the `library_details` collection exists under the `library` database, switch to the `library` database and execute the command as:

```
show collections
```

The command is executed as shown in Figure 12.4.

```
test> use library
switched to db library
library> show collections
library_details
```

Figure 12.4: View Collections in the Database

12.4 Query Documents Using Python

Users can also write Python code to search for documents from a collection, filter the search results, sort the search results, update a document, or delete a document.

12.4.1 Find a Document

Consider that the user wants to view the first document in the library_details collection. Code Snippet 2 lists the code to perform this action:

Code Snippet 2:

```
from pymongo import MongoClient
import pprint
client = MongoClient()
db = client.library
library_details = db.library_details
pprint.pprint(library_details.find_one())
```

This code must be saved in a Python file. For example, let us enter this code in a file named as python_libraryFind.py and save it to the C:\Python folder. To execute the code in the python_libraryFind.py folder, at the command prompt, run the command as:

```
python C:\Python\python_libraryFind.py
```

The command executes as shown in Figure 12.5.

```
PS C:\Users\linda> python C:\Python\python_libraryFind.py
{'_id': ObjectId('64a28b6a8985b8c5897f682b'),
 'book_author': 'John M Zelle',
 'book_id': '1010',
 'book_name': 'Python Programming',
 'volume': 3}
```

Figure 12.5: View a Document in the Collection

This command has fetched the first document in the library_details collection.

12.4.2 Find All Documents

Consider that the user wants to view all the documents in the library_details collection. Code Snippet 3 lists the code to perform this action:

Code Snippet 3:

```
from pymongo import MongoClient
import pprint
client = MongoClient()
db = client.library
library_details = db.library_details
for lib in library_details.find({}):
    print(lib)
```

The `for` statement in this code will iterate through all the documents and print those on the screen.

This code must be saved in a Python file. For example, let us enter the code in a file named as `python_libraryFindAll.py` and save it to the `C:\Python` folder. To execute the code in the `python_libraryFindAll.py` file, at the command prompt, run the command as:

```
python C:\Python\python_libraryFindAll.py
```

The command executes as shown in Figure 12.6.

```
PS C:\Users\linda> python C:\Python\python_libraryFindAll.py
{'_id': ObjectId('64a28b6a8985b8c5897f682b'), 'book_id': '1010', 'book_name': 'Python Programming', 'book_author': 'John M Zelle', 'volume': 3}
{'_id': ObjectId('64a28b6a8985b8c5897f682c'), 'book_id': '1019', 'book_name': 'Python for Data Analysis', 'book_author': 'Wes McKinney', 'volume': 1}
{'_id': ObjectId('64a28b6a8985b8c5897f682d'), 'book_id': '1009', 'book_name': 'Python Cookbook', 'book_author': 'David Beazley', 'volume': 3}
```

Figure 12.6: View All Documents in the Collection

This code displays all the documents in the `library_details` collection.

12.4.3 Filter the Result

Consider that the user wants to filter the output of a `find` method to view book details of a specific `book_author`. PyMongo provides a query object which can be used to specify the criteria to filter the output of a search. The query object is passed as the first argument to the `find` method. Code Snippet 4 lists the code to perform this action:

Code Snippet 4:

```
from pymongo import MongoClient
import pprint
client = MongoClient()
db = client.library
library_details = db.library_details
for lib in library_details.find({"book_author": "Wes Mckinney"}):
    print(lib)
```

This code must be saved in a Python file. For example, let us enter the code in a file named as `python_libraryQuery.py` and save it to the `C:\Python` folder. To execute the code in the `python_libraryQuery.py` folder, at the command prompt, run the command as:

```
python C:\Python\python_libraryQuery.py
```

The command executes as shown in Figure 12.7.

```
PS C:\Users\linda> python C:\Python\python_libraryQuery.py
{'_id': ObjectId('64a28b6a8985b8c5897f682c'), 'book_id': '1019', 'book_name': 'Python for Data Analysis', 'book_author': 'Wes Mckinney', 'volume': 1}
```

Figure 12.7: Filter the Result

12.4.4 Filter Query Results Using Comparison Operators

Consider that the user wants to fetch all the documents in the library_details collection where the volume field is greater than 1. This can be achieved using the '\$gt' comparison operator. Code Snippet 5 lists the code to perform this action:

Code Snippet 5:

```
from pymongo import MongoClient
import pprint
client = MongoClient()
db = client.library
library_details = db.library_details
for lib in
library_details.find({"volume": {"$gt": 1}}):
print(lib)
```

This code must be saved in a Python file. For example, let us enter the code in a file named as python_libraryOperator.py and save it to the C:\Python folder. To execute the code in the python_libraryOperator.py file, at the command prompt, run the command as:

```
python C:\Python\python_libraryOperator.py
```

The command executes as shown in Figure 12.8.

```
PS C:\Users\linda> python C:\Python\python_libraryOperator.py
{'_id': ObjectId('64a28b6a8985b8c5897f682b'), 'book_id': '1010', 'book_name': 'Python Programming', 'book_author': 'John M Zelle', 'volume': 3}
{'_id': ObjectId('64a28b6a8985b8c5897f682d'), 'book_id': '1009', 'book_name': 'Python Cookbook', 'book_author': 'David Beazley', 'volume': 3}
```

Figure 12.8: Comparison Operator

Note that this code has fetched only those documents where value in the volume field is greater than 1.

12.4.5 Filter Query Results Using Boolean Operators

Consider that the user wants to fetch all the documents in the library_details collection where the volume field is greater than 1 and the

`book_name` is 'Python Programming'. This can be achieved using the `and` Boolean operator. Code Snippet 6 lists the code to perform this action:

Code Snippet 6:

```
client = MongoClient()
db = client.library
library_details = db.library_details
for lib in
    library_details.find({"$and": [{"book_name": "Python Programming"}, {"volume": {"$gt": 1}}]}):
    print(lib)
```

This code must be saved in a Python file. For example, let us enter the code in a file named as `python_libraryAndOperator.py` and save it to the `C:\Python` folder. To execute the code in the `python_libraryAndOperator.py` file, at the command prompt, run the command as:

```
python C:\Python\python_libraryAndOperator.py
```

The command executes as shown in Figure 12.9.

```
PS C:\Users\linda> python C:\Python\python_libraryAndOperator.py
{'_id': ObjectId('64a28b6a8985b8c5897f682b'), 'book_id': '1010', 'book_name': 'Python Programming', 'book_author': 'John M Zelle', 'volume': 3}
```

Figure 12.9: Boolean Operator

Note that this code has fetched only those documents from the `library_details` collection where the `volume` field is greater than 1 and the `book_name` is 'Python Programming'.

12.4.6 Sort the Query Results

Consider that the user wants to sort the documents fetched from a `find` method in descending order on the `book_name` field. This can be done using the `sort` method. This method takes two parameters as input. The first parameter specifies the field on which the sort must be performed and the second parameter specifies the sort order (1 for ascending order and -1 for descending order). The default is ascending order.

Code Snippet 7 lists the code to perform this action:

Code Snippet 7:

```
from pymongo import MongoClient
import pprint
client = MongoClient()
db = client.library
library_details = db.library_details
for lib in
library_details.find().sort("book_name", -1):
print(lib)
```

This code must be saved in a Python file. For example, let us enter the code in a file named as `python_librarySort.py` and save it to the `C:\Python` folder. To execute the code in the `python_librarySort.py` file, at the command prompt, run the command as:

```
python C:\Python\python_librarySort.py
```

The command executes as shown in Figure 12.10.

```
PS C:\Users\Linda> python C:\Python\python_librarySort.py
{'_id': ObjectId('64a28b6a8985b8c5897f682c'), 'book_id': '1019', 'book_name': 'Python for Data Analysis', 'book_author': 'Wes McKinney', 'volume': 1}
{'_id': ObjectId('64a28b6a8985b8c5897f682b'), 'book_id': '1010', 'book_name': 'Python Programming', 'book_author': 'John M Zelle', 'volume': 3}
{'_id': ObjectId('64a28b6a8985b8c5897f682d'), 'book_id': '1009', 'book_name': 'Python Cookbook', 'book_author': 'David Beazley', 'volume': 3}
```

Figure 12.10: Sort Documents

Note that the documents fetched are sorted in descending order of the `book_name` field.

12.4.7 Update a Document

Consider that the user wants to update the value in the `volume` field as 4 for the document where the `book_name` is `Python Programming`. This can be done using the `update_one` method. This method takes two parameters as input. The first parameter specifies the document that must be updated, and the second parameter specifies the new values.

Code Snippet 8 lists the code to perform this action:

Code Snippet 8:

```
from pymongo import MongoClient
import pprint
client = MongoClient()
db = client.library
library_details = db.library_details
myquery = { "book_name": "Python Programming" }
updatevalue = { "$set": {"volume":4} }
library_details.update_one(myquery, updatevalue)
for lib in library_details.find():
    print(lib)
```

This code must be saved in a Python file. For example, let us enter this code in a file named as `python_libraryUpdate.py` and save it to the `C:\Python` folder. To execute the code in the `python_libraryUpdate.py` file, at the command prompt, run the command as:

```
python C:\Python\python_libraryUpdate.py
```

The command executes as shown in Figure 12.11.

```
PS C:\Users\linda> python C:\Python\python_libraryUpdate.py
{'_id': ObjectId('64a28b6a8985b8c5897f682b'), 'book_id': '1010', 'book_name': 'Python Programming', 'book_author': 'John M Zelle', 'volume': 4}
{'_id': ObjectId('64a28b6a8985b8c5897f682c'), 'book_id': '1019', 'book_name': 'Python for Data Analysis', 'book_author': 'Wes McKinney', 'volume': 1}
{'_id': ObjectId('64a28b6a8985b8c5897f682d'), 'book_id': '1009', 'book_name': 'Python Cookbook', 'book_author': 'David Beazley', 'volume': 3}
```

Figure 12.11: Update a Document

Note that the value of the `volume` field has changed to 4 for the document where the `book_name` is `Python Programming`.

12.4.8 Delete a Document

Consider that the user wants to delete a document in a collection where the `author_name` is `'David Beazley'`. The `delete_one` method in MongoDB can be used to delete a document. The query object is passed as the first argument to the `delete_one` method.

Code Snippet 9 lists the code to perform this action:

Code Snippet 9:

```
from pymongo import MongoClient
import pprint
client = MongoClient()
db = client.library
library_details = db.library_details
myquery = {"book_author": "David Beazley"}
library_details.delete_one(myquery)
for lib in library_details.find():
    print(lib)
```

This code must be saved in a Python file. For example, let us enter this code in a file named as `python_libraryDelete.py` and save it to the `C:\Python` folder. To execute the code in the `python_libraryDelete.py` file, at the command prompt, run the command as:

```
python C:\Python\python_libraryDelete.py
```

The command executes as shown in Figure 12.12.

```
PS C:\Users\linda> python C:\Python\python_libraryDelete.py
{'_id': ObjectId('64a28b6a8985b8c5897f682b'), 'book_id': '1010', 'book_name': 'Python Programming', 'book_author': 'John M Zelle', 'volume': 4}
{'_id': ObjectId('64a28b6a8985b8c5897f682c'), 'book_id': '1019', 'book_name': 'Python for Data Analysis', 'book_author': 'Wes McKinney', 'volume': 1}
```

Figure 12.12: Delete a Document

Note that the document where the `author_name` is `David Beazley` is deleted from the collection.



The `library_details.delete_many({})` command can be used to delete all the documents from the `library_details` collection.

12.5 Summary

- The combination of Python, a versatile programming language, and MongoDB, an extremely scalable and reliable database, forms a robust database application.
- PyMongo is MongoDB's official Python driver that helps to create a connection between Python and MongoDB.
- PyMongo can be installed using the command `pip install PyMongo`.
- After establishing the connection between Python and MongoDB, databases and collections can be created in MongoDB using Python.
- The collections in the MongoDB database can be queried to retrieve either a document or all the documents in the collection.
- The results of a query can be filtered using the comparison or Boolean operators.
- The results of a query can be sorted using the `sort` method.
- The documents in a MongoDB collection can be updated or deleted using Python by specifying conditions.

Test Your Knowledge

1. Which driver should you install to interact with the MongoDB database through Python?
 - a. PythonMongo
 - b. PyMongo
 - c. MongoPy
 - d. Phmongo

2. Which of the following class of the PyMongo driver is used to connect Python with MongoDB?
 - a. ConnectMongo
 - b. MongoConnection
 - c. PythonClient
 - d. MongoClient

3. Which of the following code will allow you to create a PyMongo client?
 - a. mongod = MongoClient()
 - b. server = ClientMongo()
 - c. client = MongoClient()
 - d. client = connectMongo()

4. Consider that you have connected Python with MongoDB using the PyMongo driver. Which of the following code allows you to create a database named employee?
 - a. db = client.employee
 - b. client = db.employee
 - c. db = MongoClient.employee
 - d. client= db.client.employee

5. Which of the following methods are used to return all documents from a collection?
 - a. find()
 - b. findAll()
 - c. find({ })
 - d. findMany()

Answers to Test Your Knowledge

1	b
2	d
3	c
4	a
5	a, c

Try it Yourself

1. Install the PyMongo driver to connect Python with MongoDB.
2. Create a Python file `connectMongo.py` and perform the given tasks:
 - a. Import a `MongoClient` class from PyMongo driver.
 - b. Create a `MongoClient` instance.
 - c. Create a database named `Customer_purchase` and a collection named `Purchase_detail`. Three documents in the collection are:

```
[  
    {  
        Cust_id:94608  
        Product_id:112  
        Cust_name:Adam Richard  
        Purchase_product:Laptop  
        Discount:0.05  
    },  
    {  
        Cust_id:94609  
        Product_id:118  
        Cust_name:Michael Faraday  
        Purchase_product:Tablet  
        Discount:0.1  
    },  
    {  
        Cust_id:94602  
        Product_id:103  
        Cust_name:Richard David  
        Purchase_product:Smartwatch  
        Discount:0.05  
    }  
]
```
 - d. Write Python code to insert the given documents into the `Purchase_detail` collection.
3. Execute the Python file `connectMongo.py`.
4. Connect to Mongo shell to view the `Customer_purchase` database and the `Purchase_detail` collection.
5. To query the documents, write Python code for each of the given tasks and execute the Python codes to view the results.
 - a. Find the documents where the `Discount` value is equal to 0.5.
 - b. Find all the documents in the collection `Purchase_detail`.
 - c. Update the `Purchase_product` of 'Richard David' as 'Smartphone'.

Appendix

Sr. No.	Case Studies
1.	<p>In a supermarket, the daily sales of products are stored in a MongoDB database named <code>product_inventory</code>. At the end of each day, the manager of the supermarket must analyze sales of each product based on the data stored in the <code>product_inventory</code> database.</p> <ul style="list-style-type: none"> a. Create the <code>product_inventory</code> database with a collection named <code>product_sales</code>. b. Create a user named <code>manager</code> for the <code>product_inventory</code> database and grant the <code>readwrite</code> role to <code>manager</code> on the <code>product_inventory</code> database. c. Authenticate <code>manager</code> on the <code>product_inventory</code> database and insert the given six documents into the <code>product_sales</code> collection. <pre>[{ cust_id:1001 cust_name: "John", cust_city:"Atlanta " current_purchase:"Baby Food", quantity:3, unit_price:300 last_purchase:["dairy", "grocery", "snacks"] } { cust_id:1006 cust_name: "Alexander", cust_city:"Boston" current_purchase:"diary", quantity:4, unit_price:250, last_purchase:["care_products", "snacks"] } { cust_id:1003 cust_name: "Gabriel", cust_city:"Los Angeles" current_purchase:"care_products", quantity:7, unit_price:500, last_purchase:["dairy", "dried goods"] } { cust_id:1004 cust_name: "Williams", cust_city:"Texas" }</pre>

```

        current_purchase:"snacks",
        quantity:2,
        unit_price:250,
        last_purchase:["grocery", "diary"]

    }
{
    cust_id:1007
    cust_name: "Nicholas",
    cust_city:"Boston"
    current_purchase:"Baby Food",
    quantity:5,
    unit_price:150,
    last_purchase:[ "Fruits", "snacks"]
}
{
    cust_id:1008
    cust_name: "Richard",
    cust_city:"Austin"
    current_purchase:"diary",
    quantity:3,
    unit_price:250,
    last_purchase:[ "Fruits", "vegetables"]

}
]

```

- d. Create an index for `product_sales` collection on the `cust_id` field in ascending order.
- e. Fetch only the documents where the customer belongs to the Texas city.
- f. Fetch the city details of all the customers who have currently purchased only dairy products.
- g. Add a new field named `Date_of_purchase` which shows the current date to all the documents.
- h. Fetch details of all the customers who purchased `grocery` in their last transaction.
- i. Exclude the `_id` field and display all the documents of the `product_sales` collection which include only the `cust_name`, `cust_city`, `current_purchase`, and `quantity` fields.
- j. Retrieve the distinct `current_purchase` values from the `product_sales` collection.
- k. Calculate the total quantity of sales per `current_purchase` item and return only the `current_purchase` item with a total quantity greater than or equal to 5.

	<p>i. Calculate the total price ($\text{unit_price} * \text{quantity}$) of sales per <code>current_purchase</code> item for all the documents in the <code>product_sales</code> collection.</p> <p>m. Count the number of customers who have purchased 'snacks' as one of the items in their last transactions.</p>
2.	<p>Consider that the supermarket has three branches all over the country and maintains the data of all three branches on a single server. When the data is accessed from different branches, in order to ensure the high availability of data, create a replica set(P-S-S) in MongoDB where a primary server has two secondary members.</p> <p>a. In the primary server, create a database with the name <code>product_inventory</code> that contains a collection named <code>product_sales</code>. Insert the six documents into <code>product_sales</code> collection and check whether the <code>product_inventory</code> database is replicated in the secondary servers.</p> <p>b. In the primary server, create an index for the field <code>cust_id</code>. Start a session and start a transaction to update the <code>current_purchase</code> field of the customer with <code>cust_id</code> as 1008 to care_products. Commit this transaction and check whether the update operation executed in this transaction is visible in the secondary servers.</p>
3.	<p>Manager of a supermarket wants to analyze and visualize the data in the <code>product_sales</code> collection by generating reports using Business Intelligence (BI) tools. To do this, use MongoDB compass to connect the <code>product_inventory</code> database to Microsoft Excel using Open Data Base Connectivity (ODBC). Import data from the <code>product_sales</code> collection to Microsoft Excel.</p>